Search Lessons Learned from Crossword Puzzles

Matthew L. Ginsberg*
Michael Frank
Michael P. Halpin
Mark C. Torrance
Computer Science Department
Stanford University
Stanford, California 94305
ginsberg@cs.stanford.edu

Abstract

The construction of a program that generates crossword puzzles is discussed. As in a recent paper by Dechter and Meiri, we make an experimental comparison of various search techniques. The conclusions to which we come differ from theirs in some areas – although we agree that directional arc consistency is better than path-consistency or other forms of lookahead, and that backjumping is to be preferred to backtracking, we disagree in that we believe dynamic ordering of the constraints to be necessary in the solution of more difficult problems.

1 Introduction

Appearances notwithstanding, this is not a paper about crossword puzzles. It is a paper about search. More specifically, it is a paper about constraint satisfaction in large databases.

What we have done is to write a program that generates crossword puzzles by filling words into an empty frame. For frame sizes greater than 4×4, the associated search space is large enough to make brute force depth-first search impractical; heuristics must be used. The large branching factor is a consequence of the fact that any particular word can be chosen in many possible fashions – the dictionary used in this research contained some 24,000 entries. Crossword puzzle generation can therefore be used to compare the general techniques that have been proposed for solving constraint-satisfaction problems in large databases.

Crossword puzzles are fairly typical constraint-satisfaction problems; we share, for example, the constraint-satisfaction community's interest in finding a single solution to a given problem, as opposed to finding all such solutions. But crossword puzzles also tend to be far more difficult than most of the problems that have been discussed in the literature thus far. The recent work of Dechter and Meiri, for example, considers randomly-generated problems with 10-15 variables and 5 values [Dechter and Meiri, 1989]; our crosswords involved perhaps 60 variables and 7000 values.

In the next section, we discuss the general framework in which our experiments were performed. We also discuss some simple improvements to existing techniques that we developed in order to allow our program to solve more difficult crosswords.

Section 3 contains our experimental data. For each combination of the heuristics discussed in Section 2, we attempted to complete a variety of crossword frames; the average times needed to complete the search are shown, and the computational merits of the various techniques are discussed. Concluding remarks are contained in Section 4.

2 Existing work

There is very little existing work on the automatic generation of crossword puzzles; an old paper of Mazlack's [Mazlack, 1976] is the only one of which we are aware.[1] More relevant to our purposes is the work on constraint satisfaction.

As noted in [Dechter and Pearl, 1988], there are four choices to be made when solving a constraint-satisfaction problem: which variable to instantiate next, what value to use as the instantiation, how to handle backtracking, and what sort of preprocessing to do.

By a "variable," we will mean a particular word slot in the puzzle being generated; the constraints correspond to the fact that if two words intersect in a square s, they must use the same letter in that square.[2]

*This work has been supported by the Rockwell Palo Alto Laboratory.

[1]We will have little to say about Mazlack's work here. His techniques are very different from ours, and the performance of his program – even correcting for hardware advances since it was written – appears to be at least one or two orders of magnitude worse than ours.

[2]Constraint-satisfaction problems have duals, where the roles of the constraints and the variables are swapped.

Choice of variable

When choosing which variable to instantiate next (i.e., which word to fill in), one must decide whether the choice should be made when the puzzle is first examined (at "compile time") or whether it should be made dynamically as the other variables are assigned values (at "run time").

A variety of compile time heuristics have been discussed elsewhere in the literature. The so-called "cheapest-first heuristic" suggests constructing a statistical estimate of the number of choices remaining for each variable, and to then instantiate the variable that is the most constrained.[3] The "connectivity heuristic" suggests instantiating a variable that is constrained by the last variable instantiated, in order to ensure that one backtracks effectively when a dead end is reached.[4]

The best-known run time heuristic is known as "dynamic search rearrangement" and suggests that at any particular point in the search one should pick the variable that actually *is* the most constrained (by counting the number of possible solutions for each uninstantiated variable), instead of simply the one that is statistically *expected* to be the most sharply restricted as in the cheapest-first heuristic.

The two techniques that we chose to consider were the cheapest-first heuristic and its run time analog, dynamic search rearrangement. Connectivity was not considered because, as noted by Dechter and Meiri in [Dechter and Meiri, 1989], many of its advantages can be obtained by using backjumping [Gaschnig, 1979] instead of simple backtracking.

