ELSEVIER

Artificial Intelligence 134 (2002) 23–55

Artificial Intelligence

www.elsevier.com/locate/artint

A probabilistic approach to solving crossword puzzles

Michael L. Littman [a,*], Greg A. Keim [b], Noam Shazeer [c]

[a] *AT&T Labs Research, Florham Park, NJ 07932-0971, USA*
[b] *Farfield Language Technologies, 165 South Main St., Harrisonburg, VA 22801, USA*
[c] *Google, 2400 Bayshore Parkway, Mountain View, CA 94043, USA*

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. After an analysis of a large collection of puzzles, we decided to use an open architecture in which independent programs 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 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).

Keywords: Crossword puzzles; Probabilistic reasoning; Information retrieval; Loopy belief propagation; Probabilistic constraint satisfaction; Posterior probability

1. Introduction

Crossword puzzles are attempted daily by millions of people, [1] and require of the solver both an extensive knowledge of language, history and popular culture, and the reasoning ability to execute a search over possible answers to find a set that fits in the grid. This dual

* Corresponding author.
E-mail addresses: mlittman@research.att.com (M.L. Littman), keim@rosettastone.com (G.A. Keim), noam@google.com (N. Shazeer).

[1] One common estimate is that a third of Americans solve a crossword puzzle regularly.

0004-3702/02/$ – see front matter
PII: S0004-3702(01)00114-X

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. While PROVERB's performance is well below that of human champions, it exceeds that of casual human solvers, averaging over 95% words correct per puzzle over a test set of 370 puzzles.

We first describe the problem and some of the insights we gained from studying a large database of crossword puzzles, as this motivated our design choices. We next discuss our underlying probabilistic model and the architecture of PROVERB, including how answers to clues are suggested by expert modules and how the system searches for an optimal fit of these possible answers into the grid. Finally, we present the system's performance on a large test suite of daily crossword puzzles, as well as on 1998 and 1999 tournament puzzles.

2. 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. Fig. 1 shows the solution to a puzzle published in the New York Times on October 10th, 1998. This will serve as a running example throughout the paper.

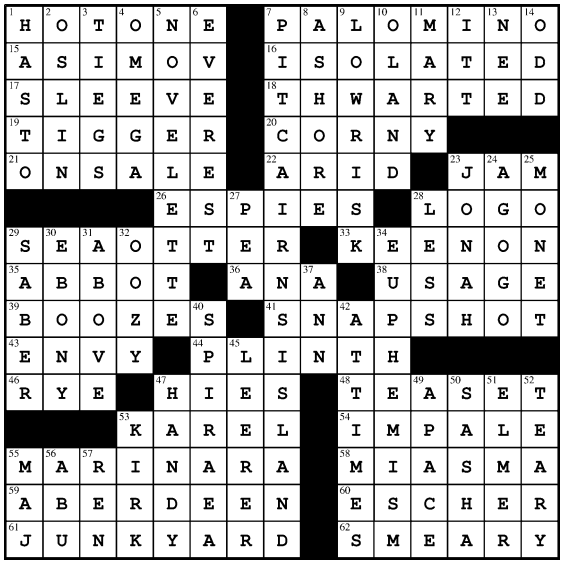


Fig. 1. An example crossword grid, published in the New York Times on October 10th, 1998, shows the breadth of targets.

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

		Puzzles	
Source	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

The clues cover a broad range of topics and difficulty levels. Some clue-target pairs are relatively direct: ≺ **44A** Statue base: `plinth`≻, [2] while others are more oblique and based on word play: ≺**8D** Taking liberty: `ashore`≻. Clues are between one and at most a dozen or so words long, averaging about 2.5 words per clue in a sample of clues we've collected. This is quite close to the 2.35 words per query average seen in web search engines [6,23].

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* will be 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. [3] By convention, all targets are at least 3 letters in length and long targets can be constructed by stringing multiple words together: ≺**7D** 1934 Hall and Nordhoff adventure novel: `pitcairnsisland`≻. Each empty square in the grid must be part of a down target and an across target (no "unchecked" squares).

As this is largely a new problem domain, distinct from the crossword-puzzle creation problem studied in search and constraint satisfaction [4,11], we wondered how hard crossword solving really was. To gain some insight into the problem, we studied a

[2] Targets appear in fixed-width font. When a clue number is given, the example is taken from the NYT October 10th, 1998 puzzle.

[3] Crossword Maestro is a commercial solver for British-style crosswords and is published by Genius 2000 Software (http://www.genius2000.com/cm.html). 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). For comparison, an example cryptic clue is ≺Split up rotten platforms: `pulpits`≻, because "rotten" suggests creating an anagram of "split up" that means "platforms".

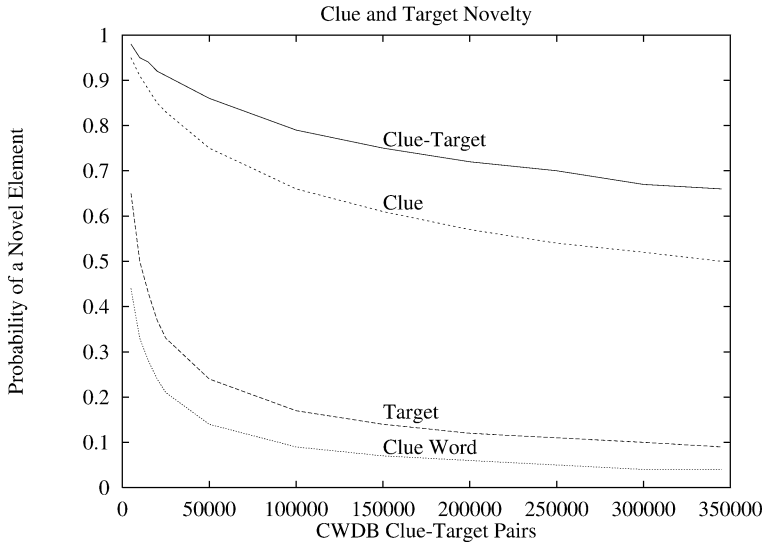


Fig. 2. Clue and target novelty decreases with the size of the CWDB. Given all 350,000 clues, we would expect a new puzzle to contain about a third previously seen clue–target pairs.

large corpus of existing puzzles. We collected 5582 crossword puzzles from a variety of sources, summarized in Table 1: online versions of daily print newspaper puzzles (The New York Times, The Los Angeles Times, USA Today, TV Guide), online sites featuring puzzles (Dell, Riddler), syndicates specifically producing for the online medium (Creator's Syndicate, CrosSynergy Syndicate). A subset of puzzles constitute a crossword database (CWDB) of around 350,000 clue–target pairs, with over 250,000 of them unique, which served as a potent knowledge source for this project.

2.1. 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: `erie`≻ to be a common clue–target pair in many puzzles.[4] Our 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 existing database—how novel are crossword puzzles? In Fig. 2, we graph the probability of novel targets, clues, clue–target pairs, and clue words as we increase the number of elements in the database.

[4] The five most common targets in the CWDB are `era`, `ore`, `area`, `erie` and `ale`. The target `erie` appears in over 7% of puzzles. The five most common clues are "Exist", "Greek letter", "Jai `____`", "Otherwise", and "Region". The five most common clue–target pairs are ≺Exist: `are`≻, ≺Jai `____`: `alai`≻, ≺Otherwise: `else`≻, ≺Region: `area`≻, and ≺Anger: `ire`≻.

