Search Lessons Learned from Crossword Puzzles

Matthew L. Ginsberg
Rockwell International
Palo Alto Laboratory
444 High Street
Palo Alto, California 94301
Computer Science Department
Stanford University
Stanford, California 94305

Michael Frank
Center for the Study of Language and Information
Stanford University
Stanford, California 94305

Michael P. Halpin
Development Systems Group
Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014

Mark C. Torrance
Computer Science Department
Stanford University
Stanford, California 94305

Search Lessons Learned from Crossword Puzzles

Abstract

The construction of a program that generates crossword puzzles is discussed. The aim of this research is to draw conclusions that apply to conjunctive search generally, and the experimental results obtained in the crossword-puzzle domain lead us to the following: (1) Lookahead is extremely important when solving conjunctive queries, (2) Compile-time control of search is far less effective than its run-time counterpart, and (3) Connectivity is best used as a backtracking heuristic and not as a heuristic that determines which of a particular set of conjuncts should be expanded next.

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 conjunctive search 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 illuminate the problems that will be encountered when solving declarative problems using large databases generally.[1]

In the next section, we will describe a translation of the crossword puzzle problem into a conventional one in deductive databases. Existing work on these problems is discussed in Section 3. In Section 3.1, we describe work on crossword puzzle generation itself, while in Section 3.2, we discuss work on the associated declarative tasks.

Our own approach is discussed in Section 4. Since we are interested principally in abstracting to the declarative domain lessons learned from the crossword puzzle problem, we have restricted our attention throughout to domain-independent heuristics such as the cheapest-first heuristic or connectivity. In Section 6.4, we very briefly discuss some domain-specific heuristics that might have been investigated.

[1]There is an additional feature that makes the problem interesting, but this feature unfortunately cannot be described effectively in print. It is easy to have a program that is generating a crossword display its activities graphically, so that the user can watch words being entered and erased as the program progresses through the search space. The result is that the difficulties encountered in the search are displayed with a clarity that we have seen nowhere else.

Figure 1: A typical puzzle

Section 5 contains our experimental data. For each combination of the heuristics discussed in Section 4, 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. We also describe some techniques that were tried but appeared *not* to lead to computational speedups, and go on in Section 6 to discuss some ideas that we chose not to investigate for one reason or another. Concluding remarks are contained in Section 7.

2 The declarative connection

To see the connection between the crossword puzzle problem and declarative issues, consider the small puzzle in Figure 1. This translates into the declarative query

$$\begin{gathered}\mathtt{word}_3(l_1, l_2, l_3) \wedge \mathtt{word}_3(l_4, l_5, l_6) \wedge \mathtt{word}_2(l_7, l_8) \wedge \\ \mathtt{word}_3(l_1, l_4, l_7) \wedge \mathtt{word}_3(l_2, l_5, l_8) \wedge \mathtt{word}_2(l_3, l_6),\end{gathered} \tag{1}$$

where we have used the predicate $\mathtt{word}_n(l_{i_1}, \ldots, l_{i_n})$ to indicate that the letters in squares i_1 through i_n spell out a word of length n.

Given this translation, the crossword puzzle problem has the following features that are not normally found in conventional conjunctive queries:

1. There are exactly 26 objects in the domain.

2. For each positive integer n, there is only one n-ary predicate, namely $\mathtt{word}_n$.

3. Each variable appearing in the query appears at most twice.

4. There are no "rules of inference" in the dictionary. The conjunctive subgoals are solved by lookup only.

5. There is a convention in crossword puzzle construction that the same word not be used more than once in a single puzzle.