Choice of instantiation

Suppose, then, that we have selected a variable (i.e., word) to be filled next. How do we select among the possible values for it? In doing so, it is important to select one that restricts the possible choices for subsequent variables as little as possible. Why use a word with a Q when one with an S could be used instead?

In crossword puzzles, it is impractical to use this idea exactly; there are simply too many possible choices for each variable. Instead, we did the following: Suppose that we have decided to instantiate some particular variable by filling a particular word in the puzzle. The program considers the first k words that can legally fill this slot; suppose that we denote them by $w_1, \ldots, w_k$. For each w_i, the number of possibilities for each unfilled crossing word is computed, and the product of all of these values is calculated. The word actually chosen is that w_i that maximizes this product.

Of course, the behavior of this heuristic will be sensitive to the choice made for k. If $k = 1$, the first available word will be used at all times. But making k too large is also a mistake – all we really need to do is to make it large enough that one of the first k words is a fairly good choice. The time spent examining the rest of the possible words is unlikely to be justified by the small impact on the size of the subsequent search space. Some experimentation indicated that $k = 10$ was a reasonable value, and this is the value used in Section 3, where it is compared with the choice $k = 1$. The parameter k was called `min-look` in the implementation and this is how we will refer to it in Section 3.

Backtracking

When a dead end is reached (i.e., some slot is found for which there is no legal word), the program needs to backtrack and try something else. Simple chronological backtracking (backtrack to the last choice point, as in PROLOG) suffers from the problem that it may fail to address the source of the difficulty. If the program is having trouble filling the upper-left-hand corner of the puzzle, it is a mistake to make changes in a portion of the puzzle that have no effect on this problematic region.

This difficulty can be overcome using a technique known as *backjumping* [Gaschnig, 1979], which actually backs up to the source of the difficulty. It has been shown both theoretically [Dechter, 1990] and experimentally [Dechter and Meiri, 1989] that backjumping outperforms its chronological counterpart.

If no lookahead is done, backjumping can be implemented simply by always backtracking to a word that intersects a word that cannot be filled satisfactorily; the connection with the connectivity heuristic is clear in this case. If lookahead information is used as well (see the next section), then backjumping requires us to maintain, for each instantiated variable, a list of those subsequent variables that it affects in some way. This idea is obviously closely related to dependency-directed backtracking [Stallman and Sussman, 1977], although it is not quite the same because the expense of maintaining complete dependency information is avoided.

We also considered a small further improvement. Suppose that we have reached a dead end, and have decided to backjump to a particular word, w_1, that is the source of the difficulty. It is not too hard to determine what letter or letters in w_1 are causing the problem, and to then ensure that the new choice for w_1 avoids it. In conventional constraint-propagation terms, we

As Rich Korf has pointed out, the dual problem in the crossword-puzzle domain is also a natural one – the variables are the letters in the puzzle, and the constraints come from the fact that each letter sequence must be a legal English word.

[3]For example, if there are 2139 4-letter words in the dictionary, it is assumed that 2139/26 choices remain after the second letter is filled in. This is independent of whether the choice is E (in which case 281 completions remain) or Q (in which case there is in fact only 1).

[4]Thus having filled in one word, we would next fill a word that intersects it. By not jumping from one section of the puzzle to another, we ensure that chronological backtrack always considers a difficulty related to the one that actually caused the backup.

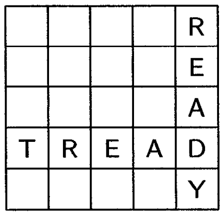


Figure 1: Is multiple lookahead worthwhile?

realize that the variable w_1 is causing trouble because of the constraints it places on a subsequent variable w_2 (or perhaps a collection of subsequent variables). Having done so, we make sure that the new choice for w_1 allows us to change our selection for w_2.

In Section 3, we will refer to the possible choices as "bt" (simple backtrack), "bj" (backjump to the relevant problem, but make no effort to ensure that the difficulty has been addressed) and "sbj" (smart backjump, ensuring that some relevant letter changes value).

