PROVERB: The Probabilistic Cruciverbalist

Greg A. Keim, Noam M. Shazeer, Michael L. Littman,
Sushant Agarwal, Catherine M. Cheves, Joseph Fitzgerald,
Jason Grosland, Fan Jiang, Shannon Pollard, Karl Weinmeister
Department of Computer Science
Duke University, Durham, NC 27708-0129
contact: {keim, noam, mlittman}@cs.duke.edu

Abstract

We attacked the problem of solving crossword puzzles by computer: given a set of clues and a crossword grid, try to maximize the number of words correctly filled in. In our system, "expert modules" specialize in solving specific types of clues, drawing on ideas from information retrieval, database search, and machine learning. Each expert module generates a (possibly empty) candidate list for each clue, and the lists are merged together and placed into the grid by a centralized solver. We used a probabilistic representation throughout the system as a common interchange language between subsystems and to drive the search for an optimal solution. PROVERB, the complete system, averages 95.3% words correct and 98.1% letters correct in under 15 minutes per puzzle on a sample of 370 puzzles taken from the New York Times and several other puzzle sources. This corresponds to missing roughly 3 words or 4 letters on a daily 15×15 puzzle, making PROVERB a better-than-average cruciverbalist (crossword solver).

Introduction

Proverbs 022:021 *That I might make thee know the certainty of the words of truth...*

Crossword puzzles are attempted daily by millions of people, and require of the solver both an extensive knowledge of language, history and popular culture, and a search over possible answers to find a set that fits in the grid. This dual task, of answering natural language questions requiring shallow, broad knowledge, and of searching for an optimal set of answers for the grid, makes these puzzles an interesting challenge for artificial intelligence. In this paper, we describe PROVERB, the first broad-coverage computer system for solving crossword puzzles[1]. While PROVERB's performance is well below that of human champions, it exceeds that of casual human solvers, averaging over 95% words correct over a test set of 370 puzzles.

We will first describe the problem and some of the insights we gained from studying a large database of crossword puzzles; these motivated our design choices. We will then discuss our underlying probabilistic model and the architecture of PROVERB, including how answers to clues are suggested by expert modules, and how PROVERB searches for an optimal fit of these possible answers into the grid. Finally, we will present the system's performance on a test suite of daily crossword puzzles and on 1998 tournament puzzles.

The Crossword Solving Problem

The solution to a crossword puzzle is a set of interlocking words (*targets*) written across and down a square grid. The solver is presented with an empty grid and a set of clues; each clue suggests its corresponding target. Some clue-target pairs are relatively direct: ≺Florida fruit [6]: `orange`≻[2], while others are more oblique and based on word play: ≺Where to get a date [4]: `palm`≻. Clues are between one and a dozen or so words long, averaging about 2.5 words per clue.

To solve a crossword puzzle by computer, we assume that we have both the grid and the clues in machine readable form, ignoring the special formatting and unusual marks that sometimes appear in crosswords. The *crossword solving problem* is the task of returning a grid of letters, given the numbered clues and a labeled grid.

In this work, we focus on American-style crosswords, as opposed to British-style or cryptic crosswords. By convention, all targets are at least 3 letters in length and long targets can be constructed by stringing multiple words together: ≺Don't say another word [13]: `buttonyourlip`≻. Each empty square in the grid must be part of a down target and an across target.

As this is largely a new problem domain, distinct from crossword-puzzle creation (Ginsberg *et al.* 1990), we wondered how hard crossword solving really was. To

[1]Crossword Maestro is a commercial solver for British-style crosswords published by Genius 2000 Software. It is intended as a solving aid, and while it appears quite good at thesaurus-type clues, in informal tests it did poorly at grid filling (under 5% words correct).

[2]Target appears in fixed-width font; all examples are taken from our crossword database (the CWDB). We will note the target length following sample clues in this paper to indicate a complete specification of the clue.

