Constructing Crossword Grids: Use of Heuristics vs Constraints

Gary Meehan* and Peter Gray†
Dept. of Computer Science
University of Aberdeen
Kings College
Aberdeen
UK, AB24 3UE

September 17, 1997

Abstract

This paper reports on the construction of crossword puzzles, by filling in dense blank grids (consisting mostly of white squares) until all the white squares contain letters and all across/down words in the grid are valid English words in the usual crossword style. Two approaches are compared.

The first approach used Prolog and C to build puzzles on a word-by-word basis, using a variety of heuristics to determine the order in which to fill in words and then which words from a large dictionary to put in the grid. The second method used a constraint (CLP) language and its C language interface, to construct puzzles on both a word-by-word and a letter-by-letter basis, using the same constraints as the Prolog/C method but a different search strategy. Conclusions are drawn about the need to incorporate better facilities in the CLP language which will use constraints of this kind actively in the search mechanism.

1 Introduction

The construction of crossword grids by hand is a surprisingly labour-intensive task, involving much perusal of dictionaries and backtracking over choices. It precedes the even more difficult stage of actually generating clues, and is an area that is ripe for automation. The earliest serious work was by Mazlack [5], who viewed it as a heuristic (state-space) search problem. However, he only had a small dictionary (2000 words of 3-4 letters) and a 212K IBM370 mainframe. He could not solve it with whole words as variables (see Section2.1) and so retreated to a letter by letter approach with backtracking.

*Currently at the Department of Computer Science, University of Warwick, Coventry, U.K., CV4 7AL. E-mail garym@dcs.warwick.ac.uk

†E-mail: pgray@csd.abdn.ac.uk

```
s a t e   p o d   m l e a
i m i t   l e w   o o z y
m e c h   a r e   r a r e
a s k e r s   l l a m a s
      r i m   l e s
p l a s m       i s l a m
o e r               e t a
d w e l l       o s i e r
      l e o   i n k
m a c a w s   n e a r e r
i t e m   i n k   t a m e
c o d a   e e l   e r i e
a m e s   r o e   d e l l
```

Fig. 1. Example output from filling a 13×13 grid

Fifteen years later Ginsberg *et al* [4] (1990) revisited the problem using a Symbolics Lisp machine with 2 megawords and a 24,000 word dictionary with words of up to 15 letters. They formulated the problem in terms of constraint satisfaction, alternately choosing a variable to solve (word to fill in) and then picking a value from the dictionary. Various combinations of strategies for filling and picking were experimented with. They concluded (a) that *arc-consistency* (see Section 2.1.3) and the cheapest-first heuristic were essential to solve any of the puzzles; (b) that backjumping (a kind of intelligent backtracking) was needed for the harder puzzles; and (c) they advocated dynamically recomputing the number of combinations of choices remaining for filling in words crossing a given word, rather than estimating it from pre-computed averages. With these techniques they managed to fill up to 13×13 crosswords with remarkably dense grids (high numbers of words crossing other words) before their method failed to terminate on bigger examples.

Since then, the technique of Constraint Logic Programming (*CLP*) has been developed by van Hentenryck [6] and others, and has attracted wide interest. It overcomes the classic shortcomings of backtracking, by dynamically reordering the goals, and has built-in checks for arc consistency (called lookahead). We have been using one such CLP system CHIP [1, 3] for various projects and thought it was timely to re-investigate this classic problem, using the builtin facilities of CHIP for solving finite domain problems. CHIP looks syntactically like Prolog, but many of the goals are delayed and stored in an internal constraint network until the so-called labelling phase which tries to solve constraints on specified variables for values.

Thus we followed the approach of [4] and used three of their test grids, including the hardest 13×13 one, a completed example of which can be seen in Figure 1. We used the *most-constrained* strategy provided by CHIP for choosing which word to fill in, and hoped to discover that the problem was much easier written for a CLP solver than as a special purpose Prolog program. However, although we solved for all the grids, the problem proved less well adapted to CHIP than expected.

As [4] points out, the problem contains large numbers of variables (up to 60) with surprisingly large domains (up to 16,000 values in our case, or 7,000 in theirs). Thus it is worthy of serious consideration, even with machines of large main memory (up

to 320 Mb in our case).

2 Constructing Crossword Puzzles

2.1 Word-by-word Instantiation

The word-by-word instantiation method fills the grid by repeatedly picking an empty word pattern from the grid and filling it with a word from our dictionary (see Figure 2). The method terminates when the grid is full, i. e. there are no more word patterns to instantiate. If, at any point, we are unable to find a word from the dictionary that fills the pattern then we backtrack to find another match for a pattern that was filled previously. An example of this can be seen Figure 3 (we use '?' to represent an empty square).