The final condition can obviously be dropped when solving crossword problems, and the first two can be avoided by creating new dictionaries. If there are more (or less) than 26 objects in the domain, the "words" in the new dictionary will involve suitably more or less than 26 letters; if there are two predicates p_1 and p_2 of the same arity n, a new predicate p_2' can be introduced of arity $n' > n$ such that

$$p_2'(x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n'}) \equiv p(x_1, \ldots, x_n) \wedge x_n = x_{n+1} = \ldots = x_{n'}. \qquad (2)$$

The third condition can also be sidestepped. The easiest way to describe this is probably via an example, so we consider the conjunctive goal

$$a(x, y, z) \wedge b(x, y) \wedge c(x)$$

in which x appears three times. This can be rewritten as

$$a(x, y, z) \wedge b(x', y) \wedge c(x'') \wedge (x = x') \wedge (x = x''),$$

which can in turn be rewritten as in (2) as

$$a(x, y, z) \wedge b(x', y) \wedge c(x'') \wedge d(x, x', x'', x''')$$

where $d(x_1, x_2, x_3, x_4)$ holds if and only if $x_1 = x_2 = x_3 = x_4$.

Given these modifications, it is clear that *any* declarative query that involves lookup only (as opposed to inference) can be translated into a suitable crossword puzzle problem, and that the classes of crossword problems considered here is in fact equivalent to the class of database queries considered by Smith in [3].[2] Of course, the experimental data that we will present concerns only a conventional 26-letter dictionary, and it is not clear to what extent our experimental results will change if a dictionary such as that suggested by (2) is used.

3 Existing work

3.1 On crosswords

There is very little published work on automatic generation of crossword puzzles; the old paper by Mazlack [2] is the only work of which we are aware.

Mazlack discusses and dismisses the approach of completing the puzzle a word at a time (i.e., successively solving the subgoals in an expression such as (1)). Instead, he suggests completing the puzzle a letter at a time, essentially finding successive bindings for the various variables appearing in (1).

We have not chosen to pursue the letter-by-letter approach in this paper. For a start, such an approach would make it difficult to draw connections between our work and existing work on conjunctive query answering; it would also make it difficult for us to take advantage of progress in this latter field when developing domain-independent heuristics to help in the construction of crossword puzzles.

[2]Not quite; there are also topological considerations that might prevent a crossword puzzle from being realized on a two-dimensional grid. All of the algorithms we used were independent of such issues, however.

In addition, Mazlack's arguments against the word-by-word approach are principally experimental ones that we do not share. The bitwise hashing scheme that we use and discuss in Section 4.1 addresses his concerns regarding the amount of memory required by this approach, and the largest puzzle he solved successively (in some 1600 seconds of CPU time) is solved by our system in 4 seconds. Although there have obviously been tremendous advances in computing power since the mid 70's, this improvement in performance exceeds them.

3.2 On related declarative issues

The work on related declarative issues is far more substantial. There are essentially three commonly-accepted domain-independent heuristics for addressing problems of this sort:

1. The cheapest-first heuristic,
2. Connectivity, and
3. Smith's adjacency restriction [3].

Cheapest-first The cheapest-first heuristic suggests that when deciding which conjunct to expand next, one should expand the conjunct that is the most difficult to solve, in that it has no more solutions than any of the other remaining conjuncts.[3] Although it does not always lead to optimal conjunct orderings (as shown in [3]), this heuristic has the advantage of being cheap to compute and easy to apply. In the crossword-puzzle domain, it tends to lead to the early completion of long words (since there are not many possible choices for them) and the completion of words that are partially completed as the search proceeds (since the partial completion will drastically reduce the number of remaining choices). The justification for this heuristic is that solving difficult conjuncts early makes it more likely that the remaining conjuncts will have solutions at all, thereby reducing the need for backtracking.

When discussing the cheapest-first heuristic in [3], Smith makes two assumptions that we wish to avoid. First, he assumes that the choice for an ordering among the conjuncts is made at "compile time." In our domain, this means that the choice is made when the frame is first encountered, and before any of the words have been inserted. As we will see, there is compelling experimental evidence indicating that it is important to be able to select the order as the tree is actually being searched.

