Crossword Compiler-Compilation

H. BERGHEL[1] AND C. YI[2]

[1]*Department of Computer Science, University of Arkansas, Fayetteville, AR 72701, USA*
[2]*Computer Science Program, University of Missouri – Kansas City, Kansas City, MO 64110, USA*

We describe a procedure, which we refer to as* crossword compiler-compilation, *which will create source code for a crossword compiler from the puzzle geometry alone. This procedure complements earlier results of ourselves and others which automate only the latter stages of compilation.

Received May 1988, revised July 1988

1. INTRODUCTION

'Crossword compilation' is the term used by Smith and Steen[10] to refer to the stages of generating crossword puzzles. In an earlier paper,[2] we identified six activities related to crossword compilation in so far as the automation of the compilation is concerned:

(1) creation of the host matrix,
(2) determination of the overall design of the matrix,
(3) specification of word slots,
(4) identification of shared cells,
(5) construction of one or more solutions,
(6) composition of a clue set for each solution.

From the point of view of the crossword compiler (man or machine), the first four activities are seen to be largely technical issues dealing with the design and complexity of the resulting puzzle. (5) is basically a methodological issue, having to do with the mechanics of solving crossword puzzles. (6) is the 'human factors' dimension of puzzle compilation, which integrates a variety of topics in psychology, linguistics and artificial intelligence. In addition, there are a host of aesthetic issues which involve all of the above.

The literature on automated crossword compilation initially dealt with the construction of solution sets, which we find to be the most interesting issue from a computational point of view. At this writing, a dozen or so papers have been published on this subject (see references below), including an earlier work of ours.[2] The art of mechanised clue set construction has also received attention, primarily in the work of G. Smith and J. duBoulay[9] (see also Ref. 11). The relationship between the general issue of puzzle aesthetics (i.e. what constitutes a 'good' puzzle) and the automation of crossword compilation has been dealt with by P. Smith.[11] However, the technical issues have not, so far as we can determine, received much attention. We propose to discuss one approach toward the automation of (1) to (4) in this paper. Specifically, we describe a procedure which produces a program which generates the solution set for a puzzle from the geometrical specification of that puzzle. We refer to such procedures as *crossword compiler-compilers*.

2. OVERVIEW

For the discussion which follows, we place certain restrictions on the type of crossword puzzles to be considered. First, we assume that all geometrical patterns of the puzzle be defined within a host rectangular matrix. Non-rectangular geometrical forms must be created within the rectangle by creating pattern borders with closed (black) cells. Secondly, we define a word slot to be a linear arrangement of open (non-black) cells, bounded by either the puzzle border or closed cell. Thirdly, only orthogonal cell sharing or interlocking is allowed. Fourthly, orthogonal interlocking is assumed to apply to all open cells for which it is appropriate. In addition, we follow the custom of avoiding solutions which contain more than one occurrence of a given word.

The first restriction has little practical significance, but does affect the way in which we define the problem space. The program which we describe below will work for any arbitrary puzzle, regardless of external shape or internal design, so long as the puzzle fits within a 25 × 25 matrix. The second restriction prevents consideration of puzzles where two word slots abut each other in the same direction. The third restriction limits the range of word slot linking to the most common, orthogonal type. Finally, our program will assume that complete, orthogonal interlocking is desired if possible. These last three restrictions are specifically included to produce the types of puzzles normally found in America. They would be relaxed to accommodate the British-type puzzles. Such relaxation would involve trivial modifications of the code.

Our general orientation toward solving crossword puzzles has been discussed elsewhere.[2] As a result, we will limit the present discussion to the basic issues.

In our opinion, the most intuitive description of the problem is one which is based upon first-order logic. That is, we see the solution of puzzles (ignoring clues, for the moment) as essentially inferential: one infers the solutions from the puzzle geometry, the 'rules' of the game and a lexicon. Each of these parameters is axiomatised. Then the axiom system may be run through an inference engine (e.g. Prolog), the consequences of which are solutions to the puzzle. Were we interested in solving the puzzle based upon the clue set, we would add additional sets of axioms relating to clues. We note that in the absence of a fixed corpus of rules governing clue generation, this would be a non-trivial activity!