Source	Puzzles CWDB	Train	Test
New York Times (NYT)	792	10	70
Los Angeles Times (LAT)	439	10	50
USA Today (USA)	864	10	50
Creator's Syndicate (CS)	207	10	50
CrosSynergy Syndicate (CSS)	302	10	50
Universal Crossword (UNI)	262	10	50
TV Guide (TVG)	0	10	50
Dell	969	0	0
Riddler	764	0	0
Other	543	0	0
Total	**5142**	**70**	**370**

Table 1: The crossword database (CWDB) was drawn from a number of machine-readable sources. The TV Guide puzzles were added after finalizing the CWDB.

gain some insight into the problem, we studied a large corpus of existing puzzles. We collected 5142 crossword puzzles from a variety of sources, summarized in Table 1. Several are online versions of daily print newspaper puzzles (The New York Times, The Los Angeles Times, The USA Today, TV Guide), from online sites featuring puzzles (Dell, Riddler) or from syndicates specifically producing for the online medium (Creator's Syndicate, CrosSynergy Syndicate). These puzzles constitute a crossword database (the CWDB) of around 350,000 clue-target pairs (over 250,000 unique), which served as a potent knowledge source for this project.

Novelty

Human solvers improve with experience, in part because particular clues and targets tend to recur. For example, many human solvers will recognize ≺Great Lake [4]: `erie`≻ to be a common clue-target pair in many puzzles[3]. The CWDB corresponds to the number of puzzles that would be encountered over a fourteen-year period, at a rate of one puzzle a day.

What percentage of targets and clues in a new puzzle presented to our system will be in the CWDB—how novel are crossword puzzles? In Figure 1, we graph the probability of novel targets, clues, clue-target pairs, and clue words as we increase the size of the CWDB.

After randomizing, we looked at subsets of the CWDB ranging from 5,000 clues to almost 350,000. For each subset, we calculated the percentage of the item (target, clue, clue-target, clue word) that are unique. This is an estimate for the likelihood of the next item being novel. Given the complete CWDB (344,921 clues) and a new puzzle, we would expect to have seen 91% of targets, 50% of clues, and 34% of clue-target pairs. We would also expect to have seen 96% of the words appearing in the clues. The CWDB contains a tremendous amount of useful domain-specific information.

[3]The five most common targets in the CWDB are `era`, `ore`, `area`, `erie` and `ale`. The target `erie` appears in over 7% of puzzles.

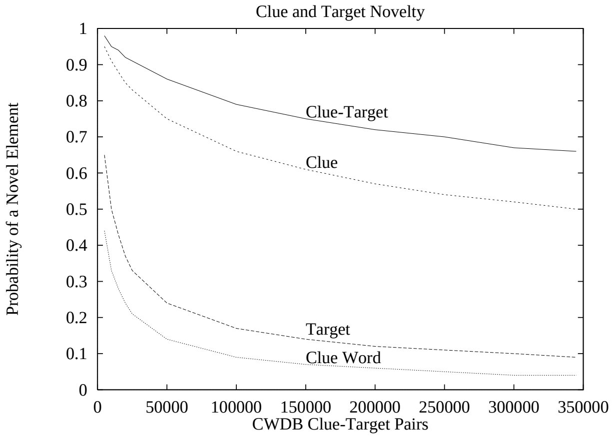

Figure 1: Clue and target novelty decreases with the size of the CWDB.

	Mon	Tue	Wed	Thu	Fri	Sat
#puz	89	92	90	91	91	87
#clues	77.3	77.2	76.7	74.7	70.0	70.2
3	16.5	18.2	17.5	18.6	17.3	16.3
4–5	64.6	61.1	62.5	54.7	44.2	40.2
6–10	15.8	17.7	16.9	23.1	35.2	41.7
11–15	3.1	2.9	3.2	3.7	3.3	1.9
Blank	8.4	8.0	6.4	6.4	5.2	4.8
Blank & " "	3.1	3.1	2.7	2.2	2.0	1.7
Single Word	15.6	14.9	16.0	17.2	16.9	20.6
(Year)	1.4	1.6	1.9	2.1	2.5	2.7
Final '?'	0.8	1.2	2.5	3.2	3.5	2.6
X, in a way	0.0	0.1	0.2	0.4	0.6	0.8

Table 2: NYT clue statistics vary by day of week.

The New York Times Crossword Puzzle

The New York Times (NYT) crossword is considered by many to be the premiere daily puzzle. NYT editors attempt to make the puzzles increase in difficulty from easy on Monday to very difficult on Saturday and Sunday. We hoped that studying the Monday-to-Saturday trends in the puzzles might provide insight into what makes a puzzle hard for humans.

In Table 2, we show how the distributions of clue types change day by day. For example, note that some "easier" clues, such as fill-in-the-blank clues ≺Mai ___ [3]: `tai`≻ get less and less common as the week goes on. In addition, clues with a trailing question mark (≺T.V. Series? [15]: `sonyrcamagnovox`≻), which is often a sign of a themed or pun clue, get more common. The distribution of target lengths also varies, with words in the 6 to 10 letter range becoming much more common from Monday to Saturday. Sunday is not included in the table as its larger (up to 23×23 versus 15×15 for the other days) makes it difficult to compare.