After randomizing, we looked at subsets of the database ranging from 5,000 clues to almost 350,000. For each subset, we calculated the percentage of the particular 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 database (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 clearly contains a tremendous amount of useful domain-specific information.

2.2. The New York Times crossword puzzle

The New York Times (NYT) crossword is considered by many to be the premiere daily puzzle. Will Shortz, the NYT's crossword puzzle editor, attempts 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 (≺**28D** Nothing `____`: `less`≻) get less and less common as the week goes on. In addition, clues with the phrase "in a way" or a trailing question mark (≺**4D** The end of Plato?: `omega`≻), which are often a sign of a themed or pun clue, get more common. Single word clues decrease. 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 it is a bit of an outlier on some of these scales, partly due to the fact that the puzzles are larger (up to 23×23, as opposed to 15×15 for the other days).

Table 2
Percentages of various clue and target types vary by day of week in NYT puzzles and show a trend of increasing difficulty from Monday to Saturday

	Mon	Tue	Wed	Thu	Fri	Sat
#puzzles	89	92	90	91	91	87
#clues	77.3	77.2	76.7	74.7	70.0	70.2
3-letter target	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
Single Word	15.6	14.9	16.0	17.2	16.9	20.6
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

2.3. Categories of clues

In the common syntactic categories shown in Table 2, clue structure leads to simple ways to answer those clues. For example, given a fill-in-the-blank clue, we might write a program to scan through text sources looking for all phrases that match on word boundaries and known letters. If we encounter a clue such as ≺**55D** Key abbr.: `maj`≻, we might want to return a list of likely abbreviations.

In addition, a number of non-syntactic, *expert* categories stand out, such as synonyms (≺**40D** Meadowsweet: `spiraea`≻), kind-of (≺**27D** Kind of coal or coat: `pea`≻, since "pea coat" and "pea coal" are both standard phrases), movies (≺**50D** Princess in Woolf's "Orlando": `sasha`≻), geography (≺**59A** North Sea port: `aberdeen`≻), music (≺**2D** "Hold Me" country Grammy winner, 1988: `oslin`≻) and literature (≺**53A** Playwright/novelist Capek: `karel`≻). There are also clues that do not fit a simple pattern, but might be solved using information retrieval techniques (≺**6D** Mountain known locally as Chomolungma: `everest`≻). Given the many different sources of information that can be brought to bear to solve different types of clues, this suggests a two-stage architecture for our solver: one consisting of a collection of special-purpose and general 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 from a Duke seminar class wrote code for the project) to build 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.

3. Architecture

Fig. 3 illustrates the components of PROVERB. Given a puzzle in the format suggested in Fig. 4, the *coordinator* separates the clues from the grid and sends a copy of the clue

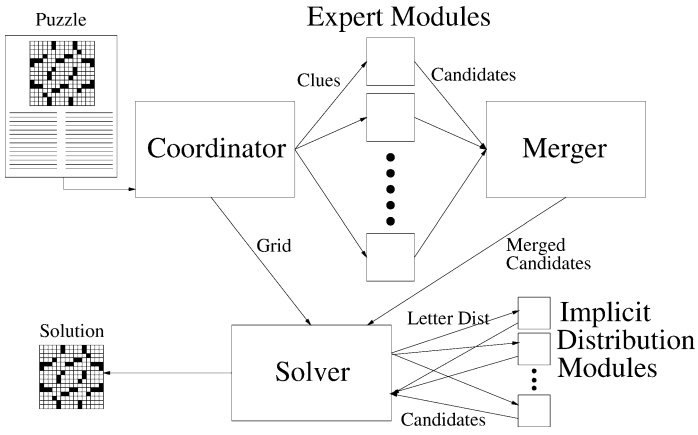


Fig. 3. PROVERB consists of a set of independent communicating programs written in Java, C, C++, and Perl.

```
1998 The New York Times  W Shortz NY Times Thu Oct 10 1998
ACROSS
1 hotone Knee-slapper
7 palomino Trigger, for one
...
61 junkyard Rusty locale
62 smeary Not kissproof
DOWN
1 hasto Can't help but
2 oslin "Hold Me" country Grammy winner, 1988
...
56 abu "The Thief of Baghdad" role
57 ren Cartoon character who says "You eediot!"
GRID
6 -1 8
6 -1 8
...
8 -1 6
8 -1 6
```

Fig. 4. Puzzles in our CWDB format include across and down clues with their answers, as well as a run-length encoding of the puzzle grid.

list (with target lengths) to each *expert module*. The expert modules generate probability-weighted candidate lists, in isolation from the grid constraints. Each expert module receives a copy of all clues and is free to return no candidates for any clues, 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 that also satisfies the grid constraints.

The *implicit distribution modules* are used by the solver, and are described in Section 3.3.

3.1. The probabilistic model

To create a unified semantics for the candidate-generation modules, we found it useful to specify a generative model for crossword-puzzle creation. First, assume that crossword puzzles are created by starting with an empty grid and repeatedly choosing words for the slots according to a particular creator's distribution (ignore clues and crossing constraints at first). After choosing the words, if the crossing constraints are satisfied, then the creator keeps the puzzle and generates a set of clues. Otherwise, the creator tosses the entire puzzle and 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. We will discuss this optimization more in Section 4, 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, and how we can combine them into unified distributions.

3.2. Candidate-list 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": `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, while giving 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 in Section 3.4 attempts to account for the error in these estimates by examining the performance of expert modules on some held-out tuning 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 can "trust" the module's predictions.

In all, the developers 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 then 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 single 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.

Table 3
Performance of the expert modules on 70 puzzles (5374 clues) shows differences between modules in the fraction of clues attempted (**Guess**), their accuracy (**Acc**), number of targets returned (**Len**) and contribution to the overall lists (**Best**). Also measured but not shown are the implicit modules

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

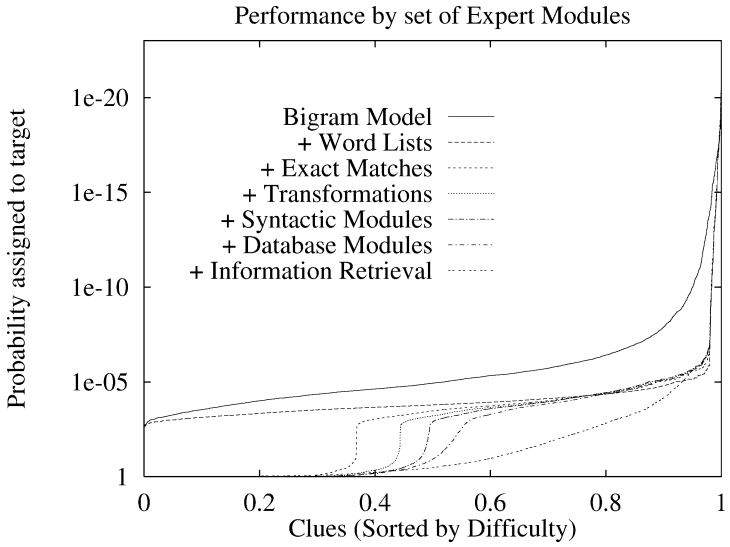


Fig. 5. The cumulative probability assigned as module groups are added shows that different types of modules make different contributions. Each line is sorted independently.

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 each clue's target 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 Fig. 5, the top line represents the probability assigned by the Bigram module (described in Section 3.3). This probability is low for all targets, but very low for the hard targets (right side of graph). 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 (closer to 1.0) to the target. Note the large increase due to the Exact Match module. Finally, notice that there is a small segment of hard targets that the system performs very poorly on—the targets that no module other than Bigram returns. Section 3.3 introduces extensions to the system designed to help with this class of clues.