We may best illustrate our approach by reference to Fig. 1. The existence of a solution implies the truth of a series of existential claims. Specifically, let the universe of discourse be characters of the Roman alphabet. If a solution to the puzzle exists, then it must be the case that

$$(\exists x,y_1,y_2,y_3,z_1,z_2,z_3,z_4)\quad (W(x,y_1,y_2,y_3)\,\&\,W(x,z_1,z_2,z_3,z_4))$$

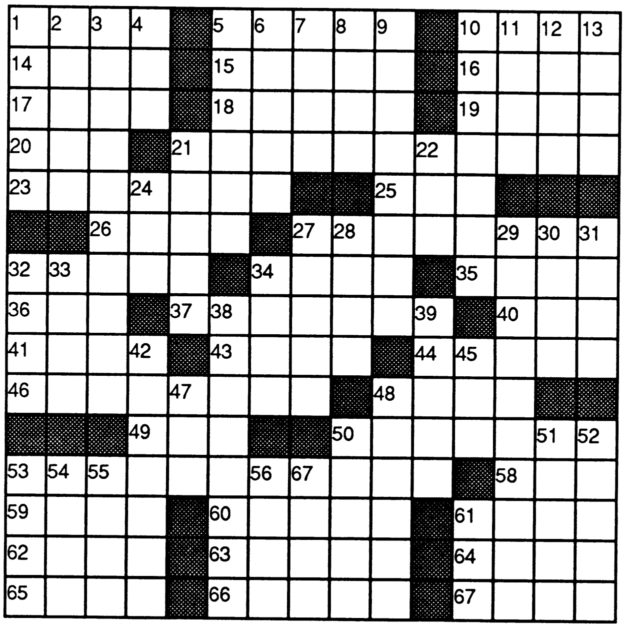


Figure 1. Layout of typical American crossword puzzle.

where the quaternary and quinary predicates, $W/4$ and $W/5$, mean 'is a 4-letter word' and 'is a 5-letter word', respectively, and refer to the word-slots 1 across and 1 down. By generalising the individual variables to $x_{i,j}$, $1 < i, j < 15$, where i and j refer to the column and row coordinates of the puzzle matrix, we may represent the entire puzzle as an existentially quantified, prenexed conjunction with apparent variables corresponding to the open (non-black) cells are taken from the set $\{x_{1,1},\ldots,x_{15,15}\}$ and where each conjunct is an n-ary predicate, for some individual word-slot of length n. Inasmuch as the formula is prenexed, and the common cell variables are shared by interlocking word slots, word insertions which create impossible interlocking situations will be detected and avoided on backtracking.

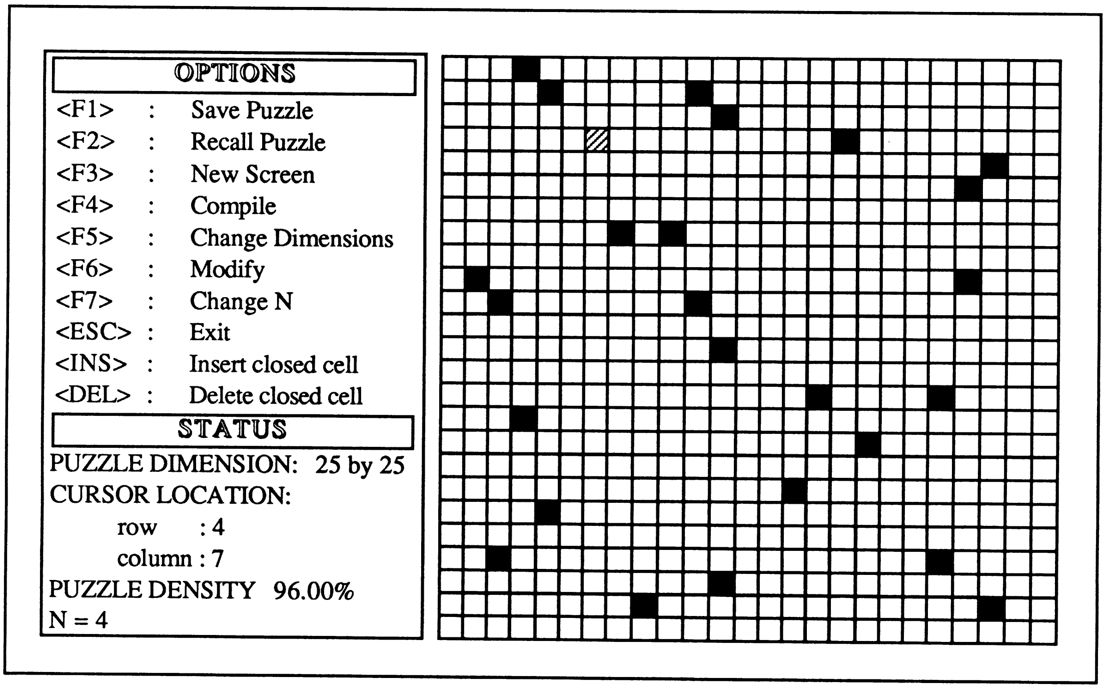
This approach involves what Mazlack[7,8] calls a 'whole word' search. However, unlike Mazlack's unsuccessful attempt,[7] our method does support a full range of character-level heuristics. For example, by inserting goals of the form $(\exists x_1,\ldots x_k)$ $(W(x_1,\ldots,x_k,\_,\ldots,\_))$, as intermediate conjuncts (where '_' is interpreted as a 'don't care' variable), one may avoid the inherent problem of exhaustive search. In this way, the intermediate conjuncts 'look ahead' to interlocking word slots to ensure that the word insertions do not create impossible combinations of characters for word slots to be filled in later. Without such a mechanism, the fail points would appear too far down in the search tree. For crossword solution, the most challenging aspect of the program design involves such heuristics. Just as a viable crossword compiler must include reasonable heuristics, so the crossword compiler-compiler must be able to generate them.