Smith's second assumption is a consequence of the first. Specifically, he uses statistical information to determine how many choices are likely to remain for any particular conjunct (i.e., word in the puzzle). Since there are 2139 4-letter words in the dictionary, Smith assumes that there will be 2139/26 choices remaining after the second position has been filled. Since he is choosing an ordering before actually filling in the puzzle, this is all the information available to him.

[3]The name "cheapest-first" refers to the fact that this is the conjunct for which it is the cheapest to enumerate all solutions.

If the order is chosen at "run time" (i.e., as the puzzle is completed), more information is available. Now, we may know that some particular 4-letter word has an `E` in the second position (281 possible completions) or that it has a `Q` there (only 1 completion). This allows us to use the cheapest-first heuristic much more accurately at run time than at compile time; one of the questions this paper is intended to answer is how important the difference is between these two approaches.

Connectivity The next best-known heuristic for ordering conjunctive subgoals is usually referred to as *connectivity*. Roughly speaking, the idea is to always solve a conjunct that involves a variable that was bound by the previous conjunct; the justification is that this ensures that one always backtracks usefully.

In the crossword puzzle domain, this suggests that we always fill in a word that intersects the previously completed word. If we are unable to fill a particular word, backtracking to an intersecting word is likely to introduce new possibilities for the word causing trouble.

What is happening here is that connectivity is being used to produce a cheap version of dependency-directed backtracking [4] that does not require us to maintain dependency information during the search process. The computational expense of maintaining justification information is often prohibitive; de Kleer, for example, has observed that it is *not* computationally practical to maintain a list of nogoods when using an ATMS.[4]

There are two points we would like to make here. The first is that the connectivity and cheapest-first heuristics are very similar in practice. The reason for this is that filling a word will generally make the crosswords to that word more difficult to complete, and is therefore likely to result in cheapest-first choosing a crossing word to process next in any event.

The second point to be made is that the purpose of connectivity is to ensure effective backtracking. Given this, it seems natural to use connectivity not when expanding a particular node of the search space, but only when the backtracking is actually performed. As an example, suppose that we fill some word w_1 in the puzzle, then fill an unrelated word w_2, and then notice that we are unable to fill the word w_3 that crosses w_1. The reason that connectivity is valuable is that it prevents us from trying another choice for w_2 when it is in fact w_1 (or something that preceded it) that is the source of the difficulty. Our proposed solution is simply to backtrack over w_2 and to replace w_1 itself. Although this forces us to regenerate w_2, the expense of doing so will in general be small, and this appears experimentally to be a far better choice than either dropping the cheapest-first heuristic or maintaining complete dependency information. We will have more to say about this in Sections 4.5, 5.2 and 5.3.

Adjacency In [3], Smith presents a provably correct restriction on the ordering of the conjuncts in a query of the form we are considering. This condition, which he calls the *adjacency restriction*, reduces the number of possible orderings that need to be considered at compile time in order to determine an optimal ordering. The reason the adjacency restriction is of interest is that there are situations where both the cheapest-first and connectivity heuristics lead to orderings that are clearly suboptimal.

[4]Personal communication.

There are two separate reasons that we chose not to use Smith's results in this work. The first is that even with the adjacency restriction, the number of possible orderings is prohibitively large. The simpler of the two 13×13 puzzles considered in Section 5.2 contains 64 words; the number of possible orderings of these words that satisfy the adjacency restriction is approximately 10^{77}. Even for the puzzle of size 5×5, there are over 50,000 orderings.

In addition, Smith's techniques must be used at compile time only. As we will see, there is good experimental evidence for the conclusion that run time information about conjunct difficulty plays a key role in the sorts of problems we are considering.

4 Techniques used

Having described the basic crossword puzzle problem and existing work on it and related issues, let us now discuss the techniques that we used to attack it.

4.1 Pointers

The first problem that needs to be addressed is that of dictionary access. It is important that we be able to find as rapidly as possible all 5-letter words with an `A` in the first position and an `E` in the fourth, and so on.