Preprocessing

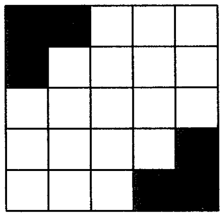
Finally, there is the possibility of preprocessing the data in some way that will reduce the need to backtrack in the first place. In [Dechter and Meiri, 1989], Dechter and Meiri suggest that the most effective way to do this is to preprocess the information at each node of the constraint graph in a way that ensures that when a particular variable w is instantiated, there will always be an instantiation for every other variable that shares a constraint with w. In terms of crossword puzzles, we make sure that the choice made for one word is consistent with the choices that will need to be made for the words that intersect it. This is called *directional arc-consistency* [Dechter and Pearl, 1988].

There are other possibilities as well. If we think of directional arc-consistency as a simple lookahead to depth 1, directional *path-consistency* is lookahead to depth 2, so that for the word being instantiated and every choice for a word w' that meets it, there will be a choice for every word that meets w or w'.

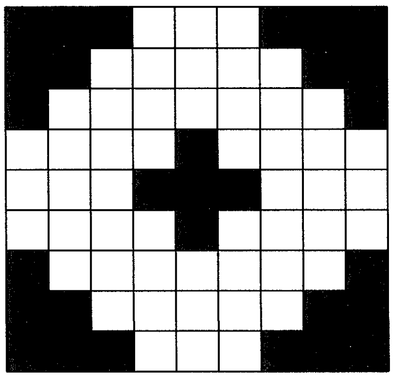
In a crossword puzzle, it is impractical to store all of this consistency information; there are simply too many possibilities. Indeed, this was already observed by Dechter and Meiri on the simple problems they investigate in [Dechter and Meiri, 1989].

It is possible, however, to repeat the analysis at run time, essentially doing a lookahead to a depth of greater than one when each variable is instantiated.

It seems at first that this should be a good idea. Consider the puzzle in Figure 1, for example. It might be the case that there are no two five letter words w_1 and w_2 such that w_1 ends in Y, w_2 has T as its fourth letter, and the last letter of w_2 is the same as the first letter of w_1. A two-level lookahead would notice this, and one of the two words in Figure 1 would be withdrawn immediately.



(a)



(b)

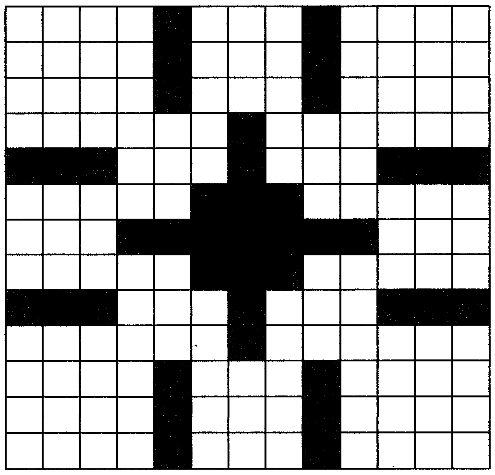
Figure 2: Test puzzles

In practice, this does not work so well. The reason is that the computation involved is a fairly difficult one – we need to look at the possible choices for w_1, check to see which letters are still possible in which spaces (this is the expensive part, since it involves examining each of the choices for w_1), and then to use this information to prune the set of possibilities for w_2. The analysis is expensive enough that the cost incurred is not in general recovered by the associated pruning of the search space. More conventionally put, the forward branching factor for the problem is high enough that additional levels of lookahead draw conclusions no more effectively than their backward counterparts.

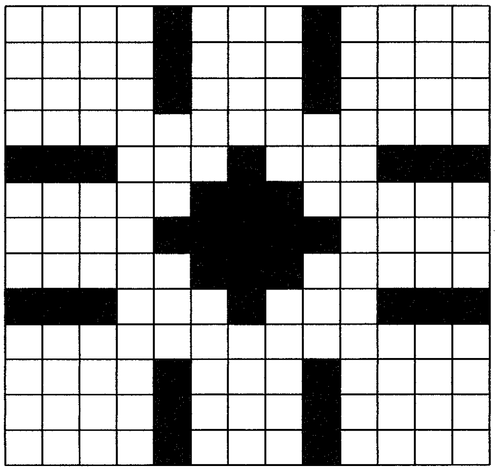
3 Experimental results

Frames used and raw data

In order to evaluate the usefulness of the ideas in the last section, the four puzzles appearing in Figures 2 and 3 were solved by the program. The program always used one level of lookahead (i.e., arc-consistency)



