Computer Construction of Crossword Puzzles using Horn Clauses and Constraint Programming

Yvette Graham & Carl Vogel[*]
O'Reilly Institute
Computational Linguistics Laboratory
Trinity College
Dublin 2
Ireland
{grahamy,vogel}@tcd.ie

Abstract

Improvements on existing methods of automatically generating crossword puzzles are presented. The system builds on recent work which generates Prolog programs, and resolves the constraints inherent in a grid layout and word selection. Novel results described here reside in the optimisation of the automatically generated Prolog clauses.

1 Introduction

Crosswords provide an interesting domain of study in constraint reasoning as well as a pleasant diversion for humans who attempt to solve them. From

[*]The authors gratefully acknowledge funding from the Intel Corporation and Trinity College, University of Dublin.

both perspectives they represent constraint satisfaction problems: setting a puzzle involves settling on a grid (typically symmetric on one or more axes) and a set of words that populate the grid; generally, the more engaging puzzles involve thematic relations among the clues for the words that populate the grid, and correspondingly the words themselves. This paper reports on advances in the automatic generation of both grids and population of them. Specifically, this is not about automated crossword solving, but crossword construction. While evidently humans find the task of generating grids and finding mutually compatible wordlists a tractable task (possibly involving a large degree of creativity), approaching the problem algorithmically quickly encounters intractability. The research reported on here extends on past work by Berghel (1987) and Berghel and Yi (1989) as well as more recent work by Gibbons (2002) and Graham (2003). It focusses on the issue of constraining a grid with the proposed solution (refer to Gibbons (2002) for the primary description of the algorithms used to set symmetric grids automatically). The problem is clearly one of simultaneous constraint satisfaction, and the novelty of the research is partly in the fact that the constraints are tested using Prolog code that is generated automatically by a suite of programs implemented in Java. The main advances here are optimizations in automatically generated code, code that has better performance properties than that reported by Berghel (1987) or by Gibbons (2002). The paper proceeds as follows: firstly, past work in the area of crossword generation is described; secondly, iterative improvements on that work achieved within the reported research are described; finally, thoughts about future directions are considered (these include potential for lexicon variation and alternative constraints on grid symmetry definitions).

2 Starting Point

Berghel (1987) broke the problem of computer construction of crossword puzzles into the following 6 steps:

1. creation of the host matrix
2. determination of the overall design (i.e. pattern of the open and closed cells) within a 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

This decomposition requires some explanation of terminology. The host matrix is simply the grid which is to be populated by words that the solver is intended to identify from the clues. An open cell is one that may host a letter, and a closed cell is one that is blackened out. An open cell may be shared between more than one word, and if it is shared it constitutes a shared cell. Individual words are represented as word slots — lists of variables, one for each letter in the word (and thus shared variables encoded constraints from one word upon intersecting words) — each word slot corresponds to one of the answers to be supplied by the solver within the grid. The degree of interlocking associated with a puzzle is related to its difficulty. Ironically, the higher the degree of interlocking, the harder it is to set the puzzle due to the quantity of constraints to be satisfied, and the easier it is to solve because of the partial information supplied towards some answers yielded by definite solutions to intersecting words. Again, however, we emphasize that we are not working on crossword solution, but crossword construction. Thus, in this paper we concentrate on step 5. The other aspects of the problem are addressed effectively (Gibbons, 2002; Graham, 2003), but are not relevant to the present discussion.

We simplify the problem furthermore by finding a single solution to the grid as opposed to all possible solutions see Berghel. Because of the enormity of the search space involved in jointly identifying an acceptable grid and populating it with words for a solver to discover, and therefore the usual exessive length of time it takes to construct a crossword, we thought it would be useful if we attempted to find the first solution in the fastest possible time. Our method takes a predefined grid as input, as well as a lexicon of words, which together are used to construct a single crossword puzzle.

2.1 Method

Berghel converted the problem of constructing the two-dimensional crossword puzzle into a linear list of predicative expressions. The conjunction of these

```
solution([SLOT_1, SLOT_2, SLOT_3, SLOT_4, SLOT_5, SLOT_6]):-

        word(C1, C2, C3, C4),
        name(SLOT_1, [C1, C2, C3, C4]),

        word(C5, C6, C7, C8),
        name(SLOT_2, [C5, C6, C7, C8]),

        word(C9, C10, C11, C12),
        name(SLOT_3, [C9, C10, C11, C12]),

        word(C13, C14, C15, C16),
        name(SLOT_4, [C13, C14, C15, C16]),

        word(C1, C5, C9, C13),
        name(SLOT_5, [C1, C5, C9, C13]),

        word(C2, C6, C10, C14),
        name(SLOT_6, [C2, C6, C10, C14]),

        word(C3, C7, C9, C15),
        name(SLOT_7, [C3, C7, C9, C15]),

        word(C4, C8, C12, C16),
        name(SLOT_8, [C4, C8, C12, C16]).
```