The natural way in which to do this is to have lists of 5-letter words with `A`'s in the first position and of 5-letter words with `E`'s in the fourth position, and to intersect them. In order to make the intersection operation as fast as possible, we set up a list of the 3102 5-letter words in the dictionary, and then use a 3102-bit integer to indicate which of these words have an `A` in the first position. More generally, we construct a 26×5 array of 3102-bit integers; if bit b is on in the (l, p)'th entry of this array, it means that word b has the letter l in the position p. The bitwise conjunction of the $(1,1)$ and $(5,4)$ entries in this array now corresponds to words that have an `A` in the first position and an `E` in the fourth.

Common Lisp is well suited to support manipulations of this sort [5]. Integers of arbitrary length are supported, and the Common Lisp functions `integer-length` and `logcount` can be used to find the first word matching a given pattern and to count the number of such words respectively.

Note also that this hashing scheme is fairly compact. The bit patterns for the entire dictionary use approximately 560K bytes; the bit patterns for all words of length 4 or shorter (the longest word appearing in any of Mazlack's puzzles is 4 letters long) can be stored in 35K bytes.

4.2 Lookahead

The most important single technique used in the search is a simple one that has not been discussed elsewhere – lookahead. When a new word is inserted into the puzzle, all of the unfilled crossing words are examined, and a list of possible choices for each of these words is updated (this list is in the form of a many-bit integer, of course). If no possibilities remain for some crossing word, the word that has just been inserted is retracted and something else is tried. Whatever the order in which the words are inserted, this simple check prunes large portions of the search space and is needed if the problem is to be manageable.

4.3 Cheapest-first heuristic

An additional advantage of maintaining accurate lists of possible entries for each word slot is that the program knows, at any given time, *exactly* how many words are still candidates for each position. This means that it is possible to apply the cheapest-first heuristic much more accurately than at compile time – the next word filled is always the one that really *is* the most constrained, and not simply the one that one might expect to be the most constrained because of the number of letters that have been filled in (as opposed to the values chosen for these letters).

In fact, both versions of cheapest-first were used during different runs of the program. The "statistical" version is similar to that used by Smith in the compile time work, and assumes that if there are $w(n)$ words of length n in the dictionary, then the number of words of length n in which k of the letters have already been filled is given by $w(n)/26^k$. The "exact" version of the cheapest-first heuristic is one in which this approximation is not made, but precise values are used as described in the previous paragraph.

4.4 Intelligent instantiation

There is still another advantage to maintaining precise information about the number of possible choices from crossing words when a given word is chosen. If we are filling a particular word slot, it is advantageous to fill it with a word *that restricts the possible choices for the crossing words as little as possible.* Why use a word with a `Q` when one with an `S` could be used instead? This, too, is a technique that has not been explored elsewhere.

This idea was implemented as follows: Suppose that the cheapest-first heuristic has been used to decide what word to fill next. 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 in order to obtain a small increase in the size of the subsequent search space is unlikely to be justified. Some experimentation indicated that $k = 10$ was a reasonable value, and this is the value used in Section 5.2, where it is compared with the customary choice $k = 1$. The parameter k was called `min-look` in the implementation and this is how we will refer to it in Section 5.2.

4.5 Connectivity

Finally, there is the connectivity heuristic. As mentioned in Section 3.2, we use connectivity as a backtracking heuristic only. Even so, there are two possible ways in which to do it:

In the more conventional approach, we backtrack to a word that can be proved to be relevant to the problem encountered. If we fill word w_1, then an unrelated word w_2, and then find that we cannot fill the word w_3 that crosses word w_1, we would, as discussed in Section 3.2, backtrack directly to w_1 instead of trying a new choice for w_2.

A more subtle approach is to determine what *letter* (or letters) in w_1 is causing the problem, and to then ensure that the new choice for w_1 does not reproduce the difficulty. This is an idea that has an obvious analog in declarative problems generally – instead of simply backing up to a relevant conjunct, we back up to this conjunct and then ensure that the new solution binds a relevant variable to a new value, so that the problem is not repeated.