(c)



(d)

Figure 3: Test puzzles (ctd.)

and some form of the cheapest-first heuristic, since it was quickly discovered that without these, all but the simplest puzzles were intractable. The other parameters were set as follows:

1. `cheapest-first` could be either `stat`, indicating that a compile-time statistical approximation was used, or `exact`, indicating that the exact (run-time) value was used.
2. If the run-time ordering were used, `min-look` could be either 1 (always use the first acceptable word) or 10 (use the best choice among the first ten acceptable words). This calculation was not performed when the statistical ordering was used, since only dynamic information about variable orderings can be used to distinguish among words in this fashion.
3. `connected-backtrack` could be either `bt`, `bj` or `sbj`, as described in the previous section.

For each allowable selection of parameters, each of the above puzzles was solved 10 times; the dictionary was shuffled between each solution attempt to ensure that the performance of the program was not affected by a particularly fortunate or unfortunate choice of word at any point.

The results are as reported in Figure 4; the times reported are in seconds for a Symbolics 3620 with 2 megawords of memory. We feel that time of solution is a more valuable gauge of performance than the number of nodes examined or the number of backtracks (as used in [Dechter and Meiri, 1989]) because it is often possible to prune the search space but only at a prohibitive cost in terms of the time spent expanding a single node. This is the argument we made when considering lookahead to multiple depths.

For the harder puzzles, many of the choices of parameters did not lead to solutions being found within 20 minutes of CPU time, and no timing information is reported for these parameter choices. The most difficult puzzle ((d) in Figure 3) was solved in only 8 of 10 cases with `min-look` set to 1.

Analysis

As already mentioned, arc-consistency and cheapest-first were needed to solve any of the puzzles. With regard to the other choices, we observed the following:

Choice of variable Unlike the results reported by Dechter and Meiri in [Dechter and Meiri, 1989], it is apparent even for the 5×5 puzzle that runtime information plays an important role in the choice of word to be filled next. The difference in performance between the programs that used an exact version of the cheapest-first heuristic and those that used the statistical approximation available at compile time is significant in all cases; the two most difficult puzzles could not be solved at all within the twenty minute time limit unless runtime information was used.

Choice of instantiation The overhead involved in finding a word that minimally restricts the subsequent search is worthwhile only on puzzles of size 9×9 and larger, and it is not until the most difficult of the four puzzles is considered that this heuristic begins to play a significant role. This suggests that the choice of `min-look` (the number of words considered to fill a particular slot) should be closely coupled to the apparent difficulty of the puzzle being constructed.

Backtrack Backjumping (as opposed to simple backtracking) is another heuristic the value of which is only apparent on the larger puzzles; on smaller ones, the cheapest-first heuristic tends to order the words in a way that results in a particular word intersecting the word being filled next and the two techniques coincide. "Smart" backjumping is needed for puzzle (d) only – but here, it turned out to be absolutely crucial. The reason is that many of the 13-letter words have endings like "tion" and if this choice made the upper-right hand corner of the puzzle impossible to fill, it was important not to try another word with the same ending. Of course, it is not clear to what extent other constraint-satisfaction problems will share these features, but it is not unreasonable to think that they will.

Things that didn't work

It is also probably worthwhile to report on search heuristics that we tried, but that *didn't* reduce the time needed to find a solution to the puzzle.

Tree reordering while backtracking Suppose that backjumping has caused us to backtrack over a word w because it was not relevant to the problem that caused the backtrack in the first place. Is it reasonable to put w back into the puzzle before resuming the search, thereby modifying the order in which the search is conducted so that we can reuse the information that would otherwise be lost?

Unfortunately not. Just because removing w doesn't directly alleviate a particular difficulty is no reason to believe that *keeping* w won't commit us to the same problem. As an example, suppose that we put in a word w_1, then a crossword w_2, then discover that another crossword w_3 to w_1 cannot be filled satisfactorily. Provided that the choices for w_3 are not constrained by the selection of w_2, we will backtrack directly to w_1, and the above suggestion would therefore cause us to replace w_2 in the puzzle. Unfortunately, this replacement might well commit us to w_1 once again.

It may be possible to identify a limited set of situations where words that are passed over during backtracking can be safely replaced, but these situations appear to be fairly rare in practice and the cost of searching for them seems not to be justified.