We proceed with a description of the expert modules in PROVERB. Once again, examples with clue numbers are taken from the October 10th, 1998 puzzle mentioned earlier.

3.2.1. Word list modules

WordList, WordList-Big. These modules ignore their clues and return all words of the correct length from several dictionaries. WordList contains a list of 655,000 terms from a wide variety of 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: first and last names, adjacent words from clues in the CWDB, etc. WordList-Big contains over 2.1 million terms. All terms are weighted equally. These modules work as crucial "backstops" for indirect clues such as ≺**5D** 10,000 words, perhaps: `novelette`≻.

WordList-CWDB. 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. This involves examining their frequency in crossword puzzles and normalizing to account for the bias caused by letters intersecting across and down terms.

3.2.2. CWDB-specific modules

Exact match. This module 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. This module returns the best correct answer 35.9% of the time. It returns nothing on an unseen clue and can return an incorrect answer on an ambiguous clue (for example, it returns `eeyore` for ≺**19A** Pal of Pooh: `tigger`≻).

Transformations. This module learns in advance a set of textual transformations that, when applied to clue–target pairs in the CWDB, generates other clue–target pairs in the database. When faced with a new clue, it executes all applicable transformations and returns the results, weighted based on the previous precision/recall of these transformations. Transformations in the database include single-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

A concrete example is ≺**51D** Bugs chaser: `elmer`≻, which is solved by the combination of the entry ≺Bugs pursuer: `elmer`≻ in the CWDB and the transformation rule "X pursuer $\leftrightarrow$ X chaser".

The CWDB-specific modules have been reimplemented and are available for interactive exploration on the web (http://www.oneacross.com/).

3.2.3. Information retrieval modules

Crossword clues present an interesting challenge to 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 existing IR techniques over several document collections. Each is sketched for completeness below.

Encyclopedia. This module is based on an indexed set of encyclopedia articles. For each query term, we compute 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 of the correct target length 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 sum to one. 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. Extremely common terms in the query such as "as" and "and" are ignored.

Partial match. Consider the standard vector space model [17], 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, we find all clues in the CWDB that share words with c. For each such clue, we give its target a weight based on the dot product of the clue with c. The assigned weight is geometrically interpolated between $1/\text{size(dictionary)}$ and 1 based on this dot product. This module has low precision compared to Exact match and Transformations, but it has significantly higher recall. The clue ≺**58A** Bad atmosphere: `miasma`≻ is solved because of its close relation to ≺Poisonous atmosphere: `miasma`≻.

LSI-Ency, LSI-CWDB. Latent semantic indexing (LSI) [2] 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 problem of synonym selection on a standardized test [8], which is closely related to solving crossword clues. Our LSI modules were trained on CWDB (all clues with the same target were treated as a document) and separately on an online encyclopedia and returned the closest words (by cosine) for each clue. Normalized cosine scores were used as confidence scores.

Dijkstra modules. The Dijkstra modules were inspired by the intuition that related words either co-occur with one another or co-occur with similar words. This suggests a measure of relatedness based on graph distance. From a selected set of text, the module builds a weighted directed graph on the set of all terms. For each database d and each pair of terms (t, u) that co-occur in the same document, the module places an edge from t to u in the graph 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, a term u receives a score of

$$-\log(\text{fraction of documents containing } t) - \text{weight}(\text{minimum weight path } t \to u).$$

We find the highest scoring terms with a shortest-path-like search. For a multi-word clue, we break the clue into individual terms and add the scores as computed above.

For databases, we used an encyclopedia index, two thesauri, a database of wordforms and the CWDB. The clue ≺**7A** Trigger, for one: `palomino`≻ was solved by noting the path from `trigger` to `palominos` to `palomino`.

The four Dijkstra modules in our system use variants of this technique: Dijkstra1 is the basic module, Dijkstra2 uses this approach to score clues in the CWDB and returns their targets, Dijkstra3 uses only the CWDB to form the edges of the graph, and Dijkstra4 is a combination of Dijkstra2 and Dijkstra3.

3.2.4. Database modules

Movie. The Internet Movie Database (www.imdb.com) is an online resource with a wealth of information about all manner of movies and TV shows. This module looks for a number of patterns in the clue (for example, quoted titles as in ≺**56D** "The Thief of Baghdad" role: `abu`≻ or Boolean operations on names as in ≺Cary or Lee: `grant`≻), and formulates queries to a local copy of the database. This resource is amazingly complete, and one of our design questions was whether we should artificially limit its coverage to avoid spurious matches (we chose not to).

Music, literary, geography. These modules use simple pattern matching of the clue (looking for keywords "city", "author", "band" and others as in ≺**15A** "Foundation Trilogy" author: `asimov`≻) 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. There are four distinct synonym modules, based on three different thesauri. Using the WordNet [13] database, one module looks for root forms of words in the clue, and then finds a variety of related words (for example, ≺**49D** Chop-chop: `apace`≻). 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 converted back to the form of the original clue word (number, tense, etc.), for example ≺**18A** Stymied: `thwarted`≻.

3.2.5. Syntactic modules

Fill-in-the-blanks. Over five percent of all clues in CWDB have a blank in them. We searched a variety of databases to find clue patterns with a missing word (music, geography, movies, literary and quotes). For example, given ≺**36A** Yerby's "A Rose for ___ Maria": `ana`≻, these modules would search for the pattern `for ... maria`, allowing any three characters to fill the blanks, including multiple words. In some of our pretests we also ran these searches over more general sources of text like encyclopedias and archived news feeds, but for efficiency, we left these out of the final experimental runs.

KindOf. "Kind of" 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, "type of" (≺A type of jacket: `nehru`≻), "starter for" (≺Starter for saxon: `anglo`≻), "suffix with" (≺Suffix with switch or sock: `eroo`≻), and "middle" (≺Cogito sum middle: `ergo`≻). The principle differences between the KindOf and Fill-in-the-blanks modules are the data sources searched (short phrases and compound words vs. proper nouns and titles) and the handling of the Boolean operators "and" and "or".

3.3. Implicit distribution modules

When a target is not included in any of our databases (a rare word or a novel sequence of words), it is still important that it be assigned a probability above that of random letter sequences—neither `schaeffer` nor `srhffeeca` appears as a target in the CWDB, but the former is a much more reasonable candidate. We augmented the solver to reason with probability distributions over candidate lists that are implicitly represented. These *implicit distribution modules* 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 the given length by returning a letter bigram distribution over all possible strings, learned from the CWDB. Because the bigram probabilities are used throughout the solution process, this module is actually tightly integrated into the solver itself.

Note in Fig. 5 that there are some clues for which no module except Bigram is returning the target. In a pretest run on 70 puzzles, the single clue-target with the lowest probability was ≺Honolulu wear: `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 pairs in the target (even so, its probability is still much higher than that of a random sequence of 14 letters). Once the grid-filling process is underway, we have probability distributions for each letter in these longer targets and this can limit our search for candidates.

To address longer, multiword targets, we created free-standing implicit distribution modules. During grid filling, 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. The tetragram module 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. Note that there are plenty of training examples of each tetragram in this collection (at least 400 for each of the $26^4 = 450$ thousand sequences).