In the next section, we will refer to the possible choices as "none" (no connectivity used at all), "simple" (backtrack to the relevant problem, but make no effort to ensure that the difficulty has been addressed) and "smart" (backtrack to the problem and make sure that some relevant letter changes value).

5 Experimental results

5.1 Frames used and raw data

In order to evaluate the usefulness of the ideas in the last section, the four puzzles appearing in Figure 2 were solved by the program. The program always used lookahead 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 statistical approximation was used, or `exact`, indicating that the exact value was used.

2. If `cheapest-first` was `exact`, `min-look` could be either 1 (always use the first acceptable word) or 10 (use the best choice among the first ten acceptable words).

3. `connected-backtrack` could be either `none`, `simple` or `smart`, 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 3; the times reported are in seconds for a Symbolics 3620 with 2 megawords of memory. 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

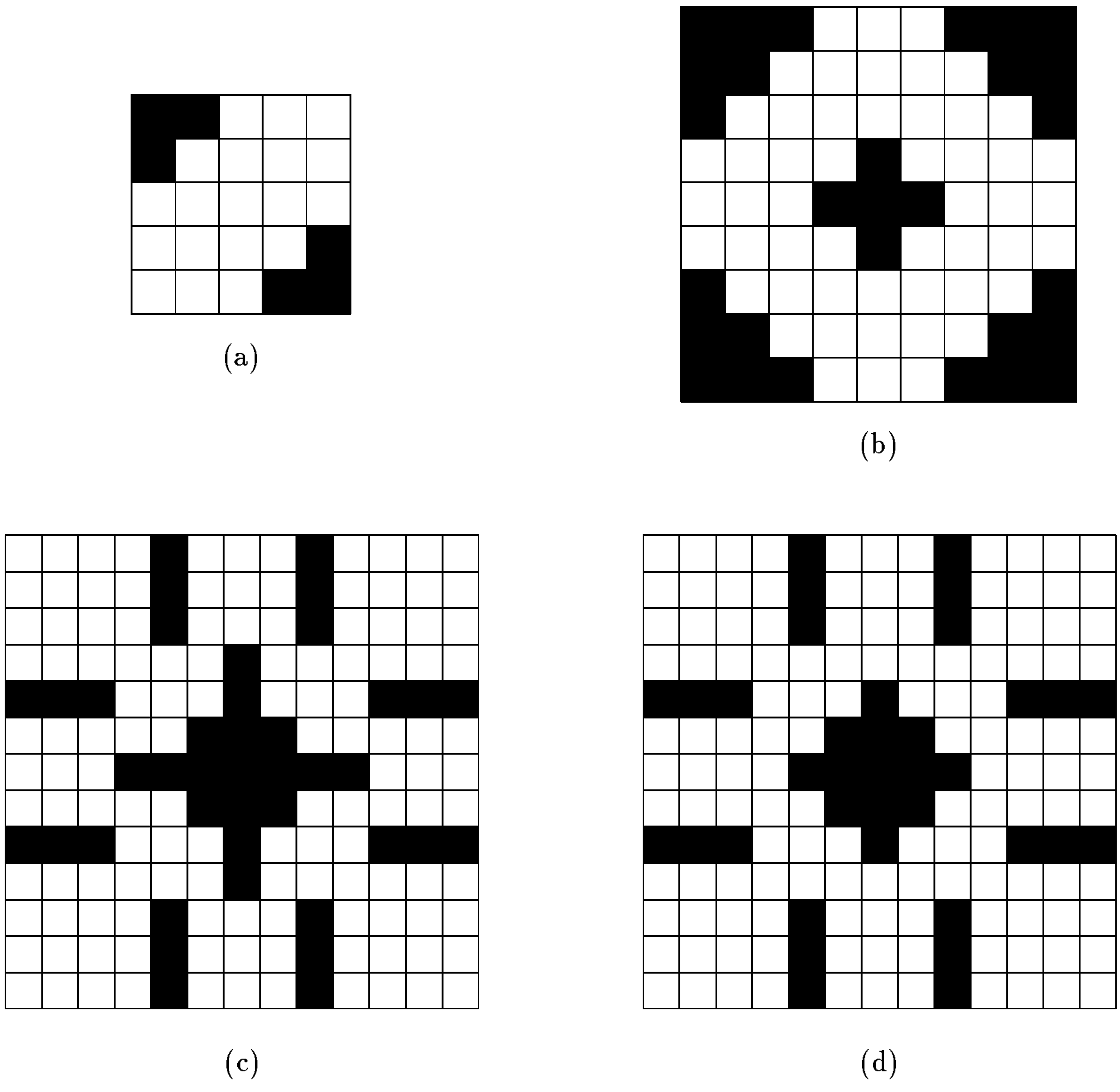

Figure 2: Test puzzles

information is reported for these parameter choices. The most difficult puzzle ((d) in Figure 2) was solved in only 8 of 10 cases with `min-look` set to 1.

5.2 Analysis

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

Cheapest-first heuristic It is apparent even for the 5×5 puzzle that runtime information plays an important role in the control of search. 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; we will return to this point briefly in Section 6.

Connectivity Connectivity is another heuristic the value of which is only apparent on the larger puzzles; on smaller ones, the relationship between the cheapest-first heuristic and connectivity served many of the same purposes. "Smart" backtrack especially 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 declarative domains will share these features, but it is not unreasonable to think that they will.

As an example, suppose that we are trying to find a friend who will sell us tickets to a 49er's game. Tom has season tickets, but is a big fan and won't want to give them up; Bill only has tickets to a single game, but might be prepared to let them go. It is useful to be able to realize that we need to find a friend other than Tom who has tickets; if we don't add this constraint to the problem, we are likely to extend our search considerably.