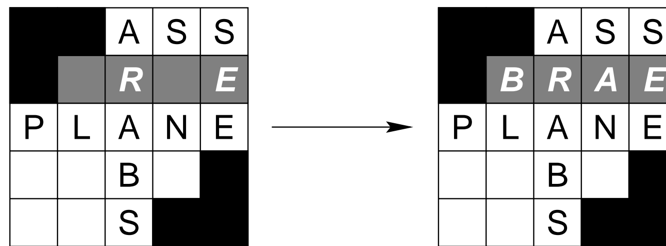

Fig. 2. Finding a pattern to fill and filling it

The efficiency of our instantiation relies on the efficiency of the three parts of our algorithm (as noted by [2]):

1. Choosing which pattern to fill (i.e. which variable to solve for).
2. Picking a suitable word (i.e. which value to select).
3. Choosing where to backtrack to when we reach an impasse.

As noted above, Prolog employs a simple backtracking method which goes back to the last choice point (as seen in Figure 3) but CHIP has an 'intelligent' backtracking method. It was part of our research to compare the two forms of backtracking.

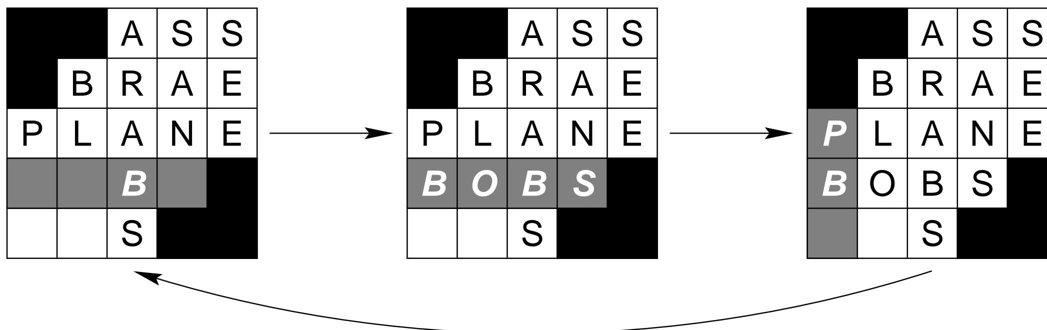

Fig. 3. After instantiating '??b?' to 'bobs' we find that we we have no match for 'pb?' and thus we must backtrack and find another match for '??b?'.

Pattern	Number of Matches
‘s?n?’	30
‘?l??’	151
‘p??’	52
‘?r?e’	10
‘??b?’	73
‘??s’	38

■	■	A	S	S
■		*R*		*E*
P	L	A	N	E
		B		■
		S	■	■

Fig. 4. Finding the most constrained pattern.

2.1.1 The Fill (Delete) Strategy

The process of picking a pattern from the grid is alternately known as the *Fill* or the *Delete* strategy. The latter is from CHIP nomenclature and comes from the process of deleting a uninstantiated constraint variable from a list of such variables and instantiating it.

Our aim when picking a pattern to fill is that we should maximise the number of choices for the remaining words in the grid and thus maximising our chances of filling the grid with the minimal number of backtracks. To achieve this maximisation we use the CLP strategy of instantiating the *most constrained* pattern, i. e. the pattern which can be filled by the minimum number of words (see Figure 4).

The reasoning behind using the most constrained strategy is as follows. Suppose we have n patterns, $p_1, p_2, \ldots, p_n$ say, yet to instantiate and that each of these p_i patterns has c_i choices, i. e. words that match it. Then a (very) rough estimate of the total number of choices is $\prod_{i=1}^{n} c_i$. If we instantiate pattern p_j, say, then the number of choices is reduced to $\prod_{i=1, i \neq j}^{n} c_i$. This product is maximised when c_j (the deleted factor) is minimised, i. e. when p_j is the most constrained pattern.

Of course, this proof is only applicable if the patterns are independent of each other, a case which certainly does *not* arise in crossword puzzles. Indeed, it is the dependence of the patterns which forms the whole basis for crosswords. However the most constrained strategy forms a useful heuristic in practice. It only remains to determine how we determine the most constrained pattern in the grid. CHIP will determine this automatically (see Section 4.2); with Prolog we develop a number of different methods for determining the most constrained pattern based on exact counting methods and statistical analysis (see Section 3.2).

2.1.2 The Pick Strategy

Above we noted that the various patterns in the grid are heavily dependent on each other, a fact we ignored when developing our fill strategy, but one that we acknowledge when determining which word from the dictionary to instantiate our pattern to.