Compile-time dictionary ordering We also considered the possibility of ordering the dictionary not randomly, but in a way that would prefer the use of words containing common letters.

For small puzzles, this led to significant performance improvements; the preference of common letters virtually eliminated the need to backtrack on puzzles (a) and (b). On puzzle (c), however, the performance gain was much more modest (perhaps 20%), while on puzzle (d), the performance appeared to actually *worsen* – the program was no longer able to solve the puzzle within the twenty minute time limit if `min-look` was set to 10. (For `min-look` set to 1, however, only 15 seconds were required.)

It is difficult to know what to make of such conflicting data; since the ordered dictionary can no longer be shuffled to eliminate statistical fluctuations in solution time, it is possible that the observed behavior is not reflecting the fundamental nature of the algorithm. The best explanation we can offer is the following one:

On large puzzles, where backtracking is inevitable, the ordered dictionary is likely to result in the first ten choices for any particular word being fairly similar. As a result, the program might just as well select the first word as any of the first ten; in fact, the time spent considering the others is unlikely to be repaid in practice. This is consistent with the observed behavior – the performance for an ordered dictionary with `min-look` set to 1 was uniformly better than if this parameter were set to 10. We feel this to be undesirable for the following reasons:

1. The performance of the program becomes quite brittle. If lucky, it will solve a puzzle very quickly; if unlucky, it may not solve it at all. This is the behavior that was observed on puzzle (d).
2. The program cannot improve its performance on very difficult puzzles by increasing the value of `min-look`, since this technique has been essentially invalidated by the dictionary ordering.

In addition, further experimentation showed that it was not possible to avoid this problem by segmenting the dictionary or ordering it in any other way (such as maximizing the letter differences between words appearing near each other).

We wish that we could make more definitive remarks about this technique, but cannot.

4 Conclusion

Summarizing, the conclusions that we have drawn from the experimental data in Figure 4 are the following:

1. Arc-consistency is needed if difficult constraint-satisfaction problems are to be solved effectively.
2. It is far more efficient to order variables at run time than to use the statistical information available at compile time.

3. It is important to select the instantiation for each variable carefully, although not so important that every choice should be considered in large domains.
4. Backjumping is to be preferred to simple backtracking. In addition, it is important to ensure that subsequent labels for a particular variable actually address a difficulty that was found previously in the search; this can be done without incurring the prohibitive costs involved in maintaining complete dependency information when the tree is expanded.

Acknowledgement

The first author is indebted to Greg Arnold, David Smith, Phil Stubblefield, Rich Washington and the students of his introductory artificial intelligence class for many enlightening discussions. The comments of Rich Korf and one other anonymous referee were also helpful.

References

[Dechter, 1990] Rina Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 1990. To appear.

[Dechter and Meiri, 1989] Rina Dechter and Itay Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 271–277, 1989.

[Dechter and Pearl, 1988] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[Gaschnig, 1979] John Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* Technical Report CMU-CS-79-124, Carnegie-Mellon University, 1979.

[Mazlack, 1976] Lawrence J. Mazlack. Computer construction of crossword puzzles using precedence relationships. *Artificial Intelligence*, 7:1–19, 1976.

[Stallman and Sussman, 1977] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

Puzzle	Cheapest first	min-look	Backtrack	Time (sec)
(a)	exact	1	bt	0.505
	exact	1	bj	0.533
	exact	1	sbj	0.605
	exact	10	sbj	0.607
	exact	10	bt	0.612
	stat	–	sbj	0.693
	exact	10	bj	0.719
	stat	–	bj	1.015
	stat	–	bt	1.041
(b)	exact	10	bt	2.363
	exact	10	sbj	2.453
	exact	10	bj	2.497
	exact	1	sbj	4.967
	exact	1	bt	6.327
	exact	1	bj	6.587
	stat	–	sbj	17.022
	stat	–	bj	17.170
	stat	–	bt	31.526
(c)	exact	10	sbj	11.904
	exact	10	bj	15.539
	exact	1	sbj	17.668
	exact	1	bj	17.798
(d)	exact	10	sbj	71.693
	exact	1	sbj	408.555*

* Completed on only 8 of 10 attempts

Figure 4: Test results; techniques not listed were not able to solve the puzzles in question