Figure 1: Berghel's Representation of a Crossword Grid

predicative expressions was then used as the body of a headed Horn clause, the head of the Horn clause being the unnegated literal which if suceeded would prove at least one solution existed to the given crossword grid. This Horn clause was expressed in Prolog as follows (see Fig. 1, a representation of the grid displayed in Fig. 2):

By translating the problem into Prolog Berghel took advantage of Prolog's built-in theorem prover. It works in a left-right depth-first manner on a set of Horn clauses like the one above. Since each distinct variable within the Horn clause is bound by the same implicit quantifier (named with a number corresponding to the number assigned to the cell in Figure 2), each cell of

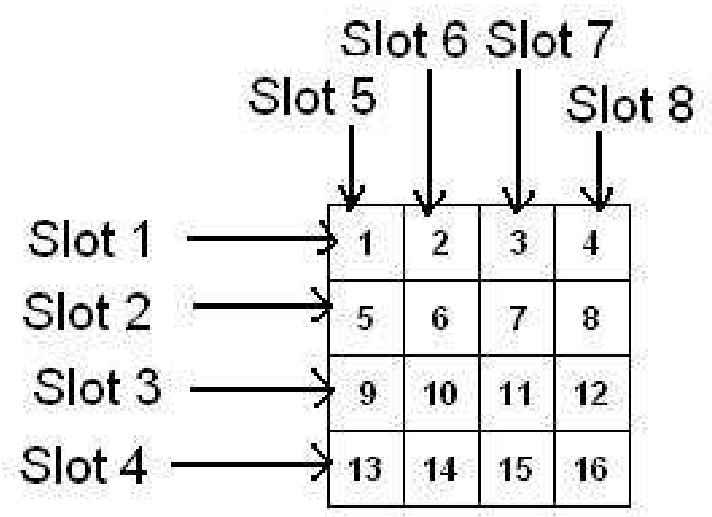

Figure 2: Grid Corresponding to Initial Prolog Code.

the grid that's part of two separate word slots (i.e. the common cell of two interlinking word slots) will only be instantiated to one character. The lexicon is encoded as a series of Prolog facts, ensuring that each word slot is only ever instantiated to a valid word that exists in the lexicon see below.

Prolog representation of the word 'ACE': `word(97,99,101).` The integers correspond to the ASCII encodings of the corresponding letters.

A great advantage of Berghel's method of constructing crossword solutions is its simplicity. The entire program consists of one prolog rule representing the structure of the grid and a series of prolog facts representing the words one wants to use to create a solution. One major disadvantage of this method however is the amount of time the program takes to execute. When the program executes at several different stages it backtracks, redoing the insertion of numerous words. Some of this backtracking is of course necessary to find a solution to the grid. However much of it is also unnecessary and therefore could be eliminated from the process reducing the execution time significantly. We decided to extend Berghal's method of crossword compilation by introducing constraints which manipulate the execution of the program to eliminate naive backtracking. Below are the constraints we put on the program:

1. Control the order in which the word slots are assigned words
2. Do not assign a word to a word slot that is already assigned to a different word slot
3. Do not assign a word to a word slot that is going to result in an imposssible combination of characters in interlinking word slots
4. When searching for a word to assign to a given word slot evaluate the prospective words using a heuristic value
5. Only redo the insertion of a given word slot if redoing that word slot affects the word slot that is causing the programm to backtrack

The section that follows describes why these constraints have such a great affect on the execution time of the program and how we went about implementing them.

2.2 Order of Word Slot Assignment

Controlling the order in which the word slots are assigned words. The order in which the word slots are filled is important. As an example take the grid structure described in figure 3. Consider if the grid's word slots were filled in in the following order $< 0, 2, 3, 6, 1, 4, 5 >$. If the words are filled in in this order the first point of failure will be when word slot 1 is assigned a word. It is important that if a word has been entered into a particular word slot, and this word will eventually cause a problem in the grid, that the code will fail as soon after this insertion as possible to avoid naive backtracking. If the words in an order which delays the first possible point of failure until the fifth inserted word slot, this will slow down finding the solution significantly. Thus we defined an algorithm, which to figure out the order in which the word slots should be filled up so that the number of early points of failure was maximized.