In the sections to follow we describe one approach toward the automation of those elements of crossword compilation which deal with the technical issues and which will culminate in the creation of a program which will generate clue-independent solutions to arbitrary crossword puzzles based entirely upon puzzle geometry.

3. CROSSWORD-COMPILER COMPILATION

Given the restrictions mentioned in the previous section, the specification of word slots and the identification of interlocked cells is a consequence of the puzzle geometry. Thus, of the original technical issues, only the size of the host matrix and the internal design (distribution of closed (black) cells) is required as input. As Fig. 2 shows, our program interface has been created with this in mind. Our objective is to create the source code for an efficient and effective puzzle solution generator from this input, alone. Were it not for the heuristics, compiler compilation would amount to a simple sequence of lexical searches.

The necessary heuristics appear at two levels. At the



OPTIONS

<F1>	:	Save Puzzle
<F2>	:	Recall Puzzle
<F3>	:	New Screen
<F4>	:	Compile
<F5>	:	Change Dimensions
<F6>	:	Modify
<F7>	:	Change N
<ESC>	:	Exit
<INS>	:	Insert closed cell
<DEL>	:	Delete closed cell

STATUS

PUZZLE DIMENSION: 25 by 25
CURSOR LOCATION:
row : 4
column : 7
PUZZLE DENSITY 96.00%
N = 4

Figure 2. Display.

lower level we are concerned with intelligent backtracking control. This control may be exercised by means of judicious applications of extra-logical control predicates (e.g. cut and snip). At the higher level, that of program organisation, we must concern ourselves with strategic issues which involve the very orientation which we take toward the problem. Any crossword compiler-compiler worthy of the name must create source code with both levels of control built in.

While backtracking control is a *sine qua non* in efficient program design, there is not much that may be said of it which has enduring importance. Unfortunately, such mechanisms (at least in the Prolog family) have a 'fuzzy' semantics, due primarily to the lack of a single language standard at the moment, which creates inconsistencies between implementations. Further, while such control constructs have no effect on the declarative semantics of program, they do alter the procedural semantics. The reader should be forewarned that there are troublesome, although possibly ephemeral, issues involved in backtracking control with which we shall not deal. Instead, we refer the reader to a standard text on the subject[12] and an earlier report of ours.[3]

The example of heuristics which we gave in the previous section illustrates an overall strategy. As we mentioned, the goal is to raise the fail points as high in the search tree as possible. However, the particular example which we gave has two shortcomings: it does not take into account non-rectangular geometries, and it fails to address the fact that the distribution of word lengths in an actual dictionary is not uniform (see Fig. 3). Thus further enhancement is called for.

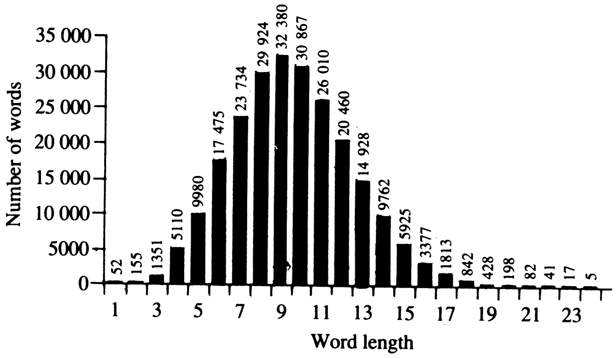


Figure 3. Distribution of lexicon by word length.