Categories of Clues

In the common syntactic categories shown in Table 2, such as fill-in-the-blank and quoted phrases, clue structure leads to simple ways to answer those clues. For example, given the clue ≺_____ miss [5]: `hitor`≻, a scan through text sources could look for all 9-letter phrases that match on word boundaries and known letters. With the clue ≺Map abbr. [3]: `rte`≻, a list of likely abbreviations could be returned.

In addition, a number of non-syntactic, *expert* categories stand out, such as synonyms (≺Covered [5]: `awash`≻), kind-of (≺Kind of duck or letter [4]: `dead`≻), movies (≺1954 mutant ants film [4]: `them`≻), geography (≺Frankfurt's river [4]: `oder`≻), music (≺'Upside down' singer [4]: `ross`≻) and literature (≺Carroll character [5]: `alice`≻).

There are also clues that do not fit simple patterns, but might be solved by existing information retrieval techniques (≺Nebraska tribesman [4]: `otoe`≻). The many different sources of information that can be brought to bear to solve clues led us to create a two-stage architecture for the solver: one consisting of a collection of general and special-purpose candidate-generation modules, and one that combines the results from these modules to generate a solution to the puzzle. This decentralized architecture allowed a relatively large group of contributors (approximately ten people) to create modules using techniques ranging from generic word lists to highly specific modules, from string matching to general-purpose information retrieval. The next section describes PROVERB's modular design.

Architecture

Figure 2 illustrates the components of PROVERB. Given a puzzle, the *Coordinator* separates the clues from the grid and sends a copy of the clue list (with target lengths) to each *Expert Module*. The expert modules generate probability-weighted candidate lists, in isolation from the grid constraints. Expert modules are free to return anything from no candidates for any clue to or 10,000 for every one. The collection of candidate lists is then reweighted by the *Merger* to compensate for differences in module weighting, and combined into a single list of candidates for each clue. Finally, the *Solver* takes these weighted lists and searches for the best solution it can find that also satisfies the grid constraints.

The *Implicit Distribution Modules* are used by the solver, and are described in a later section.

The Probabilistic Model

To unify the candidate-generation modules, it is important to first understand our underlying assumptions about the crossword-puzzle problem. First, we imagine that crossword puzzles are created by repeatedly choosing words for the slots according to a particular creator's distribution (ignore clues and crossing constraints for now). After choosing the words, if the crossing constraints are satisfied, then the creator keeps the puzzle.

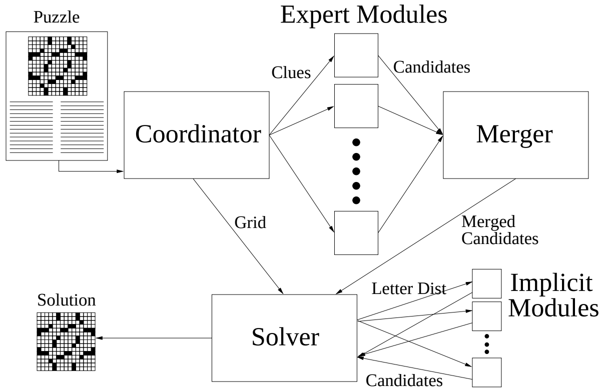

Figure 2: PROVERB consists of a set of independent communicating programs.

Otherwise, the creator draws again. Normalizing to account for all the illegal puzzles generated gives us a probability distribution over legal puzzles.

Now, suppose that for each slot in a puzzle, we had a probability distribution over possible words for the slot given the clue. Then, we could try to solve one of a number of probabilistic optimization problems to produce the "best" fill of the grid. In our work, we define "best" as the puzzle with the maximum expected number of targets in common with the creator's solution: the maximum expected overlap (Shazeer, Littman, & Keim 1999). We will discuss this optimization more in a following section, but for now it is important only to see that we would like to think of candidate generation as establishing probability distributions over possible solutions.

We will next discuss how individual modules can create approximations to these distributions, how they are combined into a unified distributions.