Segmenter. The segmenter calculates the ten 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?: `hiphopboots`≻. This is unlikely to be in any word list, or even in a huge text collection. Given a letter distribution from the crossing words and a combined word distribution, the segmenter returned the following top ten at solving time: `tiptopboots`, `hiphoproots`, `hiphopbooks`,

`hiphoptoots`, `hiphopboots`, `hiphoproofs`, `riptaproots`, `hippopboots`, `hiptaproots`, and `hiptapboots`. Note that the reweighting done by the Dijkstra module by examining the clue raises the probabilities of related words like `boots`. These novel candidates are then fed back into the solver, and the optimization continues.

In the results described in Section 5, we ran PROVERB with and without the implicit distribution modules to assess their contribution to the solver.

3.4. Merging candidate lists

After each expert module has generated a weighted candidate list, each clue's lists must somehow be merged into a unified candidate list with a common weighting scheme to be presented to the solver. This problem is similar to the problem faced by meta-crawler search engines in that separately weighted return lists must be combined in a sensible way. We exploited the existence of precise and abundant training data to attack this problem in the crossword domain.

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. In the *merger*, each expert module m has three real-valued parameters: scale(m), length-scale(m) and spread(m). For each clue, the merger reweights the candidate set by raising each weight to the power spread(m), then normalizing their sum to 1. It multiplies the confidence level by the product of scale(m) and length-scale(m)$^{\text{target length}}$. 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. We decided on this set of "knobs" through some trial and error, noting that some of the expert modules produce probabilities that were in too narrow a range, and others varied in accuracy considerably depending on the length of targets. The values for the parameters were set using a naive hill-climbing technique, described next.

The objective function for optimization is the average log probability assigned to the correct target. 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 the hill-climber achieved on the 70 puzzle training set was $\log(1/33.56)$.

4. Grid filling

After realizing how much repetition occurs in crosswords, in both targets and clues, and therefore how well the CWDB covers the domain, one might wonder whether this coverage is enough to constrain the domain to such an extent that there is not much 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, 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 described in this section is able to raise this to 89.4%. [5]

Constraint satisfaction is a powerful and general formalism. Crossword puzzles are frequently used as examples of constraint satisfaction problems (CSPs) [10], and search can be used to great effect in crossword-puzzle creation [4]. We developed a probabilistic extension to CSPs in which the confidence scores produced by candidate generation are used to induce a probability distribution over solutions. We studied two optimization problems for this model. The *maximum probability solution* corresponds to maximizing the probability of a correct solution, while the *maximum expected overlap solution* corresponds to maximizing the number of correct variable values in the solution. The former can be solved using standard constrained-optimization techniques. The latter is closely related to belief network inference, and we applied an efficient iterative approximation equivalent to Pearl's belief propagation algorithm [15] on a multiply connected network. It is also closely related to the turbo decoding algorithm [12].

In this section, we develop the underlying mathematics behind the probabilistic CSP model and describe how the two optimization problems and the approximation result in different solutions on a collection of artificial puzzles. We present the model in a general way because we think it has applicability beyond the crossword puzzle domain.

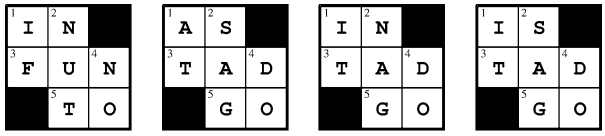
4.1. Constraint satisfaction problems

We define a (Boolean) constraint satisfaction problem [11], or CSP, as a set of variables and constraints on the values of these variables. For example, consider the crossword puzzle in Fig. 6. Here, variables, or slots, are the places words can be written. The binary constraints on variable instantiations are that across and down words mesh. The domain of a variable, listed beneath the puzzles, is the set of values the variable can take on; for example, variable 3A (3 across) can take on values `FUN` or `TAD`. A *solution* to a CSP is an instantiation (assignment of values to the variables) such that each variable is assigned a value in its domain and no constraint is violated. The crossword CSP in Fig. 6 has four solutions, which are labeled **A** through **D** in the figure. (The probability values in the figure will be explained next.)

Although CSPs can be applied to many real-world problems, some problems do not fit naturally into this framework. Of course, the example we considered is the problem of solving a crossword puzzle from its clues. The slots of the puzzle are nicely captured by CSP variables, and the grid by CSP constraints, but how do we transform the clues into domain values for the variables? A natural approach is to take a clue like ≺Small amount: `tad`≻ and generate a small set of candidate answers of the appropriate length to be the domain: `TAD`, `JOT`, `DAB`, `BIT` are all examples from our CWDB.

This approach has several shortcomings. First, because of the flexibility of natural language, almost any word can be the answer to almost any clue; limiting domains to small sets will likely exclude critical candidates. Second, even with a direct clue, imperfections in the candidate generation process may cause a reasonable candidate to be excluded.

[5] These scores are averages based on the 70 NYT puzzles in the test suite.



	A	B	C	D
P	0.350	0.250	0.267	0.133
Q	2.367	2.833	3.233	2.866
Q^{∞}	2.214	2.793	3.529	3.074

slot 1A

v	p	q	$q^{(\infty)}$
AS	.5	.250	.190
IN	.3	.617	.645
IS	.2	.133	.165

slot 1D

v	p	q	$q^{(\infty)}$
IT	.4	.400	.496
IF	.3	.350	.314
AT	.3	.250	.190

slot 3A

v	p	q	$q^{(\infty)}$
FUN	.7	.350	.314
TAD	.3	.650	.686

slot 2D

v	p	q	$q^{(\infty)}$
NAG	.4	.267	.331
SAG	.3	.383	.355
NUT	.3	.350	.314

slot 5A

v	p	q	$q^{(\infty)}$
GO	.7	.650	.686
TO	.3	.350	.314

slot 4D

v	p	q	$q^{(\infty)}$
NO	.7	.350	.314
DO	.3	.650	.686

Fig. 6. This crossword puzzle with probabilistic preferences (p) on the candidate words (v) has four possible solutions, varying in probability (P) and expected overlap (Q). Posteriors (q) and their approximations ($q^{(\infty)}$) are described in the text.

To avoid these difficulties, we might be tempted to over-generate our candidate lists. Of course, this has the new shortcoming that spurious solutions will result.

This is a familiar problem in the design of grammars for natural language parsing: "Either the grammar assigns too many structures ... or it incorrectly predicts that examples ... have no well-formed structure" [1]. A solution in the natural language domain is to annotate grammar rules with probabilities, so that uncommon rules can be included (for coverage) but marked as less desirable than more common rules (for correctness). Then, no grammatical structure is deemed impossible, but better structures are assigned higher probability.

Following this line of thought for the crossword puzzle CSP, we annotate the domain of each variable with preferences in the form of probabilities. This gives a solver a way to distinguish better and worse solutions to the CSP with respect to goodness of fit to the clues.

Formally, we begin with a CSP specified as a set of n variables $X = \{x_1, \ldots, x_n\}$ with domain D_i for each $x_i \in X$. The variables are coupled through a constraint relation match, defined on pairs of variables and values: if x_i, x_j are variables and v, w are values, the proposition $\mathsf{match}_{x_i,x_j}(v, w)$ is true if and only if the partial instantiation $\{x_i = v, x_j = w\}$ does not violate any constraints. The match relation can be represented as a set of constraint

tables, one for each pair of variables in X. The variables, values, and constraints are jointly called a *constraint network*. We then add preference information to the constraint network in the form of probability distributions over domains: $p_{x_i}(v)$ is the probability that we take $v \in D_i$ to be the value of variable x_i. Since p_{x_i} is a probability distribution, we insist that for all $1 \leqslant i \leqslant n$, $\sum_{v \in D_i} p_{x_i}(v) = 1$ and for all $v \in D_i$, $p_{x_i}(v) \geqslant 0$. This is a special case of probabilistic CSPs [18]. An opportunity for future work is to extend the algorithms described here to general probabilistic CSPs.