4. STRATEGY

As any crossword puzzle enthusiast will avow, the intuitive idea behind solving a subcomponent of the puzzle is to rule out impossible combinations of interlocking words as soon as possible. To illustrate, consider the full (i.e. 100% dense with respect to open cells) puzzle in Fig. 4. Were we to insert words 1A (across), 2A, 3A, 1D (down), 2D and 3D, in that order, we would have no fail point before the fourth insertion. On the other hand, the sequence 1A, 1D, 2A, 2D, 3A, 3D would have two fail points prior to the fourth insertion. The latter strategy is clearly more effectual.

In general, if $f(i)$ is the number of fail points (i.e. interlocking cells) generated at stage i of the insertion sequence, and $c(i) = \Sigma_{j=1}^{i} f(j)$, is the cumulative number of fail points at stage i, then we will aways prefer, other things being equal, those insertion strategies which maximise $c(i)$, for all stages, i. In the case of the puzzle in Fig. 4, the sequence of values of $c(i)$ for the first strategy is $\langle 0, 0, 0, 3, 6, 9 \rangle$, while for the second strategy the values are $\langle 0, 1, 2, 4, 6, 9 \rangle$. Clearly, the advantage lies with the approach which alternates between across and down words.

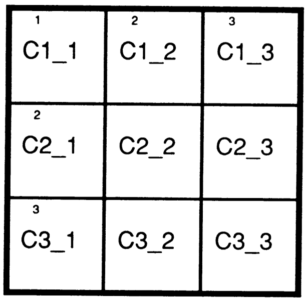


Figure 4. Full puzzle.

However, the algorithm to maximise $c(i)$ is straightfoward only if the open cells fall within a rectangle. Generalising to non-rectangular sub-puzzle geometries would not gain much, because of another mitigating circumstance: some word slots are easier to fill than others because of their length. Thus, what is called for is an algorithm which uses the alternating approach to word slot insertion (i.e. tries to maximize $c(i)$) and, at the same time, takes advantage of the effect of a non-uniform distribution of word lengths. We call our algorithm the 'highest priority neighbourhood' approach.

The algorithm works in the following way. First, we assign to all word lengths a priority according to the frequency distribution of the lexicon. Currently, we use unweighted prioritisation, which is inversely related to the frequency of occurrence. That is, if $r(x)$ is the rank order of word length, x, and N is the number of word lengths represented, then $p(x) = N - r(x)$. Of course, the intuitive idea is that we prefer to work first with the word slots which will be hardest to fill. A prioritisation table allows us to choose the 'rarest' of those which intersect the current word slot 'on the fly'. So, with regard to Fig. 3, $p(24) = 24$ while $p(9) = 1$, etc.

Next, we identify the entry point into the puzzle by selecting the word slot with the highest priority (the default for ties is the slot with the lowest row/column coordinate in the first position). Then we invoke an N-level look-ahead procedure, where $N \geqslant 1$ is selected by the user, to determine the next sequence of slots to fill, alternating horizontally and vertically as we go.

We have approached the look-ahead procedure from two different points of view: exhaustive and focused. In the exhaustive case, we determine the priorities of all word slots which intersect our target, then the priorities of all word slots which intersect the intersecting word slots, and so forth, for the next N levels. The preferred path is defined as the one with the highest aggregate priority (e.g. simple summation). Since the procedure is exhaustive, the time complexity should be somewhere between factorial and exponential. For the puzzle in Fig. 1, 40, 51, 99, 286 and 991 seconds were required for compilation when N was set to 1, 2, 3, 4 and 5, respectively. Higher values of N caused heap/stack collision. Of course, the user interface is essential here, for it allows the user to control the progressive deepening of the search by varying the value of N. The advantage of the exhaustive approach is that the search sequence for m words slots is optimal with respect to the frequency distribution of word slots when $N = m$.