Candidate Generation

The first step is to have each module generate candidates for each clue, given the target length. Each module returns a confidence score (how sure it is that the answer lies in its list), and a weighted list of possible answers. For example, given the clue ≺Farrow of 'Peyton Place' [3]: `mia`≻, the movie module returns:

1.0: 0.909091 `mia`, 0.010101 `tom`, 0.010101 `kip`, ⋯
⋯, 0.010101 `ben`, 0.010101 `peg`, 0.010101 `ray` .

The module returns a 1.0 confidence in its list, and gives higher weight to the person on the show with the given last name and lower weight to other cast members.

Note that most of the modules will not be able to generate actual probability distributions for the targets, and will need to make approximations. The merging step discussed next will attempt to account for the error in these estimates by testing on training data, and adjusting scaling parameters to compensate. It is important for modules to be consistent, and to give more likely candidates more weight. Also, the better control a module exerts over the overall confidence score when uncertain, the more the merger will "trust" the module's predictions.

In all, we built 30 different modules, many of which are described briefly below. To get some sense of the contribution of the major modules, Table 3 summarizes performance on 70 training puzzles, containing 5374 clues. These puzzles were drawn from the same sources as the test puzzles, ten from each. For each module, we list several measures of performance: the percentage of clues that the module guessed at (**Guess**), the percentage of the time the target was in the module's candidate list (**Acc**), the average length of the returned lists (**Len**), and the percentage of clues the module "won"—it had the correct answer weighted higher than all other modules (**Best**). This final statistic is an important measure of the module's contribution to the system. For example, the WordList-Big module generates over 100,000 words for some clues, so it often has the target in its list (97% of the time). However, since it generates so many, the individual weight given to the target is usually lower than that assigned by other modules, and, thus, it is the best predictor only 0.1% of the time.

We conducted a series of "ablation" tests in which we removed each module one at a time, rerunning the 70 training puzzles with the other $n-1$ modules. No one module's removal changed the overall percentage of words correct by more than 1%, which implies that there is considerable overlap in the coverage of the modules. We also tried removing all modules that relied in any way on the CWDB, which reduced the average percentage words correct from 94.8% to 27.1%. On the other hand, using *only* the modules that exclusively used the CWDB yielded a reduction to only 87.6% words correct. Obviously, in the current system, the CWDB plays a significant role in the generation of useful candidate lists.

Another way of looking at the contribution of the modules is to consider the probability assigned to each target given the clues. Ideally, we would like all targets to have probability 1. In general, we want to maximize the product of the probabilities assigned to the targets, since this quantity is directly related to what the solver will be maximizing. In Figure 3, the top line represents the probability assigned by the Bigram module (described later). This probability is low for all targets, but very low for the hard targets. As we add groups of modules, the effect on the probabilities assigned to targets can be seen as a lowering of the curve, which corresponds to assigning more and more probability to the target. Note the large increase due to the Exact Match module. Finally, notice that there is a segment that the modules do very poorly on—the targets that only Bigram returns. We will later describe extensions to the system that help with this range.

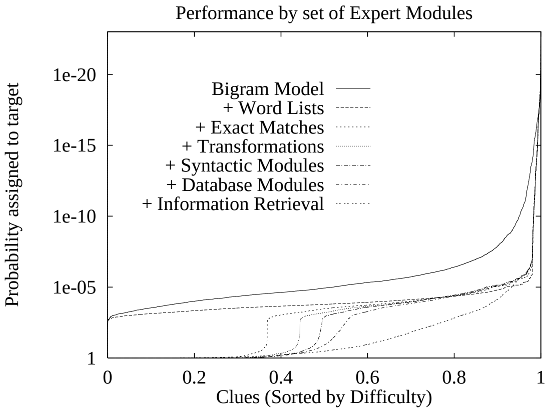

Figure 3: The cumulative probability assigned as module groups are added shows that different module types vary in contribution. Lines are sorted independently.

Word List Modules

WordList, WordList-Big ignore their clues and return all words of the correct length from several dictionaries. WordList contains a list of 655,000 terms from sources including online texts, encyclopedias and dictionaries. WordList-Big contains everything in WordList, as well as many constructed 'terms', produced by combining related entries in databases. This includes combining first and last names, as well as merging adjacent words from clues in the CWDB. WordList-Big contains over 2.1 million terms.

WordList-CWDB contains the 58,000 unique targets in the CWDB, and returns all targets of the appropriate length, regardless of the clue. It weights them with estimates of their "prior" probabilities as targets of arbitrary clues.