For instance, in Figure 5 we have two possible instantiations for the pattern ‘??a?t’: ‘start’ and ‘quart’. If we pick the former then we will eventually have to find matches for the pattern ‘s??’; if the latter then we need matches for ‘q??’. There are 60 matches in our dictionary for ‘s??’ but only 2 for ‘q??’ so obviously, following our strategy for maximising choice, we choose to instantiate ‘??a?t’ to ‘start’ rather than ‘quart’.

Therefore when choosing a word to fill a pattern with, we choose the word that

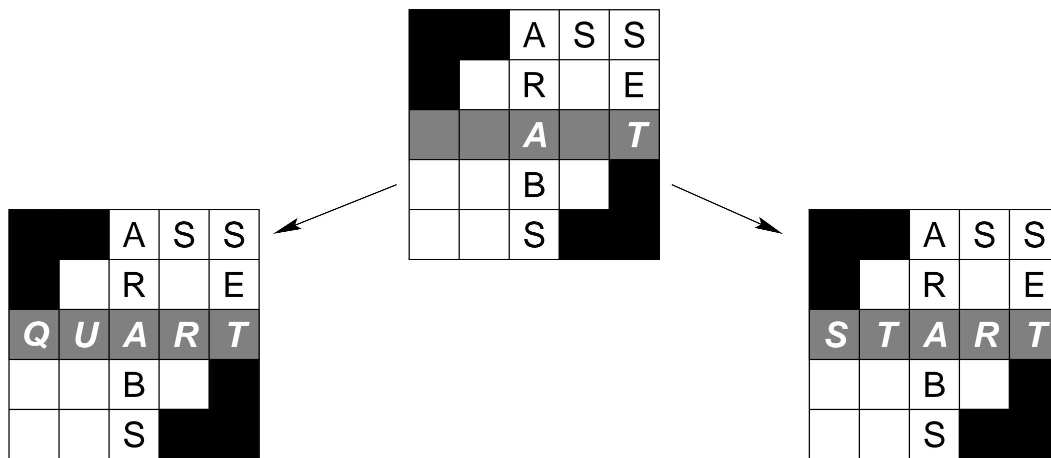

Fig. 5. Two possible instantiations for '??a?t'.

will lead to the maximum number of choices for the rest of the patterns in the grid. As with the fill strategy, there are a number of ways in which we can calculate this choice (see Sections 3.3 and 4.3)

2.1.3 Arc-consistency

Figure 3 shows us a grid that cannot possibly be completed, as there is no match for the pattern 'pb?'. In technical terms, the grid is not *(directionally) arc-consistent*. The question is, when do we discover this?

If we are lucky, then we will try to instantiate 'pb?' immediately after instantiating '??b?' to 'bobs' and thus be able to deduce that this instantiation was illegal (with respect to making sure that we can fill the grid) and we can immediately try another instantiation. However, we may not try to match 'pb?' until it is the last empty pattern in the grid. A simple backtracker, for example Prolog's, will spend many fruitless backtracks re-instantiating all the patterns between 'bobs' and 'pb?' in an effort to find one that will transform the uninstantiable 'pb?' into an instantiable pattern (this is why Ginsberg *et al* recommend smart back-jumping).

It is thus much more efficient to check that a grid is arc-consistent, i. e. still fillable, every time we instantiate a pattern. Some fill and pick methods include arc-consistency checking as an 'added bonus'; some strategies need a separate check for arc-consistency. See Sections 3.4 and 4.4 for further details.

2.2 Letter-by-Letter Instantiation

The letter-by-letter instantiation method works by repeatedly picking an empty square from the grid and instantiating it to some letter. The method terminates when all squares have been instantiated. At each instantiation, we must ensure that instantiating a word with a letter does not preclude any word patterns that contain that square from being instantiated to a word from our dictionary. This is actually a check for arc-consistency, since if we have an arc-consistent grid before we instantiate a square, and the pattern(s) that contain this square are fillable after instantiation, then the whole grid must remain arc-consistent as the instantiation affects no other words.

For instance, consider the grid in Figure 6. If we instantiate the grey square to 'q' then we have to find a match for the pattern 'qre?', a match that does not exist in our dictionary. On the other hand, instantiating with 'b' does lead to a pattern ('bre?') for which a match exists (e. g. 'bred'). If, at any point, we have a grid with patterns that cannot be instantiated then we backtrack to find another match for a square that was instantiated previously.