The first word slot to be filled in is the one with the highest degree of interlock e.g. word slot 1 above. Next are word slots that interlink the first

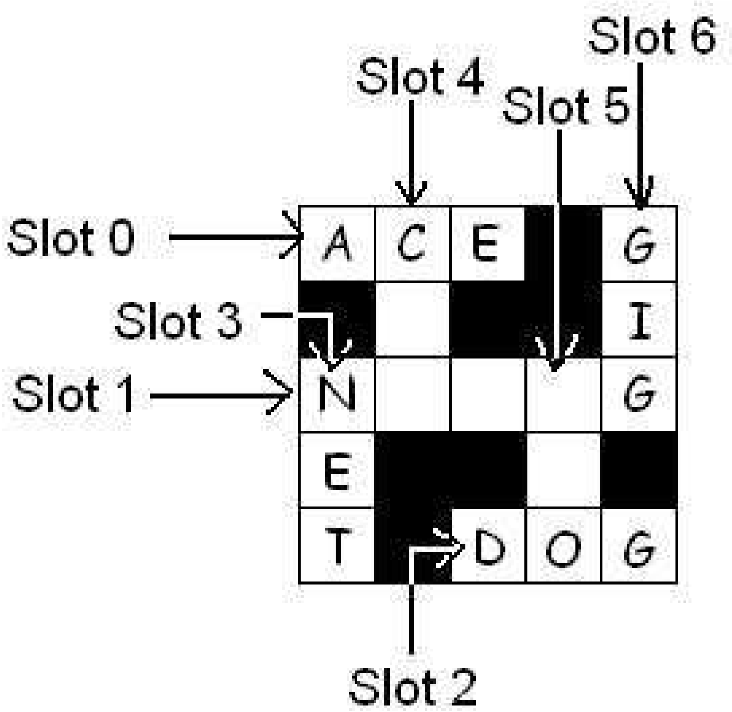

Figure 3: Grid Corresponding to Initial Prolog Code.

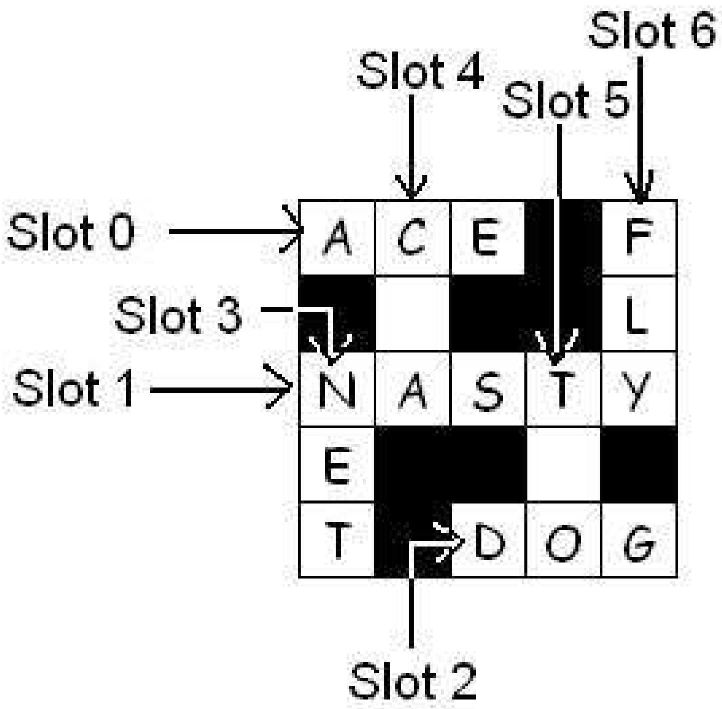

Figure 4: Initial Grid after Backtracking & Word Replacement.

word slot e.g. word slots 3, 4, 5 and 6 above. After that the next word slots that interlink the second word slot in the sequence so far i.e the word slots that interlink word slot 3. Added to the end of the sequence then are all the word slots that interlink word slot 4, then 5 etc. The sequence of word slots is traversed adding all of the interlinking word slots to the end of the sequence as we go until all of the word slots of the grid are in the list. Thus the order of the grid in figures 3 would be: $< 1, 3, 4, 5, 6, 0, 2 >$.