CWDB-Specific Modules

Exact Match returns all targets of the correct length associated with this clue in the CWDB. Confidence is based on a Bayesian calculation involving the number of exact matches of correct and incorrect lengths.

Transformations learns a set of textual transformations which, when applied to clue-target pairs in the CWDB, generates other known clue-target pairs. When faced with a new clue, it applies all applicable transformations and returns the results, weighted based on the previous precision/recall of these transformations. Transformations include word substitution, removing one phrase from the beginning or end of a clue and adding another phrase to the beginning or end of the clue, depluralizing a word in the clue and pluralizing the associated target, and others. The following is a list of several non-trivial examples from the tens of thousands of transformations learned:

nice $X \leftrightarrow X$ in france	X starter $\leftrightarrow$ prefix with X
X for short $\leftrightarrow X$ abbr	X city $\leftrightarrow X$ capital

Information Retrieval Modules

Crossword clues present an interesting challenge to

Module	Guess	Acc	Len	Best
Bigram	100.0	100.0	-	0.1
WordList-Big	100.0	97.2	$\approx 10^5$	1.0
WordList	100.0	92.6	$\approx 10^4$	1.7
WordList-CWDB	100.0	92.3	$\approx 10^3$	2.8
ExactMatch	40.3	91.4	1.3	35.9
Transformation	32.7	79.8	1.5	8.4
KindOf	3.7	62.9	44.7	0.8
Blanks-Books	2.8	35.5	43.8	0.1
Blanks-Geo	1.8	28.1	60.3	0.1
Blanks-Movies	6.0	71.2	35.8	3.2
Blanks-Music	3.4	40.4	39.9	0.4
Blanks-Quotes	3.9	45.8	49.6	0.1
Movies	6.3	66.4	19.0	2.2
Writers	0.1	100.0	1.2	0.1
Compass	0.4	63.6	5.9	0.0
Geography	1.8	25.3	322.0	0.0
Myth	0.1	75.0	61.0	0.0
Music	0.9	11.8	49.3	0.0
WordNet	42.8	22.6	30.0	0.9
WordNetSyns	11.9	44.0	3.4	0.9
RogetSyns	9.7	42.9	8.9	0.4
MobySyns	12.0	81.6	496.0	0.4
Encyclopedia	97.9	32.2	262.0	1.3
LSI-Ency	94.7	43.8	995.0	1.0
LSI-CWDB	99.1	77.6	990.0	1.2
PartialMatch	92.6	71.0	493.0	8.1
Dijkstra1	99.7	84.8	620.0	4.6
Dijkstra2	99.7	82.2	996.0	8.7
Dijkstra3	99.5	80.4	285.0	13.3
Dijkstra4	99.5	80.8	994.0	0.1

Table 3: Performance on 70 puzzles (5374 clues) shows differences in the number of targets returned (**Len**) and contribution to the overall lists (**Best**). Also measured but not shown are the implicit modules.

traditional information retrieval (IR) techniques. While queries of similar length to clues have been studied, the "documents" to be returned are quite different (words or short sequences of words). In addition, the queries themselves are often purposely phrased to be ambiguous, and never share words with the "documents" to be returned. Despite these differences, it seemed natural to try a variety of IR techniques.

Encyclopedia searches an online encyclopedia. For each query term, the module computes a distribution of terms "close" to the query term in the text. A term is counted $10 - k$ times in this distribution for every time it appears at a distance of $k < 10$ words away from the query term. A term is also counted once if it appears in an article for which the query term is in the title, or vice versa. Terms are assigned scores proportional to their frequencies in the "close" distribution, divided by their frequency in the corpus. The distribution of scores is normalized to 1. If a query contains multiple terms, the score distributions are combined linearly according to the log inverse frequency of the query terms in the corpus with very common terms such as "as" and "and" ignored.

Partial Match uses the standard vector space model (Salton & McGill 1983), defined by a vector space with one dimension for every word in the dictionary. A clue is represented as a vector in this space. For each word w a clue contains, it gets a component in dimension w of magnitude $-\log(\text{frequency}(w))$.

For a clue c, the module find all clues in the CWDB that share words with c. The target of each such clue is given a weight based on the dot product of the clue with c. The assigned weight is geometrically interpolated between 1/size(dictionary) and 1 based on this dot product.

LSI, or latent semantic indexing, is an extension of the vector space model that uses singular value decomposition to identify correlations between words. LSI has been successfully applied to the related problem of synonym selection on a standardized test (Landauer & Dumais 1997). LSI modules were trained on the CWDB (all clues with the same target were treated as a document) and on an online encyclopedia.