5.3 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.

	Cheapest first	min-look	Connected backtrack	Time (sec)
Puzzle (a)	exact	1	none	0.505
	exact	1	simple	0.533
	exact	1	smart	0.605
	exact	10	smart	0.607
	exact	10	none	0.612
	stat	–	smart	0.693
	exact	10	simple	0.719
	stat	–	simple	1.015
	stat	–	none	1.041
Puzzle (b)	exact	10	none	2.363
	exact	10	smart	2.453
	exact	10	simple	2.497
	exact	1	smart	4.967
	exact	1	none	6.327
	exact	1	simple	6.587
	stat	–	smart	17.022
	stat	–	simple	17.170
	stat	–	none	31.526
Puzzle (c)	exact	10	smart	11.904
	exact	10	simple	15.539
	exact	1	smart	17.668
	exact	1	simple	17.798
Puzzle (d)	exact	10	smart	71.693
	exact	1	smart	408.555*
	* Completed on only 8 of 10 attempts			

Figure 3: Test results

				R
				E
				A
T	R	E	A	D
				Y

Figure 4: Is multiple lookahead worthwhile?

Lookahead to greater depth One thing that was tried was lookahead to depths greater than one. For example, when a word is inserted, this will affect the crosswords allowed – that is the lookahead that we've already described. Those crosswords will in turn affect other crosswords, and so on. Is it worthwhile to compute the impact of a particular word to greater depth?

It seems at first that it should be. Consider the puzzle in Figure 4, 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 4 would be withdrawn immediately.

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), and then to use this information to prune the set of possibilities for w_2. This 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 more than a single level of lookahead draws conclusions no more effectively than its backward counterpart.

Tree reordering while backtracking Another apparently promising idea is the following: Suppose that we have backtracked over a word w because it was not relevant to the problem that caused the backtrack in the first place. It now seems reasonable to put w back into the puzzle before resuming the search, essentially modifying the order in which the search is conducted so that we can reuse the information that the connectivity heuristic would otherwise lose.

Unfortunately, things are not nearly so simple as this. Just because removing w doesn't 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 Finally, we 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 the 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.