If the word slots are filled in in this order the first point of failure will be when the second word slot is assigned a word. Also it is worth mentioning that if there are islands of word slots in the grid that do not interlink each other, the algorithm will start each new island as a new crossword and start with the highest interlinking word slot, then its linking word slots etc. This algorithm will ensure that *at least one point of failure will occur when filling up each word slot, if the word slot in question interlinks any other word slots.*

2.3 Uniqueness Constraint on Word Selection

A problem that occurs when using Berghel's Horn clause method is that multiple word slots can be assigned the same word. This obviously results in very boring crosswords and therefore it is usual when constructing a crossword not to allow multiple occurrences of the same word (certainly, seldom, if ever, do human-edited crosswords have duplicates among the words comprising the solution). Berghel suggested a way of avoiding this. He suggested that once a solution is found a check should be done for uniqueness of all the words and if a multiple occurence was found, then the program would backtrack until it found a solution to the grid that did not contain multiple occurences of the same word. This obviously resulted in naive backtracking. In our implementation we therefore included a check prior to the insertion of a word into a word slot that ensured a word would not be inserted if it was already in the solution. Clearly, backtracking is still a possibility; however, in our refinement of his proposal the proof failure which instigates backtracking occurs much earlier.

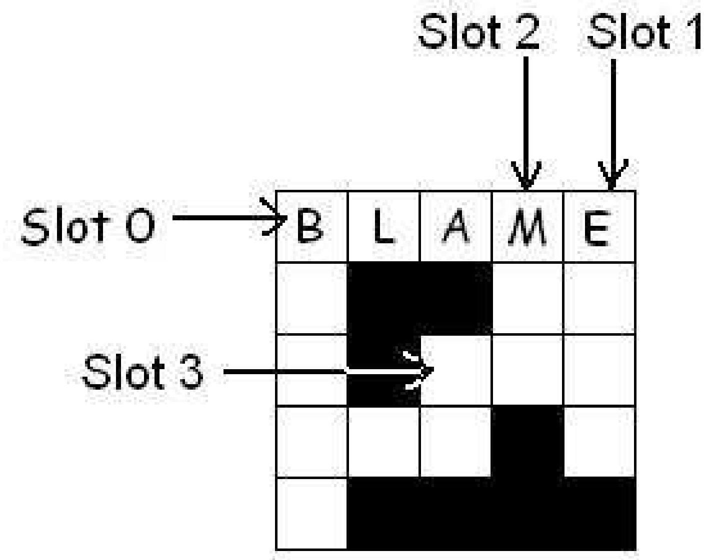

Figure 5: A Setting for Impossible Combinations to Ensue.

2.4 Eliminate Assignments Yielding Impossible Adjacent Words

An important constraint we introduced was that to eliminate the insertion of words that resulted in impossible letter combinations (suggested by Berghel). For example in Figure 5, the word ‘blame’ has been inserted into word slot 0 and then 'eave' into word slot 1. When filling in word slot 2 it could for example use the word ‘eave’. The program would then go on to fill in word slot 2 with ‘mix’. This results in an impossible combination of characters in word slot 3 and the program would backtrack to redo word slot 2. It might redo word slot 2 with another impossible combination of characters, which obviously results in naive backtracking. Therefore, prior to a word being inserted into a word slot a constraint is put in place to avoid resulting in impossible combinations of characters. In our alternative solution, the same grid constructed manually as described above would be automatically constructed as follows (each word slot is uniquely identified, but mutual constraints among words are maintained with variable sharing as before; and the word slots are not discriminated as arguments to the primary predicate, but are implicit in the list of lists that the `crossword/1` fact applies to). This fact is generated automatically on the basis of the grid, and is populated with respect to a representation of the wordlist comparable to Berghel's (1987).

```
crossword([[0, [C1, C2, C3, C4]],
        [1, [C5, C6, C7, C8]],
        [2, [C9, C10, C11, C12]],
        [3, [C13, C14, C15, C16]],
        [4, [C1, C5, C9, C13]],
        [5, [C2, C6, C10, C14]],
        [6, [C3, C7, C9, C15]],
        [7, [C4, C8, C12, C16]]]).
```

The predicates which resolve the puzzle's solution manipulate the list of lists of uninstantiated variables with reference to the section of the lexicon that is thematically identified as relevant for populating the grid, and make use of fairly standard operations to instantiate the free variables (e.g. member). Things become more interesting in rating the relative acceptability of words to populate the grid.