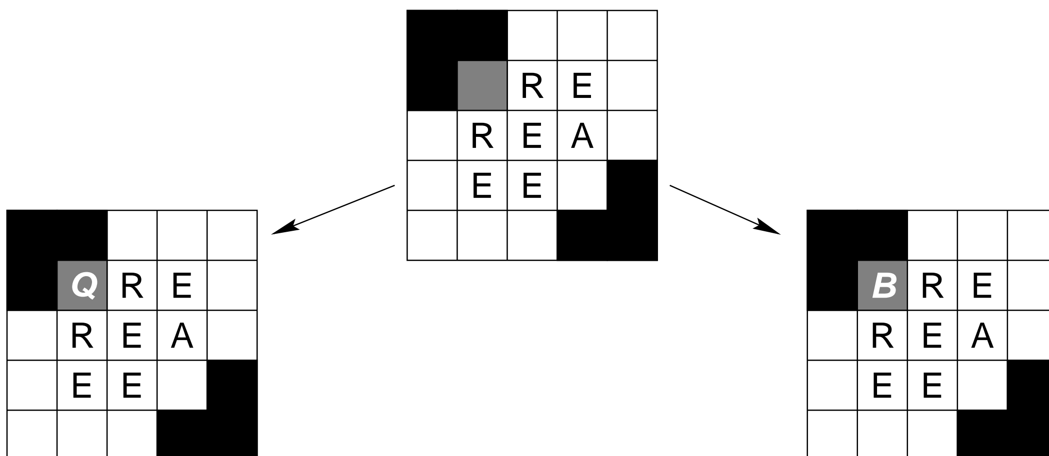

Fig. 6. Two possible instantiations for the grey square; one legal, the other not.

We have a similar strategy to the word-by-word instantiation method. Again the efficiency of the algorithm relies on the three component parts:

1. Picking which square to instantiate.
2. Picking which letter to instantiate it with.
3. Choosing where to backtrack to when we reach an impasse.

The remarks on backtracking are identical to those made about backtracking in the word-by-word method made in the previous section.

2.2.1 The Fill (Delete) Strategy

As with the word-by-word fill strategy, we adopt the principle of instantiating the most constrained square first. But how do we determine which square this is? There are a number of dynamic strategies which will be discussed in Section 5.3. In this section we shall discuss a *static* strategy, i. e. one that can be determined *before* we start instantiating the grid and never alters.

This strategy was developed by Mazlack [5] who realized that squares that appeared in long patterns and on the intersection points of patterns were the most critical and should be instantiated first[1]. He gave a formula for determining the criticality of a square, s say, which we shall denote as the Mazlack value, $M(s)$:

$$M(s) = \#\text{ squares adjacent to } s + \#\text{ squares directly connected to } s$$

[1]The more critical a square is, the less room for manoeuvre we have when instantiating it and thus the more constrained it is.

A square is adjacent to another if they share a common boundary. Two squares are directly connected if they occur in the same word pattern. Two examples of this calculation can be seen in Figure 7. Here we see that the centre square, sitting in the intersection of two five-letter patterns (the longest patterns in the grid) has a much higher Mazlack value, i. e. is more critical, than the bottom-left square which sits on the intersection of two 3-letter words, which corresponds with our intuitive belief. Hence, when using the Mazlack strategy we choose to instantiate the empty square with the highest Mazlack value.

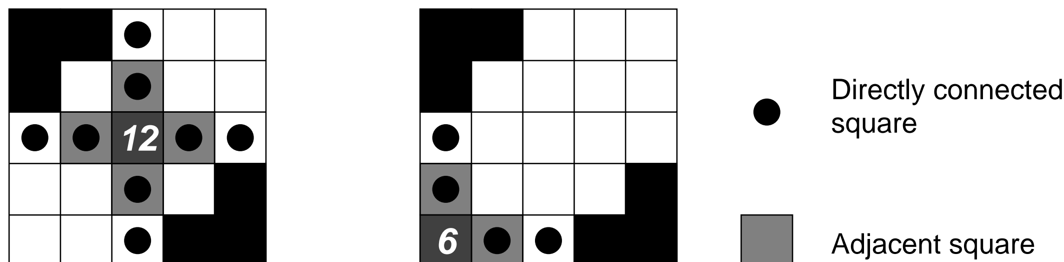

Fig. 7. The Mazlack values for two squares.

2.2.2 The Pick Strategy

When investigating the word-by-word pick strategy, we discovered that some words were more preferable to others. Similarly, when instantiating a square with a letter, some letters are better than others. Indeed, if the letters lead to non-fillable patterns then the letters are no good at all. We should choose letters that are consistent with our strategy of maximising choice, i. e. instantiate squares with letters that maximise the number of choices for the pattern(s) which the squares lie in. The exact methods are detailed in Section 5.3.

2.3 The Dictionary

The program repeatedly searches through a dictionary looking for matches to patterns, so obviously we have to ensure that this searching is as fast as possible. We thus choose to use C to handle the dictionary and pattern matching using Prolog's foreign language interface and CLIC to interface the C code with Prolog and CHIP respectively.