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

6 Techniques not tried

Finally, there were a variety of techniques that strike us as promising, but which we did not take the time to investigate. These are as follows:

6.1 Simulated annealing

Mazlack suggests in [2] that if a variety of choices have been tried at a point in the search space and none has proven successful, the search backtrack further on the assumption that the problem lies elsewhere (Mazlack backtracks after seven failed attempts, for example).

This approach has the advantage that it avoids situations where the machine has failed to recognize a problem that is preventing it from completing the puzzle; the difficulty is that the method, as described, is incomplete. Many of the puzzles that were successfully completed by our program appeared to be thrashing in just this fashion, but finally managed to terminate successfully.

In any event, if this sort of a technique is to be used, it seems natural to combine it with some of the ideas underlying simulated annealing [1]. One might envision the puzzle generator as having a "temperature" that controlled how prepared it was to backtrack significantly without exhaustively searching a portion of the search space, and how far back it went when it did so. As the puzzle "cooled," the search would become gradually more conventional.

6.2 Undirected search

Another idea is to abandon the view of the search space as a tree, and to simply fill words into slots and to withdraw them when they cause a problem. The difficulty with this is that it is hard to know how to avoid returning a particular problematic word to a particular slot, causing a loop. One possibility is to *never* reuse a word in a position where it has caused difficulty, trusting the number of solutions to be large enough that they are not eliminated by this pruning. As with the ideas discussed in the previous subsection, this approach is obviously incomplete but might well be viable provided that the set of solutions, while sparse in the entire search space, was fairly large.

6.3 Metalevel work

The remaining domain-independent possibility of which we are aware is that of doing "metalevel" work to determine dynamically what values should be assigned to search parameters. We have already suggested metalevel analysis in order to choose a value for `min-look`; it also seems reasonable to use repeated backtracks to a particular point (as discussed in Section 6.1) as an indication that metalevel work (in this case, some sort of justification analysis) should be done to identify the source of the difficulty. Both of these ideas seemed to us to be beyond the scope of our investigations.

6.4 Domain-specific information

Finally, there is the possibility of using domain-specific heuristics that are tailored to the crossword puzzle problem specifically. David Smith, for example, has suggested that the selection of word to be completed next not be based only on the number of choices for each slot, but should include considerations about the expected difficulty of neighboring slots.[5] By doing this, it is likely that results similar to those of the complete runtime adjacency computation could be obtained, but at far less cost.

As already discussed, the reason we chose not to pursue these ideas is that we would not know what to make of the results – as mentioned in the introduction, our interest in crossword puzzle construction is motivated by our belief that lessons learned from the solution to this problem can be used in work on declarative search generally.

7 Conclusion

Summarizing, the conclusions that we have drawn from the experimental data in Figure 3 are the following:

1. Some form of lookahead and some application of the cheapest-first heuristic are necessary if conjunctive queries in large databases are to be solved effectively.

2. It is far more efficient to employ the cheapest-first heuristic using the exact information available at runtime than to use the statistical information available at compile time. This information can also be used to select among competing solutions to a particular conjunctive subgoal.

3. The connectivity heuristic is best used during backtrack only. The principal reason for this is that the cheapest-first heuristic serves many of the same purposes if used to select the conjunct to be solved next. In addition, it is possible to use the connectivity heuristic while backtracking to ensure that subsequent solutions to a particular conjunct actually address a difficulty that was found later 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, and the students of his introductory artificial intelligence class for many enlightening discussions.

References

[1] L. Davis. *Genetic Algorithms and Simulated Annealing.* Pitman, London, 1987.

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

[5] Personal communication.

[3] D. E. Smith and M. R. Genesereth. Ordering conjunctive queries. *Artificial Intelligence*, 26(2):171–215, 1985.

[4] 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.

[5] G. L. Steele, Jr. *Common Lisp: The Language.* Digital Press, Billerica, MA, 1984.