In practice, however, we opt for a more focused approach. In this case, we commit to one word slot at each level, based upon priority, and only then move on to the next level. This modification yields linear performance. Since the exhaustive approach is generally not optimal for $N < m$, and since the performance degenerates radically as N approaches m, we feel that, on balance, we haven't lost much with the focused approach. We have found that for $N = 5$, run times of the 'compiler-compiler' are insignificant (e.g. 2 minutes on an 80286-class microcomputer for the puzzle in Fig. 1), and the resulting heuristics on a par with those done by hand (i.e. the run times of the solution generator produced by our program are not consistently superior or inferior to those which we produced manually).

5. OPERATION

Crossword-compiler compilation begins with the geometrical information recovered from the display memory (see Fig. 2). This figure shows the screen image after 4% of the cells have been filled. Cursor positioning (the cell shaded with diagonal lines) is by 'arrow' keys or mouse. Cells are closed or opened with the ⟨INS⟩ and ⟨DEL⟩ keys whenever the program is in the 'modify' (⟨F6⟩) mode. ⟨F4⟩ is the 'hot' key, causing the crossword-compiler compiler to be invoked on the basis of the current geometry. The remaining functions are self-explanatory.

The first step in compilation involves the creation of two lists, one each for all of the across and down words. Each entry consists of a word-slot identifier and the priority value discussed in the previous section. The creation of the sequence of word fillings continues until both of these lists are empty.

For any given step, the determination of the appropriate 'next word' causes two disc files to be updated with procedures for search as well as for backtracking control. To illustrate, consider the puzzle in Fig. 5. Were the next word to be filled ⟨C2_2,C2_3⟩, the following clause fragments would be inserted in these files:

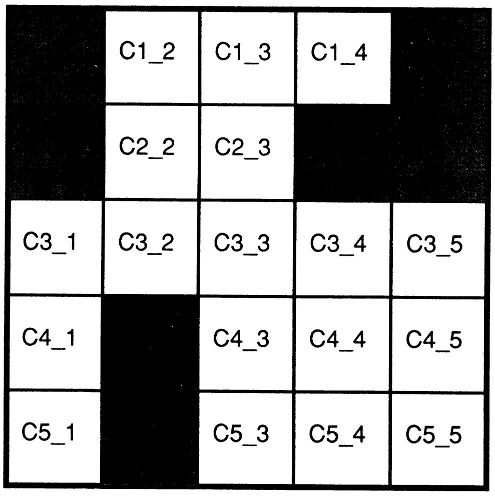


Figure 5. 72% dense puzzle on 5 × 5 matrix.

File A

```
recorded(foundword,word(C2_2,C2_3),_),
not(member([C2_2,C2_3],LISTIN),
heuristic(C2_2,C2_3),
append(LISTIN,[C2_2,C2_3],LISTOUT)
...
```

File B

```
heuristic(C2_2,C2_3):-
recorded(foundword,word(_,C2_2,_),_),!,
recorded(foundword,word(_,C2_3,_,_,_),_),!.
```

Of course, file A is in fact a single Prolog procedure, where all variables contained within are bound by the same quantifier. File B, however, consists of a set of clauses, one for each word slot, which represents the heuristics. Once invoked and satisfied, their variable bindings are irrelevant. The varying arities for the 'word' predicate name correspond to the varying-length word slots which intersect our target (down words, ⟨C1_2,C2_2,C3_2⟩ and ⟨C1_3,C2_3,C3_3, C4_3,C5_3⟩, in our example). Including this strategy prevents us from moving ahead with the solution if an impossible situation has been created. The 'not-member' predicate is used to ensure that no puzzle has more than one occurrence of a single word. Both of these sets of procedures are consistent with the restrictions discussed in Section 2, above.

Our program makes a few modifications to the resulting code which are pragmatically motivated. First, in the current version of the program we represent all of the database clauses in list form, where the heuristic predicates are elements of sublists. That is, the program appears as [..., word$_i$, [heuristic$_{i-1}$, ..., heuristic$_{i-n}$], ...] rather than as two sets of procedures, integrated through calls. Ignoring boundary conditions, the main procedures than take on the following form:

```
list(X|Y],L1):-   legitimate_word(X),
                  not (member (X, Y)),
                  heuristic (Y, X, L1).
heuristic ([X|Y], Z, L1):-check (X)
                          append ([Z], L1, L2),
                          list (Y, L2).
check ([X: Y]):-legitimate_word(X),
                !,
                check(Y).
```

In this way, we process the entire list with few recursive procedures and avoid the overhead associated with the unification of variables between separate procedures (i.e. those with search vs. heuristic predicates). Further, we generate a complete set of compiler directives. Such modifications are motivated by concern for performance and implementation limitations (e.g. maximum number of predicates per procedure which are supported), and have no effect on the declarative reading of the program. The output is the complete source code for a Prolog program, which generates one or all of the solution sets with respect to the given geometry of the puzzle. The code may then be compiled and linked with the desired lexicon.