Dijkstra Modules derive from the intuition that related words co-occur with one another or co-occur with similar words, suggesting a measure of relatedness based on graph distance. From a set of text databases, the module builds a weighted directed graph on the set of all terms. Each database d and pair of terms (t, u) that co-occur in the same document produce an edge from t to u with weight

$$-\log\left(\frac{\#\text{ documents in } d \text{ containing } t \text{ and } u}{\#\text{ documents in } d \text{ containing } t}\right).$$

For a one-word clue t, the modules assign a term u a score of $-\log(\text{fraction of documents containing } t) - \text{weight}(\text{minimum weight path } t \to u)$.

The module finds the highest scoring terms with a shortest-path search. Multi-word clues are scored by summing the results for their individual terms. The four Dijkstra modules use variants of this technique.

An encyclopedia index, two thesauri, a database of wordforms and the CWDB were used as databases. Littman, Keim, & Shazeer (1999) provide examples.

Database Modules

Movie uses the Internet Movie Database (`www.imdb.com`), an online resource with a wealth of information about all manner of movies and T.V. shows. This module looks for a number of patterns in the clue (e.g. quoted titles as in $\prec$'Alice' star Linda [5]: `lavin`$\succ$, or Boolean operations on names as in $\prec$Cary or Lee [5]: `grant`$\succ$), and formulates queries to a local copy of the database.

Music, Literary, Geography use simple pattern matching of the clue (looking for keywords "city", "author", "band" and others as in $\prec$Iowa city [4]: `ames`$\succ$) to formulate a query to a topical database. The literary database is culled from both online and encyclopedia resources. The geography database is from the Getty Information Institute, with additional data supplied from online lists.

Synonyms are found by four distinct modules, based on three different thesauri. Using the WordNet (Miller *et al.* 1990) database, one module looks for root forms of words in the clue, and then finds a variety of related words (e.g. ≺Stroller [6]: `gocart`≻). In addition, a type of relevance feedback is used to generate lists of synonyms of synonyms. Finally, if necessary, the forms of the related words are coverted back to the form of the original clue word (number, tense, etc.): ≺Contrives [7]: `devises`≻.

Syntactic Modules

Fill-in-the-Blanks constitute over 5% of clues in the CWDB. These modules find string patterns in music, geography, movies, literary and quotes databases: ≺'Time ___ My Side' (Stones hit) [4]: `ison`≻.

KindOf clues are similar to fill-in-the-blank clues in that they involve pattern matching over short phrases. We identified over 50 cues that indicate a clue of this type, for example, "starter for" (≺Starter for saxon [5]: `anglo`≻), and "suffix with" (≺Suffix with switch or sock [4]: `eroo`≻).

Merging Candidate Lists

After each expert module has generated a weighted candidate list, Proverb must somehow merge these into a unified candidate list with a common weighting scheme for the solver. This problem is similar to the problem facing meta-crawler search engines in that separately weighted return lists must be combined in a sensible way. The crossword domain has the advantage of ready access to precise and abundant training data.

For a given clue, each expert module m returns a weighted set of candidates and a numerical level of confidence that the correct target is in this set. For each expert module m, the merger uses three real parameters: scale(m), length-scale(m) and spread(m). Each candidate is reweighted by raising its weight to the power spread(m), then normalizing the sum to 1. The confidence level is multiplied by the product of scale(m) and length-scale$(m)^{\text{targetlength}}$. To compute a combined probability distribution over candidates, the merger linearly combines the modified candidate sets of all the modules weighted by their modified confidence levels, and normalizes the sum to 1.

The scale, length-scale and spread parameters give the merger control over how the information returned by an expert module is incorporated into the final candidate list. Parameters are set using hill-climbing.

The objective function for optimization is the average log probability assigned to correct targets. This corresponds to maximizing the average log probability assigned by the solver to the correct puzzle fill-in, since in our model the probability of a puzzle solution is proportional to the product of the prior probabilities on the answers in each of the slots. The optimal value achieved on the 70 puzzle training set was $\log(\frac{1}{33.56})$.