One rather useful characteristic of our problem is that we always know the length of the pattern that we are trying to find a match for. We thus order our dictionary by length and restrict our search to that subset of the dictionary which contains words of the same length of the pattern we are trying to match. We also sort these subsets of the dictionary (sub-arrays to be exact) lexicographically, thus enabling us to exploit binary search to some degree, that degree being dictated by the length of the initial segment of the given pattern that has been instantiated. The dictionary itself was sourced from

ftp://sable.ox.ac.uk/pub/wordlists/dictionaries/knuth_words.Z

and uses American English. All words were converted to lower case and all puctuation stripped out. All possible abbreviations were removed and an arbitrary limit of 15

Word length	Number	Word length	Number
2	134	9	15,045
3	853	10	12,705
4	3,342	11	9,707
5	6,894	12	6,920
6	10,745	13	4,635
7	14,574	14	2,809
8	15,901	15	1,633

Fig. 8. Frequency of words of length 2–15 letters in the dictionary

letters was placed on the lengths of the words. The resulting dictionary contains 105,897 words of length 2–15 as detailed in Figure 8.

2.4 Seeding the Grid

A number of the fill and pick strategies are deterministic and when presented with the same grid will always fill it with the same words. This is a little counter-productive, and we thus seed the grid by placing a random word in a random pattern before starting the instantiation process. The more patterns this pattern intersects with, the more random the grid will be. For this reason, we choose to seed (one of) the longest pattern(s) in the grid.

3 The Prolog/C Word-by-Word Implementation

3.1 Data Representation

The patterns in the grid are stored as an indexed list of tuples of the following form:

```
grid_string(StartX, StartY, Direction, Length, Freedom)
```

`StartX` and `StartY` hold the coordinates of the starting square for this pattern, `Direction` its direction (down or across) and `Length` its length. `Freedom` is a measure of how constrained a pattern is (see Section 3.2.

Our strategy detailed in Section 2.1.1 translates thus:

1. Delete the tuple from this list with the lowest `Freedom`.
2. Instantiate the corresponding pattern in the grid.
3. Propagate this instantiation to the remaining patterns into the grid, in particular updating the `Freedom` of each pattern. Note that only patterns that intersect with the instantiated pattern are affected and it is only these that we need to consider.
4. If the grid is not arc-consistent try another possible reinstantiation of the chosen pattern. If no further matches are possible then backtrack.

Following [4] we limit the number of choices that the program can try before backtracking to an arbitrary limit of 10. This is based on the principle that if we haven't found a successful match after a certain number of tries we aren't likely to find one.

3.2 The Fill (Delete) Strategies

We shall now detail the ways of calculating which pattern is the most constrained. For simplicity, we shall refer to them as the fill strategies even though they use the same underlying strategy and only differ on how they make their calculations.

`most_constrained` We count exactly how many matches every pattern has. This has the benefit that if the grid isn't arc-consistent then some patterns will have a `Freedom` of 0 and thus will be considered first, fail and force an immediate backtrack. Hence no separate arc-consistency check is required.

`ratio` For every pattern we calculate the ratio of uninstantiated letters to the length of the pattern, e. g. '??b?' has a ratio of 0.75. Patterns with a higher ratio should have more choice of instantiations, conversely those with a lower ratio are more constrained.

`est_constrained` We estimate the constraints on a variable by using a pre-computed probability table containing the values of $P(l, n, c)$ which is the probability that a l-letter word has character c at position n (starting from 0). Note that if c is uninstantiated then we presume that this probability is 1. If we have a pattern of the form $c_0 c_1 \cdots c_{n-1}$ then an estimate of the number of matches is:

$$T(l) \times \prod_{i=0}^{n-1} P(n, i, c_i)$$

where $T(l)$ is the total number of l-letter words (see Figure 8).

The different methods are compared in Section 3.5.1.

3.3 The Pick Strategies

We now detail the differing pick strategies:

`first_n` We ignore our principle of maximising choice and consider the first n (in our case, 10) matches to the pattern. However, this does lead to a profusion of words from the beginnings of the various sections of the dictionary. This clustering means that if one of our matches fails then there is a higher than expected chance of the other matches failing as they have more letters in common than average.

`random` To alleviate the problem of clustered matches we pick a random selection of n (again 10) matches from a large subset of matches.

`dynamic` After obtaining a set of n random matches, we sort them by how much they preserve the choice in the grid. This is done by calculating the product of the number of choices of the intersecting patterns (this number is counted exactly as in the `most_constrained` fill strategy). The higher this product, the more choice.

`prob` Similar to `dynamic` but we use the `est_constrained` method to estimate the number of choices for each intersecting pattern.

The different methods are compared in Section 3.5.2.