In PROVERB, probabilities are produced as the output of the merger. Extending the running example, we can annotate the domain of each variable with probabilities, as shown in Fig. 6 in the columns marked "p". (We have no idea what clues would produce these candidate lists and probabilities; they are intended for illustration only.) For example, the figure lists $p_{2\mathrm{D}}(\mathsf{NUT}) = 0.3$.

We next need to describe how preferences on values can be used to induce preferences over complete solutions. We consider the following probability model, mentioned in Section 3.1. Imagine that solutions are "generated" by independently selecting a value for each variable according to its probability distribution p, then, if the resulting instantiation satisfies all constraints, we "keep" it, otherwise we discard it and draw again. This induces a probability distribution over solutions to the CSP in which the probability of a solution is proportional to the product of the probabilities of each of the values of the variables in the solution. The resulting solution probabilities for our example CSP are given in Fig. 6 in the row marked P.

The solution probabilities come from taking the product of the value probabilities and then normalizing by the total probability assigned to all valid solutions (Pr(match)). For example, the probability assigned to solution **C** is computed as:

$$\begin{aligned} P(\mathbf{C}) &= p_{1\mathrm{A}}(\mathsf{IN}) \cdot p_{3\mathrm{A}}(\mathsf{TAD}) \cdot p_{5\mathrm{A}}(\mathsf{GO}) \cdot p_{1\mathrm{D}}(\mathsf{IT}) \\ &\quad \cdot p_{2\mathrm{D}}(\mathsf{NAG}) \cdot p_{4\mathrm{D}}(\mathsf{DO}) / \Pr(\text{match}) \\ &= (0.3)(0.3)(0.7)(0.4)(0.4)(0.3) / \Pr(\text{match}) \\ &= 0.00302/0.01134 = 0.26667. \end{aligned}$$

In the next section, we discuss how these values can be used to guide the selection of a solution.

4.2. Optimization problems

We can use the probability distribution over solutions, as defined above, to select a "best" solution to the CSP. There are many possible notions of a best solution, each with its own optimization algorithms. We focused on two optimization problems: maximum probability solution and maximum expected overlap solution.

4.2.1. Maximum probability

The *maximum probability* solution is an instantiation of the CSP that satisfies the constraints and has the largest probability of all such instantiations (solution **A** with $P(\mathbf{A}) = 0.350$ from Fig. 6). It can be found by computing

$$\operatorname*{argmax}_{\text{soln: } v_1,\ldots,v_n} P(v_1,\ldots,v_n) = \operatorname*{argmax}_{\text{soln: } v_1,\ldots,v_n} \prod_{i=1}^{n} p_{x_i}(v_i)/\Pr(\text{match})$$

$$= \operatorname*{argmax}_{\text{soln: } v_1,\ldots,v_n} \prod_{i=1}^{n} p_{x_i}(v_i). \tag{1}$$

That is, we just need to search for the solution that maximizes the product of the preferences p. This is an NP-complete problem [3], but it can be attacked by any of a number of standard search procedures: A*, branch and bound, integer linear programming, weighted Boolean satisfiability, etc.

Another way of viewing the maximum probability solution is as follows. Imagine we are playing a game against Nature. Nature selects a solution at random according to the probability distribution described in Section 4.1 and keeps its selection hidden. We must now propose a solution for ourselves. If our solution matches the one selected by Nature, we win one dollar. If not, we win nothing. If we want to select the solution that maximizes our expected winnings (the probability of being completely correct), then clearly the maximum probability solution is the best choice.

4.2.2. Maximum expected overlap

The *maximum expected overlap* solution is a more complicated solution concept and is specific to our probabilistic interpretation of preferences. It is motivated by the crossword puzzle scoring procedure used in the yearly human championship known as the American Crossword Puzzle Tournament (Section 5.3). The idea is that we can receive partial credit for a proposed solution to a crossword puzzle by counting the number of words it has in common with the true solution.

In a probabilistic setting, we can view this problem as another game against Nature. Once again, Nature selects a solution at random weighted by the P distribution and we propose a solution for ourselves. For every word (variable-value pair) in common between the two solutions (i.e., the overlap), we win one dollar. Again, we wish to select the solution that maximizes our expected winnings (the number of correct words).

In practice, the maximum expected overlap solution is often highly correlated with the maximum probability solution. However, they are not always the same. The expected overlap Q for each the four solutions in Fig. 6 is listed in the table; the maximum expected overlap solution is **C**, with $Q(\mathbf{C}) = 3.233$ whereas the maximum probability solution is **A**. Thus, if we choose **A** as our solution, we'd expect to have 2.367 out of six words correct, whereas solution **C** scores almost a full word higher, on average.

To compute the expected overlap, we use a new set of probabilities: $q_x(v)$ is the probability that variable x has value v in a solution. It is defined as the sum of the probabilities of all solutions that assign v to x. Whereas $p_x(v)$ is a prior probability on setting variable x to value v, $q_x(v)$ is a posterior probability. Note that for some slots, like 3A, the prior p and posterior q of the values differ substantially.

As a concrete example of where the q values come from, consider

$$q_{2\mathrm{D}}(\mathsf{SAG}) = \Pr(\mathbf{B}) + \Pr(\mathbf{D}) = 0.250 + 0.133 = 0.383.$$

For the expected overlap Q, we have

$$\begin{aligned} Q(\mathbf{D}) &= q_{1\mathrm{A}}(\mathtt{IS}) + q_{3\mathrm{A}}(\mathtt{TAD}) + q_{5\mathrm{A}}(\mathtt{GO}) + q_{1\mathrm{D}}(\mathtt{IT}) + q_{2\mathrm{D}}(\mathtt{SAG}) + q_{4\mathrm{D}}(\mathtt{DO}) \\ &= 0.133 + 0.650 + 0.650 + 0.400 + 0.383 + 0.650 = 2.867. \end{aligned}$$

By the linearity of expectation,

$$\underset{\text{soln: } v_1,\ldots,v_n}{\operatorname{argmax}} Q(v_1,\ldots,v_n) = \underset{\text{soln: } v_1,\ldots,v_n}{\operatorname{argmax}} \sum_{i=1}^{n} q_{x_i}(v_i), \tag{2}$$

thus, computing the maximum expected overlap solution is a matter of finding the solution that maximizes the sum of a set of weights, q. The weights are very hard to compute in the worst case because they involve a sum over all solutions. The complexity is #P-complete, like belief network inference [16].

In the next section, we develop a procedure for efficiently approximating q. We will then give results on the use of the resulting approximations for solving artificial and real crossword puzzles.

4.3. Estimating the posterior probabilities

Our combination of constraint satisfaction problems with probabilistic preferences has elements in common with both constraint networks and belief networks [15]. Although computing posterior probabilities in general CSPs with probabilistic preferences is intractable, when the constraint relations form a tree (no loops), computing posterior probabilities is easy.

Given a constraint network N with or without cycles, a variable x with domain D, and value $v \in D$, we want to approximate the posterior probability $q_x(v)$ that variable x gets value v in a complete solution. We develop a series of approximations of N around x, described next.