2.5 Adopt Heuristics for 'Best' Choices of Words for the Grid

An additional constraint we imposed on the program was a heuristic to estimate the likelyhood of a given word causing a problem at a later stage in the grid. The word that is inserted will be the one that is least likely to cause a problem at a later stage. If that word happens to fail then the word that was second least likely to cause a problem (according to the heuristic) will be inserted upon backtracking and so on. As stated the heuristic value that a word is given is based on how unlikely it is to cause a problem. The higher the value, the less likely the word is to be problematic. The heuristic value that we adopt is calculated as follows:

1. the program considers a grid identical to the one we are trying to solve, with all the word slots that have been given a word already retaining there words
2. the program assumes inserts the word to be evaluated into the grid
3. it then examines each interlinking word slot
4. for each word slot it counts the number of words in the lexcicon that could be inserted into the slot
5. it takes the lowest count out of all the linking word slots
6. it uses this number as the heuristic value of the initial word

It is important that the program considers a grid with all the word slots that have been instantiated already so as to get an accurate evaluation of the word in relation to the current state of the grid. It would be interesting to consider a dynamic programming approach towards yielding a best-fit solution given a greater range of possible solutions to the words selected 'so-far'; however, it has been noted that a point of this research is in enforcing as many constraints as possible as soon as possible, and thus, alternatives as just suggested arrive solely through backtracking. Given the representation of words as lists of variables, one for each letter, and the constraining impact of variables for word slots being shared, if some of the cells of the word slot that is being filled have already been instantiated with a character, this is taken into account when evaluating the word slots.

2.6 Refill Word Slots Intelligently on Backtracking

The most important constraint that we put on the program is to restrict what the program re-attempts to succeed when backtracking. As mentioned above, when the program fills up a certain number of word slots will usually reach a word slot that it cannot fill and begin to backtrack. We call this failing word slot the *problem word slot*. So it begins to backtrack and redoes the word slot before the problem word slot. It happens frequently that a word slot is preceded by a word slot that does not interlink it. Therefore redoing the word slot before a given word slot may not have any affect on the problem word slot failing, and therefore its pointless to redo the previous word slot. This fact is the main cause of naive backtracking that occurs in the program. For example if the word slot preceding a problem word slot has 50 possible words, and the word slot before that has 40 possible words and redoing both of them has no affect on the problem word slot, the program will attempt the problem word slot 2000 times before it finally attempts to redo the word slot three back from the problem word slot and so on until it reaches a word slot that actually in some way affects the success of the problem word. So in order to stop this naive backtracking a constrint was put in place that ensured that only word slots that affect the problem word slot will be redone when backtracking. This causes the program to go directly to the last word slot that has been given a word and instantly change something within the problem word slot. This results in the amount of naive backtracking being significantly reduced. The program therefore records the problem word slot and when its about to redo a word it checks first of all will it have an effect on the problem word slot. Recording which word is the problem word slot is done as follows:

- before a word is attempted (before it is a problem) each word is recorded as being a problem
- when the word is successfully given a word from the lexicon the fact that the word slot is a problem is erased

This method of recording the problem word slot gets around prolog undoing everything, recording where the point of failure occurred. It then uses this information to decide which word slots are useful to redo. This causes the program to backtrack sensibly, avoiding exhaustive and pointless inserting of words.

3 Conclusion

This paper outlines our improvements on the suggestions origionally provided by Berghel (1987) and Berghel and Yi (1989) for automatically generating crossword puzzles. Crosswords are obviously constraint satisfaction problems, and setting crosswords is arguably an even greater constraint solving problem than solving the resulting puzzle. Important aspects of the problem (e.g. lexical database management, clue generation, etc.) are not addressed here, but are developed in the actual implementation (Gibbons, 2002; Graham, 2003). An advance reported on in this work is in the automatic generation of Prolog code which is designed to incite backtracking sooner rather than later. We have provided representative examples of code implemented by hand per puzzle in the initial work as well as examples of our automatically generated optimized code. Of course, for certain grids local minima of a sort may exist in that two regions may be wholly independent, but we claim that for connected grids, our automatically created population with words to be solved for is an improvement on all other options presently available.

Future research should obviously address greater optimization. We have actually addressed the interesting problem of augmenting the lexical supply (e.g. multilingual sources) in interesting ways. It would be worth exploring the degree to which crossword puzzles generated automatically in this way are useful in pedagogy (whether language or domain-specific learning). Other dimensions for enhancement are aligned with the decomposition of the problem supplied by Berghel (1987), such as alternative grid composition, multiple solution determination, etc. The system described is fully implemented and enjoyably functional.

References

Berghel, H. (1987). Crossword Compilation with Horn Clauses. *The Computer Journal*, *30*(2), 183–88.

Berghel, H. & Yi, C. (1989). Crossword Compilation with Horn Clauses. *The Computer Journal*, *32*(3), 276–280.

Gibbons, A.-M. (2002). Random Crossword Generator with a Web Interface. Department of Computer Science, Trinity College, University of Dublin. Bachelor in Engineering. Final Project Dissertation.

Graham, Y. (2003). Random Crossword Generator with a Web Interface. Department of Computer Science, Trinity College, University of Dublin. Moderatorship in Computer Science, Linguistics and German. Final Project Dissertation.