Grid Filling

After realizing how much repetition occurs in crosswords, and therefore how well the CWDB covers the domain, one might wonder whether this coverage is enough to constrain solutions to such an extent that there is little left for the grid-filling algorithm to do. We did not find this to be the case. Simplistic grid filling yielded only mediocre results. As a measure of the task left to the grid-filling algorithm, on the first iteration of solving, using just the weighted candidate lists from the modules, only 40.9% of targets are in the top of the candidate list for their slot. However, the grid-filling algorithm is able to raise this to 89.4%.[4]

The algorithm employed by Proverb (Shazeer, Littman, & Keim 1999) models grid filling as an optimization problem: find the best way of choosing a candidate for each clue, while respecting the constraints of the grid. We can define "best" in several different ways; we attempted to maximize the expected overlap with the creator's solution. Other definitions of "best" include maximizing the probability of getting the entire puzzle correct, or maximizing expected letter overlap. The decision to use expected word overlap is motivated by the scoring system used in human tournaments (see below). Finding the optimal solution to this problem is a belief net inference problem; we use a type of "turbo decoding" (Shazeer, Littman, & Keim 1999) to approximate the solutions quickly.

Implicit Distribution Modules

Our probability measure assigns probability zero to a target that is suggested by no module and probability zero to all solutions containing that target. Therefore, we need to assign non-zero probability to all letter sequences. Clearly, there are too many to list explicitly (10^{21} for a 15-letter clue). We augmented the solver to reason with probability distributions over candidate lists that are implicitly represented. These *Implicit Distribution Modules* (Figure 2) generate additional candidates once the solver can give them more information about letter probability distributions over the slot.

The most important of these is a letter Bigram module, which "generates" all possible letter sequences of a given length by returning a letter bigram distribution over all possible strings, learned from the CWDB. The bigram probabilities are used throughout the solution process, so this module is integrated into the solver.

Note in Figure 3 there are some clues for which only Bigram returns the target. In a pretest run on 70 puzzles, the clue-target with the lowest probability was ≺Honolulu wear [14]: `hawaiianmuumuu`≻. This target never occurs in the CWDB, although both `muumuu` and `hawaiian` occur multiple times, and it gets a particularly low probability because of the many unlikely letter

[4] On average, over the 70 NYT puzzles in the test suite.

pairs in the target. Once the grid-filling process is underway, estimates of probability distributions for each letter in these longer targets are available, and this can limit the search for candidates.

To address long, multiword targets, we created free-standing implicit distribution modules. Each implicit distribution module takes a letter probability distribution for each letter of the slot (computed within the solver), and returns weighted candidate lists. These lists are then added to the previous candidate lists, and the grid-filling algorithm continues. This process of getting new candidates can happen several times during the solution process.

Tetragram suggests candidates based on a letter tetragram model, built from the WordList-Big. We hoped this would provide a better model for word boundaries than the bigram model mentioned above, since this list contains many multiword terms.

Segmenter calculates the $n = 10$ most probable word sequences with respect to both the letter probabilities and word probabilities from several sources using dynamic programming. The base word probabilities are unigram word probabilities from the CWDB. In addition, the Dijkstra module (described above) suggests the best 1000 words (with weights) given the current clue. These weights and the unigram probabilities are then combined for a new distribution of word probabilities.

For example, consider the clue ≺Tall footwear for rappers? [11]: `hiphopboots`≻. Given a letter distribution and a combined word distribution, the segmenter returned the following: `tiptopboots`, `hiphoproots`, `hiphopbooks`, `hiphoptoots`, `hiphopboots`, `hiphoproofs`, `riptaproots`, `hippopboots`, `hiptaproots`, `hiptapboots`. Note that the reweighting done by the Dijkstra module by examining the clue raises the probabilites of related words like `boots`.

Results

To evaluate Proverb's performance, we ran it on a large collection of daily puzzles, and on a set of recent tournament puzzles.

Daily Puzzles

We tested the system on puzzles from seven daily sources, listed in Table 1 (**Test**). The TV Guide puzzles go back to 1996, but the other sources were all from between August and December of 1998. We selected 70 puzzles, 10 from each source, as training puzzles for the system. The reweighting process described earlier was trained on the 5374 clues from these 70 puzzles. Additional debugging and modification of the modules was done after evaluation on these training puzzles.

Having fixed the modules and reweighting parameters, we then ran the system on the 370 puzzles in the final pool. The system acheived an average 95.3% words correct, 98.1% letters correct, and 46.2% puzzles completely correct (94.1%, 97.6%, and 37.6% without the implicit distribution modules). The NYT puzzles were the only ones that averaged under 95% words correct. Following up on our earlier observations, we split up the NYT puzzles and found that Proverb averaged 95.5% words correct on Monday through Wednesday puzzles and 85.0% words correct on Thursday through Sunday puzzles. As with people, the late-week NYT puzzles were more difficult for Proverb.