Let the "unwrapped network" $U_x^{(d)}$ be the breadth-first search tree of depth d around x where revisitation of variables is allowed, but immediate backtracking is not. For example, Fig. 7(a) gives the constraint network form of the crossword puzzle from Fig. 6. Figs. 7(b)–(f) give a sequence of breadth-first search trees $U_{3\mathrm{A}}^{(d)}$ of differing depths around 3A. The graph $U_x^{(d)}$ is acyclic for all d. The limiting case $U_x^{(\infty)}$, is a possibly infinite acyclic network locally similar to N in the sense that the labels on neighbors in the infinite tree match those in the cyclic network. This construction parallels the notion of a universal covering space from topology theory [14].

We consider $U_x^{(d)}$ as a constraint network. We give each variable an independent prior distribution equal to that of the variable in N with the same label.

Let $q_x^{(d)}(v)$ be the posterior probability that x takes value v in the network $U_x^{(d)}$. As d increases, we'd expect $q_x^{(d)}(v)$ to become a better estimate of $q_x(v)$ since the structure of $U^{(d)}$ becomes more similar to N. (In fact, there is no guarantee this will be the case, but it is true in the examples we've studied.)

Computing the posterior probabilities on unwrapped networks has been shown equivalent to Pearl's belief propagation algorithm [24], which is exact on singly connected networks but only approximate on loopy ones [15].

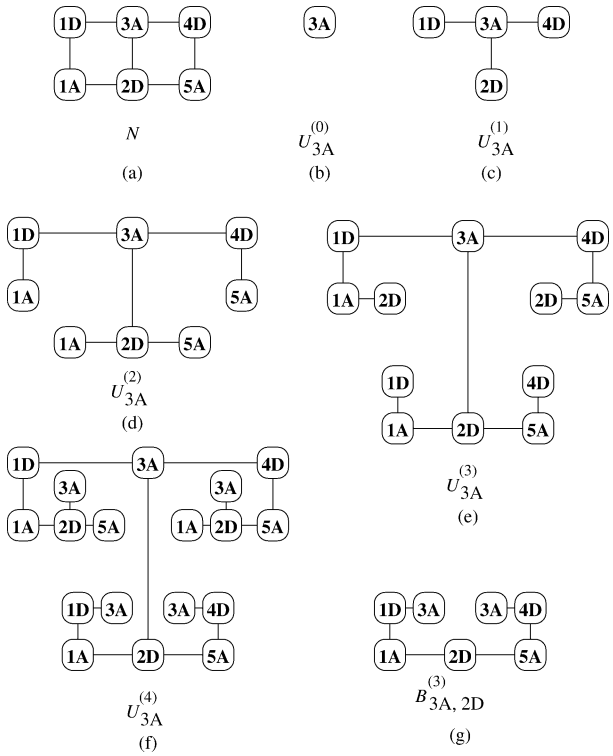


Fig. 7. A cyclic constraint network (a) can be approximated by tractable tree-structured constraint networks (b)–(f). These networks have a fractal structure and can be manipulated efficiently by exploiting shared subnetworks (g).

We will now derive efficient iterative equations for $q_x^{(d)}(v)$. Consider a variable x with neighbors $y_1, \ldots, y_m$. We define $B^{(d)}_{x,y_i}$ as the y_i-branch of $U_x^{(d+1)}$, or equivalently, $U_{y_i}^{(d)}$ with the x-branch removed (see Fig. 7(g)). Let $b^{(d)}_{x,y_i}(w)$ be the posterior probability that y_i takes value w in the network $B^{(d)}_{x,y_i}$. Note that $U_x^{(0)}$ and $B^{(0)}_{x,y_i}$ contain the single variables x and y_i respectively. Thus,

$$q_x^{(0)}(v) = p_x(v) \quad \text{and} \quad b^{(0)}_{x,y_i}(w) = p_{y_i}(w).$$

For positive d, we view $U_x^{(d)}$ as a tree with root x and branches $B^{(d-1)}_{x,y_i}$. According to our model, a solution on $U_x^{(d)}$ is generated by independently instantiating all variables according to their priors and discarding the solution if constraints are violated. This is equivalent to first instantiating all of the branches and checking for violations, then instantiating x and checking for violations. Furthermore, since the branches are disjoint, they can each be instantiated separately. After instantiating and checking the branches, the neighbors y_1 through y_m are independent and y_i has probability distribution $b^{(d-1)}_{x,y_i}$. The posterior probability $q_x^{(d)}(v)$ that x takes the value v is then proportional to the probability

$p_x(v)$ that v is chosen multiplied by the probability that $x = v$ does not violate a constraint between x and one of its neighbors. We get

$$q_x^{(d)}(v) = k_x^{(d)} p_x(v) \cdot \prod_{i=1}^{m} \sum_{w|\mathsf{match}_{y_i,x}(w,v)} b_{x,y_i}^{(d-1)}(w),$$

where $k_x^{(d)}$ is the normalization constant necessary to make the probabilities sum to one. Since $B_{y_i,x}^{(d)}$ is simply $U_x^{(d)}$ with one branch removed,[6] the equation for $b_{y_i,x}^{(d)}(v)$ is very similar to the one for $q_x^{(d)}(v)$:

$$b_{y_i,x}^{(d)}(v) = k_{y_i,x}^{(d)} p_x(v) \cdot \prod_{j=1\ldots m, j\neq i} \sum_{w|\mathsf{match}_{y_j,x}(w,v)} b_{x,y_j}^{(d-1)}(w).$$

Note that, as long as the constraint network N is 2-consistent, the candidate lists are non-empty and the normalization factors are non-zero.

The sequence $\{q_x^{(d)}(v)\}$ does not always converge. However, it converged in all of our artificial experiments. If it converges, we call its limit $q_x^{(\infty)}(v)$.

In the case in which N is a tree of maximum depth k, $U_x^{(d)} = U_x^{(\infty)} = N$ for all $d \geqslant k$. Thus, $q_x^{(\infty)}(v) = q_x(v)$, the true posterior probability. However, in the general case in which N contains cycles, $U_x^{(\infty)}$ is infinitely large. We hope that its local similarity to N makes $q_x^{(\infty)}(v)$ a good estimator of $q_x(v)$.

The running time of the calculation of $q^{(d)}$ is polynomial. If there are n variables, each of which is constrained by at most μ other variables, and the maximum size of any of the constraint tables is s, then $\{b^{(d)}\}$ and $\{q^{(d)}\}$ can be computed from $b^{(d-1)}$ in $O(n\mu^2 s)$ time. In PROVERB, the candidate lists are very large, so s is enormous. To reduce the value of s, we inserted an extra variable for each square of the puzzle. These letter variables can only take on twenty-six values and are assigned equal prior probabilities. Each of the constraints in the revised network relates a letter variable and a word variable. Thus, s is only linear in the length of the candidate lists, instead of quadratic.

4.4. Artificial puzzles

To explore how the expected overlap and solution probability relate, and how the iterative estimate compares to these, we randomly generated 100 puzzles for each of the six possible 5×5 crossword grids,[7] as shown in Fig. 8. Candidates were random binary strings. Each slot was assigned a random 50% of the possible strings of the right length. The prior probabilities were picked uniformly at random from the interval $[0, 1]$, then normalized to sum to 1. We discarded puzzles with no solution; this only happened twice, both times on Grid F.

For each puzzle, we computed the complete set of solutions and their probabilities (average numbers of solutions are shown in Table 5), from which we derived the exact

[6] We reversed subscripts in $B^{(d)}$ to maintain parallelism.

[7] By convention, all slots in American crossword puzzles must have at least three letters, and all grid cells must participate in an across and down slot. Puzzles are point symmetric. We fold out reflections and rotations because candidates are randomly created and are thus symmetric on average.