	5 × 5	9 × 9	13 × 13	15 × 15
Words	10	20	64	30
Constraints	29	52	196	92

Fig. 9. Statistics for the test grids

3.4 Ensuring Arc-consistency

All combination of pick and fill methods, bar those involving the `most_constrained` fill strategy require a separate check for arc-consistency. This is achieved by checking that the intersecting words of the just-instantiated pattern all have matches. We have no need to check the entire grid as only these patterns are affected by the instantiation presuming, of course, that the grid was arc-consistent before the instantiation.

3.5 Results

The results were computed using a SPARCserver 10/512 with 2 50MHz SuperSPARC processors (135 MIPs each) and 320MB of memory and were averaged out over 10 executions of the constructor. Quintus Prolog 3.1.3 and the Sun C compiler were the language implementations used. A '—' indicates that the program did not run to completion in satisfactory time or crashed the computer. These crashes are probably due to memory fragmentation problems (a lot of allocation/deallocation was done in C which has no automatic garbage collector).

The various grids used are shown in Appendix A. They vary in number of words and constraints (number of words + number of intersecting squares) as shown in Figure 9.

3.5.1 Comparing the Fill Strategies

The results in Figure 10 were computed using the `prob` pick strategy (Section 3.3) and show total times (fill+pick) averaged out over 10 executions.

Size	`ratio`		`most_constrained`		`est_constrained`	
	Time (s)	Backtracks	Time (s)	Backtracks	Time (s)	Backtracks
5 × 5	0.4	31.8	0.89	18.2	1.1	59.4
9 × 9	0.33	3.9	0.89	1.7	0.84	8.6
13 × 13	12.95	3167.6	6.24	226	60.15	2961
15 × 15	4.98	130.9	8.6	1.5	4.75	4.6

Fig. 10. Prolog Fill Strategies by Time and Backtracks

The first surprise is that all the different delete strategies take no longer and far less backtracks for the harder 9 × 9 grid than for the 5 × 5 one. This change in performance may be explained by the relative scarcity of the shorter words that appear in the 5 × 5 grid as opposed to the longer ones in the 9 × 9 one, though this is only a guess. The various delete strategies do not differ wildly from each other,

though the `most_constrained` is overall the best and proved to be the stablest as well.

3.5.2 Comparing the Pick Strategies

The results in Figure 11 were computed using the `most_constrained` fill strategy and again show total times averaged out over 10 executions of the constructor.

	`first_n`		`random`		`prob`		`dynamic`	
	T (s)	Btracks	T (s)	Btracks	T (s)	Btracks	T (s)	Btracks
5×5	1.02	35.3	0.96	67	0.89	18.2	0.95	75.1
9×9	0.65	28.6	0.93	33.2	0.89	1.7	1.28	72.3
13×13	—	—	8.43	526	6.24	226	6.65	293
15×15	4.79	9.2	6.98	34.8	8.6	1.5	5.12	7.9

Fig. 11. Prolog Pick Strategies

As with the differing delete strategies, we have the surprising similarity in performance from a 5×5 grid to a 9×9 one, though the difference in backtracks is not so marked this time. While the simple strategies `first_n` and `random` perform well on the smaller grids, as the grids become larger and more complex the extra work done by the more involved strategies begins to pay off.

4 The CHIP/CLIC Word-by-Word Implementation

4.1 Data Representation

We can view the word-by-word instantiation method as a constraint logic problem in which the patterns in the grid have to be instantiated, with a constraint that the instantiations form words from our dictionary.

We represent the various patterns in the grid as tuples of the form:

```
grid_string(StartX, StartY, Direction, Pattern, Word)
```

`StartX`, `StartY` and `Direction` are the same as in the Prolog method (see Section 3.1). `Pattern` is a tuple of squares from the grid, held in Chip domain variables, which form the pattern. `Word` is a domain variable representing all possible matches for the pattern. Each word, w, is represented by an integer. If w has k letters and occurs at index n in the dictionary array, and the k-letter words start at index i_k in the dictionary (remember that all the k-letter words occur in one contiguous block), then we represent w as the offset $n - i_k$. This is motivated for two reasons: we instantly discount any words which don't share the same length as the pattern that we are trying to instantiate; and CHIP has a size-limit of 100,000 on domain variables and we have over 105,000 words in our dictionary.

4.2 The Fill (Delete) Strategies

Our policy of picking the most constrained pattern in the grid to fill in next can be simply achieved using CHIP's `most_constrained` predicate. This deletes the most constrained domain variable from a list of such variables. In our case, as we have a list of `grid_string` tuples, we supply an index telling CHIP which field of the tuple holds the domain variable that we are interested in.

4.3 The Pick Strategies

After we have obtained the most constrained pattern, we can instantiate it using CHIP's `indomain` predicate. This instantiates the pattern to some word in its domain, this domain being the fifth element of the the `grid_string` tuple.