6. CONCLUSION

As we mentioned earlier, previous work on crossword compilers has largely ignored the technical issues associated with the actual construction of the compiler. In this context, automation is primarily associated with the construction of the solution sets or the generation of clues. In this paper we extend the automation to the compiler itself. In general, this process, which we call crossword compiler-compilation, involves the automatic creation of crossword compilers from geometrical specifications of the puzzle. This work can be seen as a natural extension of earlier work. Building upon our understanding of human crossword compilation, earlier work dealt with the automation of the process. Our present goal is to automate the very process of automation, based upon our understanding of the underlying principles. In particular, our approach views the automation of crossword compilation as a two-stage endeavour, involving both a logical description of the problem and an understanding of the heuristics which affect performance.

The crossword compiler-compiler which we described above is implemented in Pascal for the IBM family of microcomputers. The resulting compiler is in Prolog. The source code is created with Arity Prolog in mind, although with the exception of the compiler directives and a few minor syntax changes, the program should compile with almost any Prolog which supports Edinburgh syntax. As we argued before,[2] we find the logical representation of the puzzle to be the most direct and intuitive approach to crossword compilation. The Prolog module will work with any lexicon, formatted for Prolog, so long as the size does not exceed system limitations. Currently, we are experimenting with several dictionaries of 10,000–250,000 words.

Our interest in crossword compilation is derived from the fact that it involves an interesting application of approximate string matching. In many ways, it is similar to the problem of spelling correction in electronic documents.[1] As a result, our focus is slightly different from that of other researchers who are interested in puzzle compilation as such and in general.

One of the most noticeable differences involves aesthetic issues. As Smith demonstrates,[11] the quality of individual clues and cohesiveness of the puzzle are both important determinants in the entertainment value of the puzzle. We have heretofore ignored such issues, for they involve lexicographical considerations which fall outside our problem domain. For the moment we must consider our puzzles as a superset of those puzzles which are interesting to the cruciverbalist.

Acknowledgements

The authors wish to thank R. Hetherington, R. Rankin, P. Smith and an anonymous referee of *The Computer Journal* for their invaluable comments on earlier drafts of this paper.

REFERENCES

1. H. Berghel, A logical framework for the correction of spelling errors in electronic documents. *Information Processing and Management* **23**, (5), 477–494 (1987).
2. H. Berghel, Crossword compilation with horn clauses. *The Computer Journal* **30** (2), 183–188 (1987).
3. H. Berghel and R. Rankin, *Functional Uncertainty in Transparent, Translucent and Opaque Programming Contexts in Prolog. Proceedings of the Second Oklahoma Conference on Artificial Intelligence*, Norman, November, 1988, pp. 129–143.
4. L. Mazlack, An investigation of computer generated crossword puzzles. *Ph.D. Dissertation*, Washington University, St Louis (1972).
5. L. Mazlack, Data structures required for crossword puzzle construction. *Proceedings of the 36th Annual Meeting of the American Society for Information Science*, pp. 141–142 (1973).
6. L. Mazlack, 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).
7. L. Mazlack, Computer construction of crossword puzzles using precedence relationships. *Artificial Intelligence* **7**, (1), 1–19 (1976).
8. L. Mazlack, Machine selection of elements in crossword puzzles: an application of computational linguistics. *SIAM Journal on Computing* **5**, (1), 51–72 (1976).
9. G. Smith, and J. duBoulay, The generation of cryptic crossword clues. *The Computer Journal* **29**, (3), 282–284 (1986).
10. P. Smith and S. Steen, A prototype crossword compiler. *The Computer Journal* **24**, (2), 107–111 (1981).
11. P. Smith, XENO: computer-assisted compilation of crossword puzzles. *The Computer Journal* **26** (4), 296–301 (1983).
12. L. Sterling and E. Shapiro, *The Art of Prolog*. MIT Press, Cambridge, MA (1986).
13. P. Williams and D. Woodhead, Computer assisted analysis of cryptic crosswords. *The Computer Journal* **22**, (1), 67–70 (1979).