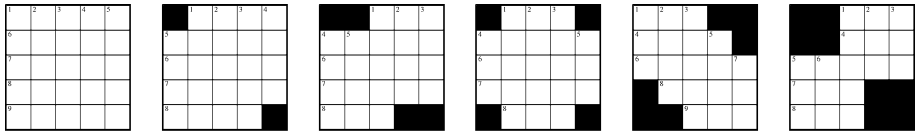


Fig. 8. After symmetries have been removed, there are six tournament-legal 5×5 crossword grids.

Table 4
The solution with maximum $\prod p$ is most likely, while the solution with maximum $\sum q$ has the most in common on average with a randomly generated solution. Averages are taken over the 600 randomly generated puzzles

Maximized quantity	P	Q	$\frac{P}{P(maxP)}$	$\frac{Q}{Q(maxQ)}$
$P \propto \prod p$	0.0552	3.433	1.00	0.943
$Q = \sum q$	0.0476	3.639	0.862	1.00
$Q^{(100)} = \sum q^{(100)}$	0.0453	3.613	0.820	0.993

Table 5
Different grid patterns generated different numbers of solutions. The probability and expected overlap of solutions varied with grid pattern. All numbers in the table are averages over 100 random puzzles

Number of solutions	$P(maxP)$	$Q(maxQ)$	$\frac{Q(maxP)}{Q(maxQ)}$	$\frac{Q(maxQ^{(100)})}{Q(maxQ)}$
A: 32846	0.004	1.815	0.854	0.994
B: 7930.8	0.014	2.555	0.921	0.991
C: 2110.2	0.033	3.459	0.925	0.992
D: 2025.4	0.034	3.546	0.940	0.994
E: 520.9	0.079	4.567	0.961	0.992
F: 131.1	0.167	5.894	0.980	0.993

posterior probabilities q on each slot. We also used the iterative approximation to compute approximate posterior probabilities $q^{(0)}, \ldots, q^{(100)}$. We found the solutions with maximum probability (*maxP*), maximum expected overlap (*maxQ*), and maximum approximate expected overlap ($maxQ^{(0)}, \ldots, maxQ^{(100)}$). For each of these solutions, we calculated its probability (P), expected overlap (Q), and the percent of optimum achieved. The results, given in Table 4, confirm the difference between the maximum probability solution and the maximum expected overlap solution. The solution obtained by maximizing the approximate expected overlap ($Q^{(100)}$) scored an expected overlap 5% higher than the maximum probability solution, less than 1% below optimum.

Over the six grids, the final approximation ($maxQ^{(100)}$) consistently achieved an expected overlap of between 99.1 and 99.4% of the optimal expected overlap $Q(maxQ)$ (see Table 5). The expected overlap of the maximum probability solution $Q(maxP)$ fell from 98.0 to 85.4% of optimal expected overlap as puzzles became less constrained

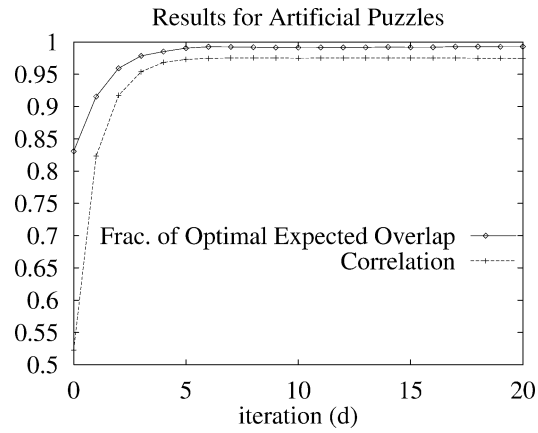


Fig. 9. Successive iterations yield better approximations of the posterior probabilities.

(F to A). One possible explanation is that puzzles with fewer solutions tend to have one "best" solution, which is both most likely and has a high expected overlap with random solutions.

The approximation tended to improve with iteration. The lower curve of Fig. 9 shows the correlation of the approximate posterior $q^{(d)}$ with the true posterior q. The upper curve shows the expected overlap of the solution that maximizes $Q^{(d)}$ ($maxQ^{(d)}$) divided by that of the maximum expected overlap solution. The approximate posterior probabilities $q^{(d)}$ seemed to converge in all cases and, for all of the 600 test puzzles, the maximum expected overlap solution was constant after iteration 38.

Computing the maximum probability solution and the maximum approximate expected overlap solution both involve finding an instantiation that maximizes the sum of a set of weights. In the first case, our weights are $\log(p_x(v))$ and, in the second case, they are $q_x^{(d)}(v)$. This is an NP-complete problem, and in both cases, we solve it with an A* search [5]. Our heuristic estimate of the value of a state is the sum of the weights of the values of all of its assigned variables and of the maximum weight of the not-yet-considered values of the unassigned variables.

In our set of artificial puzzles, this A* search was much faster when maximizing $\sum q^{(100)}$ than when maximizing $\prod p$. The former took an average of 47.3 steps, and the latter 247.6 steps. Maximizing $\sum q^{(d)}$ got faster for successive iterations d as shown in Fig. 10. We believe that optimizing $\sum q^{(d)}$ is faster because the top candidates have already shown themselves to fit well into a similar network ($U^{(d)}$), and therefore are more likely to fit with each other in the puzzle grid.

4.5. Implementation and discussion

To apply the solver to the weighted candidate lists output from the merger, a number of approximations were made. The solver used an implementation of A* to find the solution that maximizes the approximate expected overlap score $Q^{(d)}$ for each iteration d from 0 to 25. In a small number of instances, however, A* required too much memory to complete,

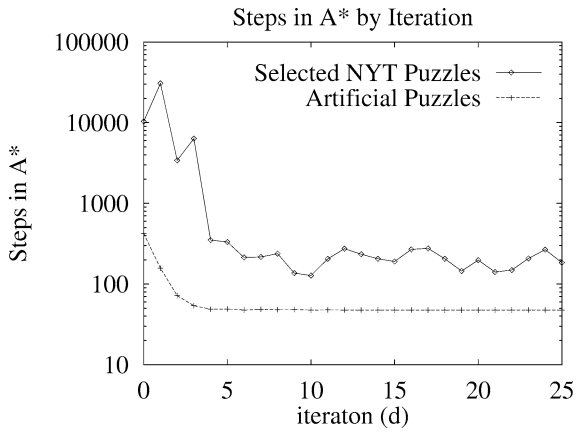


Fig. 10. Maximizing the approximate expected overlap via A* tended to get faster with successive iterations of our approximation.

and we had PROVERB switch to a heuristic estimate that was slightly inadmissible (admissible plus a small amount) to ensure that some solution was found. Maximizing $Q^{(d)}$ tended to be easier for greater d. The inadmissible heuristic was required in 47 of 70 test puzzles in maximizing $Q^{(1)}$ but only once in maximizing $Q^{(25)}$. Fig. 10 plots the number of steps required by A* for each iteration, averaged over the 23 puzzles where the inadmissible heuristic was unused.

Note that, because of the implicit bigram distribution (Section 3.3), all possible patterns of letters have non-zero probability of being a solution. As noted in Table 5, the maximum probability solution tends to give a poor approximation of the maximum overlap solution when there are many solutions; thus, the iterative approximation plays an important role in this type of puzzle.