After instantiation and arc-consistency checking (see the next section), we have to update the domain of the other patterns. This has to be done inefficiently (see Section 6) as CHIP is unaware of any connection between the squares forming the pattern and its domain variable. We remove all the words (or rather their offsets) that don't match the updated patterns from the domain variable representing the possible matches for the pattern.

4.4 Ensuring Arc-consistency

We employ the same C predicate as was used for the Prolog check (see Section 3.4).

4.5 Results

The same machine as was used for the Prolog/C method was used along with CHIP v4. The results in Figure 12 are only sample results, rather than averaged, as the program proved too brittle to get a complete set of results (Ginsberg also reported problems of non-termination). The backtracking is reduced as expected, compared to Prolog/C, but at a high cost in performance (see Section 6).

Size	Time (s)	Backtracks
5×5	7.5	7
9×9	10.8	3
13×13	71.1	206
15×15	60.3	10

Fig. 12. CHIP Word-by-Word Instantiation results

5 The CHIP/CLIC Letter-by-Letter Implementation

5.1 Data Representation

We can view the letter-by-letter instantiation method as a constraint satisfaction problem in which the letters in the grid have to be instantiated, constrained such that the patterns which they form contain words from our dictionary.

The squares in the grid are stored in a list, and are represented by tuples of the form:

```
let(X, Y, Letter, MazlackValue, Domain)
```

`X` and `Y` are the coordinates of the square. `Letter` is the domain variable representing the square. `MazlackValue` is the Mazlack value of the square. `Domain` is a list of possible instantiations (from the letters 'a'–'z') for the square. This list is sorted according to the letter's suitability for instantiating a square (see Section 5.3) and is separate (though equivalent set-wise) to the domain of the variable `Letter`.

5.2 The Fill (Delete) Strategies

We sort our list of `let` tuples according to their Mazlack value (in descending order) and simply choose the head of this list to instantiate at each stage of the algorithm. As a comparison, we also delete the squares from the list using CHIP's `first_fail` predicate (a variation of the `most_constrained` predicate recommended when the domains of our variables are of roughly the same size).

5.3 The Pick Strategies

In Section 2.2.2 we said that we should pick certain instantiations over others to maximise choice. This method is denoted the `sorted` strategy. We now elaborate on how to actually do this.

A square can occur in either one pattern or two. In the first case it must be the ith character of a n-letter word. Let $N(n, i, c)$ be the number of n-letter words with character c at the ith position. Then, to maximise choice, we sort, in descending order, our domain (the fifth element of `let`) using N and try possible instantiations from the head of this list and proceed accordingly. If the square occurs in two patterns, say as the i_ath letter of an n_a-letter word and the i_dth letter of a n_d-letter word, then we order the characters c on the product $N(n_a, i_a, c) \times N(n_d, i_d, c)$. This product represents the total number of combinations of words that intersect at that point which is instantiated to the character c. We only have to do this calculation and sort once, as with calculating and sorting on the Mazlack value. Note that if we are using the `first_fail` delete strategy, then we must also ensure that this domain and the domain of `Letter` are kept consistent.

For comparison purposes, we also instantiate letters using the `indomain` predicate.

5.4 Results

Disappointingly, the program proved even more brittle than the word-by-word implementation and none of the methods could handle the 13×13 grid. Thus the figures detailed in Figure 13 are only samples and not averages. The Mazlack heuristic was almost useless on the dense grids but, unlike the `first_fail` method, was able to deal with the the less dense 15×15 grid.

Size	`first_fail`				`Mazlack`			
	`indomain`		`sorted`		`indomain`		`sorted`	
	T (s)	Btracks	T (s)	Btracks	T (s)	Btracks	T (s)	Btracks
5×5	6.05	3024	0.67	248	67.76	14994	3605	181590
9×9	0.72	110	0.49	13	6.9	14754	0.55	17
13×13	—	—	—	—	—	—	—	—
15×15	—	—	—	—	413.35	3539	39.14	478

Fig. 13. CHIP Letter-by-Letter Delete and Pick Strategies

6 Discussion and Conclusions

In general, the selling point of CLP techniques is that they enable one to understand the problem better and to specialise and use with ease powerful constraint solving techniques that one would be unlikely to code directly in Prolog. It also overcomes the classic shortcomings of Prolog depth-first backtracking. However, in this case the Prolog version was similar in length to CHIP and outperformed it (both versions needed to use shared C routines).

There are various reasons for this:

- The Prolog version makes extensive use of call-outs to fast C routines which compute the choice factors and update an array representation of state. The CHIP version has difficulty in updating its finite domains efficiently. It has to call C routines to construct and return long lists of words that don't match, and then to recurse over this list creating CHIP inequality constraints in order to remove items from the domain. This is a slow process, made much harder by the size of domains (up to 16,000). In many cases it would be easy to supply a small list of the remaining values, but CHIP cannot use the information easily in this form.