Tournament Puzzles

To better gauge the system's performance against humans, we tested Proverb using puzzles from the 1998 American Crossword Puzzle Tournament (ACPT) (Shortz 1990). The ACPT has been held annually for 20 years, and was attended in 1998 by 251 people. The scoring system for the ACPT requires that a time limit be set for each puzzle. A solver's score is then 10 times the number of words correct, plus a bonus of 150 if the puzzle is completely correct. In addition, the number of incorrect letters is subtracted from the full minutes early the solver finishes. If this number is positive, it is multiplied by 25 and added to the score.

There were seven puzzles in the offical contest, with time limits ranging from 15 to 45 minutes. We used the same version of Proverb described in the previous section. The results over the 1998 puzzles are shown in Table 4. The best human solvers at the competition finished all puzzles correctly, and the winner was determined by finishing time (the champion averaged under seven minutes per puzzle). Thus, while not competitive with the very best human solvers, Proverb would have placed 213 out of 251; its score on Puzzle 5 exceeded that of the median human solver at the contest.

The ACPT puzzles are very challenging, and include tricks like multiple letters or words written in a single grid cell, and targets written in the wrong slot. In spite of the fact that Proverb could not produce answers that bend the rules in this way, it still filled in 80% of the words correctly, on average. The implicit distribution modules ("Proverb(I)") helped improve the word score on these puzzles, but brought down the tournament score because it runs more slowly.

Conclusions

Solving crossword puzzles presents a unique artificial intelligence challenge, demanding from a competitive system broad world knowledge, powerful constraint satisfaction, and speed. Because of the widespread appeal, system designers have a large number of existing puzzles to use to test and tune their systems, and humans with whom to compare.

A successful crossword solver requires many artificial intelligence techniques; in our work, we used ideas from state-space search, probabilistic optimization, constraint satisfaction, information retrieval, machine learning and natural language processing. We

Name	Rank	Total	Avg Time
▷ Maximum	1	13140	0:59
TP (Champion)	1	12115	6:51
JJ (75%)	62	10025	-
MF (50%)	125	8575	-
MB (25%)	187	6985	-
▷ Proverb-I (24%)	190	6880	0:59
Proverb (15%)	213	6215	9:41
Proverb-I (15%)	215	6130	15:07

Table 4: Proverb compared favorably to the 251 elite human contestants at the 1998 championship. Lines preceded by a ▷ indicate the theoretical scores if the solver did every puzzle in under a minute.

found probability theory a potent practical tool for organizing the system and improving performance.

The level of success we acheived would probably not have been possible five years ago, as we depended on extremely fast computers with vast memory and disk storage, and used tremendous amounts of data in machine readable form. Perhaps the time is ripe to use these resources to attack other problems previously deemed too challenging for AI.

Acknowledgements

We received help and guidance from other members of the Duke Community: Michael Fulkerson, Mark Peot, Robert Duvall, Fred Horch, Siddhartha Chatterjee, Geoff Cohen, Steve Ruby, Alan Biermann, Donald Loveland, Gert Webelhuth, Michail Lagoudakis, Steve Majercik, Syam Gadde. Via e-mail, Will Shortz and William Tunstall-Pedoe made considerable contributions.

References

Ginsberg, M. L.; Frank, M.; Halpin, M. P.; and Torrance, M. C. 1990. Search lessons learned from crossword puzzles. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 210–215.

Landauer, T. K., and Dumais, S. T. 1997. A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction and representation of knowledge. *Psychological Review* 104(2):211–240.

Littman, M. L.; Keim, G. A.; and Shazeer, N. M. 1999. Solving crosswords with Proverb. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*.

Miller, G. R.; Beckwith, C.; Fellbaum, C.; Gross, D.; and Miller, K. 1990. Introduction to WordNet: An on-line lexical database. *International Journal of Lexicography* 3(4):235–244.

Salton, G., and McGill, M. J. 1983. *Introduction to Modern Information Retrieval*. McGraw-Hill.

Shazeer, N. M.; Littman, M. L.; and Keim, G. A. 1999. Solving crossword puzzles as probabilistic constraint satisfaction. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*.

Shortz, W., ed. 1990. *American Championship Crosswords*. Fawcett Columbine.