- In the word-based version it is not easy for CHIP to use arc-consistency checks (lookahead) efficiently. Normally this relies on CHIP's own internal constraint representation. For example, in the paradigmic Eight Queens problem, the constraints are simple integer inequalities between domain variables, comparing the position of a queen in one column with that in a neighbouring column. Thus a change in the value of a variable is quickly and easily propagated and checked for arc consistency. In the crossword version, however, an equality holds between a component (letter) of one word and a component of a variable

holding a crossing word. This could not be represented as a direct constraint in our version of CHIP. Thus arc consistency could only be used as a passive filter, by calling out to a C routine, and not actively in the constraint network.

- In an effort to make more efficient use of CHIP we changed to a version based on letters in grid squares as variables. These values can be easily used in CHIP constraints. Now, alas, we had great difficulty in forming the constraint that a group of letters formed a valid word! Again this check could only be made by passively calling C routines. In theory one could write the constraint as an enormous disjunction of combinations of values (each representing a valid word). However, CLP is not good at using disjunctive constraints. Commonly, they are handled by testing one disjunction and then backtracking to modify the constraint in the network and try again. This can be done with modest numbers of disjunctions, but not with many thousands!

- There are minor differences in our results with those of Ginsberg on the same grid. We found 9×9 easier than 5×5, but this is probably because we had a different choice of short words in our dictionary.

- In the letter-based version we were able to retry Mazlack's original heuristic. This proved to be very weak, however, when using the much larger dictionary and more dense grids. In fact, the greatly increased memory of modern machines makes the word-based approach more viable, and it proved to be much faster.

In conclusion, we did solve a classic problem using a constraint logic solver, but found that it lacked certain key constructs, which stopped us using the constraints actively through CHIP's own constraint network. Instead they were represented in external C code. The CHIP word-based version did manage to reduce backtracking greatly compared to the Prolog one, but only at a cost of great computational overhead leading to significantly slower overall performance. Thus the CLP strategy is working in principle, but needs improvements to the constraint network before this pays off in performance.

7 Acknowledgements

This paper is based on work done while Gary Meehan was studying at Aberdeen University for his M.Sc., supported by an European Social Fund studentship. We are very grateful to Kit Hui at Aberdeen who suggested improvements to the algorithm and ran off extra results on a Sun with larger memory.

A Sample Grids

Figure 14 shows the example grids used through this paper. Grids A, B and C are sourced from Ginsberg's paper [4] while grid D was sourced from *The Times* crossword puzzle.

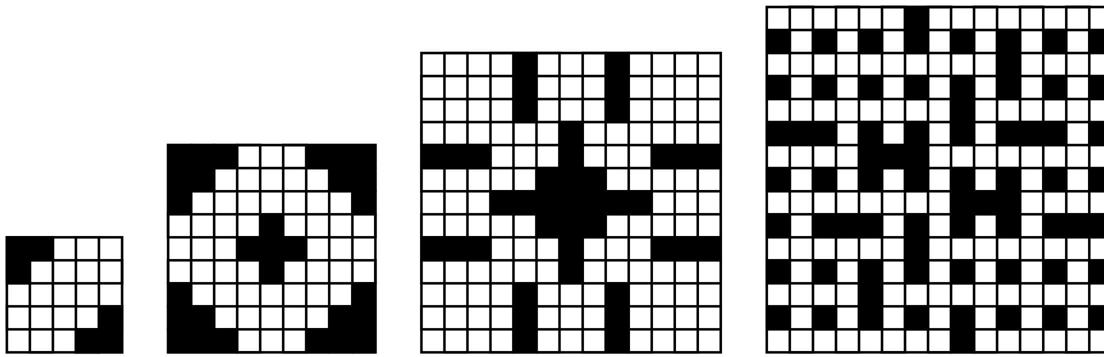

Fig. 14. Example grids.

References

[1] COSYTEC. *CHIP V4 User's Guide*, COSYTEC SA., Parc Club Orsay Universite, France, 1994.

[2] R. Dechter and J. Pearl. *Network-based heuristics for constraint-satisfaction problems* Artificial Intelligence 34, pp 1–38, 1988.

[3] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, F. Berthier. *The Constraint Logic Programming Language CHIP* Proc. Int. Conf. on Fifth Generation Computing Systems (Tokyo), pp 693–702, 1988

[4] M. L. Ginsberg *et al. Search Lessons Learned From Crossword Puzzles* Automated Reasoning, pp 210–215, 1990.

[5] L. J. Mazlack. *Computer Construction of Crossword Puzzles Using Precedence Relationships* Artificial Intelligence 7, pp 1–19, 1976.

[6] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming* MIT Press, 1989.