Because of some of the broad-coverage expert modules, candidate lists are extremely long (often over 10^5 candidates), which makes the calculation of our approximate posterior probabilities $q^{(d)}$ expensive. To save time, we compute $b^{(d)}$ using *truncated* candidate lists. To begin, these lists contain the candidates with the greatest priors: We remove all candidates with prior probability less than a factor of 10^{-3} of the greatest prior from the list. Doing this usually throws out some of the correct targets, but makes the lists shorter. To bring back a possibly correct target once the approximation has improved, at every iteration we "refresh" the candidate lists: We compute $q^{(d)}$ for all candidates in the full list (based on $b^{(d-1)}$ of only the truncated list). We discard our old abbreviated list and replace it with the list of candidates with the greatest $q^{(d)}$ values (at least 10^{-3} of the maximum). The missing probability mass is distributed among candidates in the implicit bigram-letter model. (In a faster version of the solver we only refresh the candidate lists once every seven iterations. This does not appear to affect accuracy.)

As we mentioned, our approximate inference is Pearl's belief propagation algorithm on loopy networks. This approximation is best known for its success in decoding turbo codes [12], achieving error correcting code performance near the theoretical limit. In retrospect, it is not surprising that the same approximation should yield such positive

results in both cases. Both problems involve reconstructing data based on multiple noisy encodings. Both networks contain many cycles, and both are bipartite, so all cycles have length at least four.

Controlled experiments with NYT puzzles can be found elsewhere [19]. In the next section, we describe experiments on a large set of real puzzles.

5. Results

To evaluate PROVERB's performance, we ran it on a large collection of daily puzzles and on puzzles from two recent human championships.

5.1. Example puzzle

First, let's look at how PROVERB behaves on the October 10th, 1998 NYT example puzzle. Fig. 11 illustrates the grid after 11 iterations of the solver. Each grid square shows all 26 letters with their darkness proportional to their estimated posterior probability (it's like penciling in all the guesses, but pushing harder on the pencil for the more certain answers). The squares themselves are shaded if the most probable letter in the square is not correct.

This series of snapshots gives a compelling picture of how the solver's accuracy increases over successive iterations. At iteration 0 (using priors as estimates of posterior probabilities), only the most confident answers are correct. Some examples are: ≺**46A** Highball ingredient: `rye`≻, ≺**54A** Transfix: `impale`≻, ≺**51D** Bugs chaser: `elmer`≻ and ≺**25D** "Rouen Cathedral" painter: `monet`≻. At this stage, maximizing the sum of the posterior estimates using A* results in 30% words correct.

After the first iteration, crossing words are already beginning to support each other. A* gets 38% words correct based on these estimates, and this increases to 55% after iteration 2. At this point, the middle left section of the grid has solidified. By the end of iteration 5, A* gets 81% words correct as several more sections of the grid have settled. After iteration 11, the process has converged and the system completes with 88% words correct. This is actually a low score for an average puzzle, but a bit higher than average for late week NYT puzzles.

The eight missed clues at the end are ≺**1A** Knee-slapper: `hotone`≻, ≺**21A** Redtagged: `onsale`≻, ≺**33A** Eager to try: `keenon`≻, ≺**62A** Not kissproof: `smeary`≻, ≺**1D** Can't help but: `hasto`≻, ≺**3D** "The Way to Natural Beauty" author, 1980: `tiegs`≻, ≺**9D** Having little exposure: `lowrisk`≻, and ≺**52D** Starting to break down: `teary`≻. Five of these are somewhat obscure multi-word combinations and two are syntactic problems (`tears`/`smears` instead of `teary`/`smeary`). The last problem is that we missed former supermodel Cheryl Tiegs on our list of famous authors. A later version of PROVERB that includes web search modules was able to solve this clue.

In visualizing the behavior of the solver, it is also useful to look at how individual slots change during solving. For example, at iteration 3, ≺**60A** "Belvedere" artist: `escher`≻ is filled by `etcher` (because of the artist-etcher relationship), which is repaired the next iteration when ≺**34D** Egad, e.g.: `euphemism`≻ comes through. ≺**61A** Rusty locale: `junkyard`≻ is `backyard` (another locale) through most of the iterations. It becomes

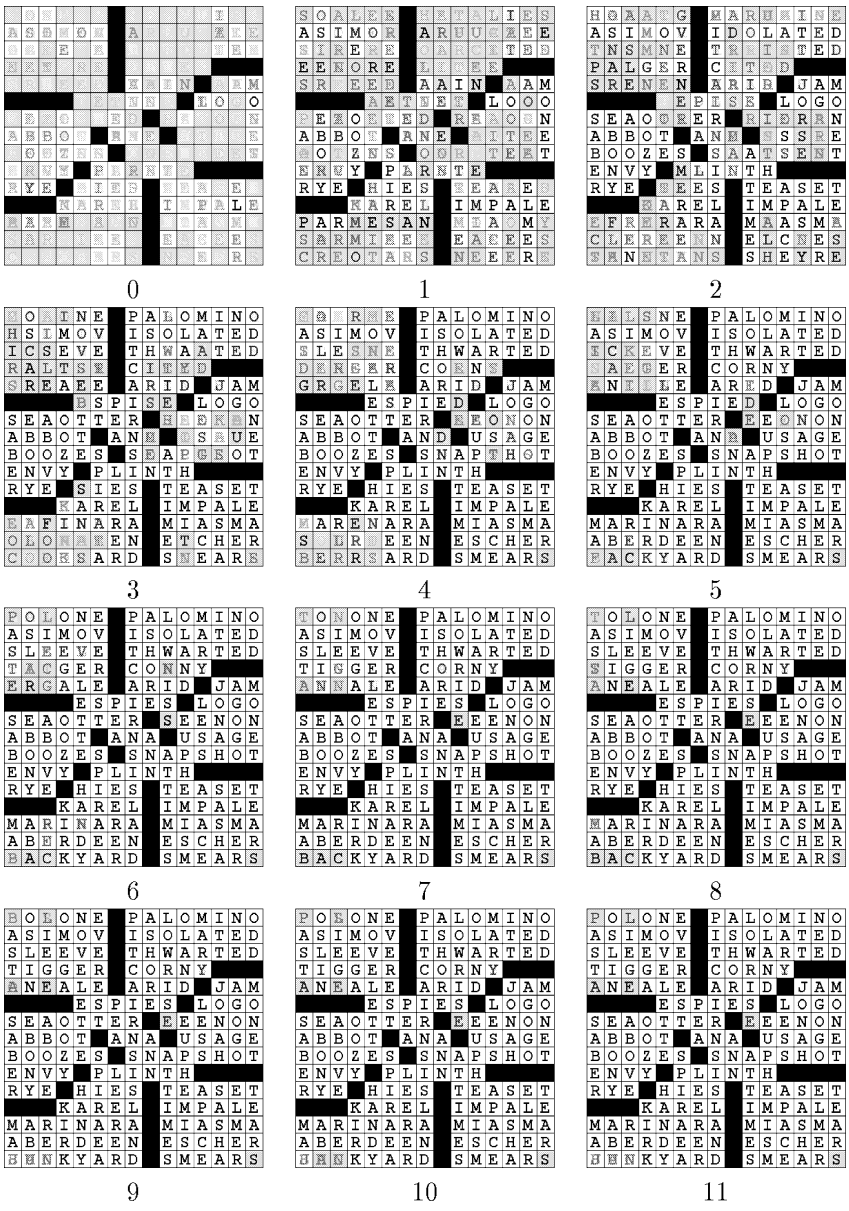


Fig. 11. The performance of the system improves steadily with iterations on the October 10th, 1998 NYT example puzzle.

`junkyard` (though not very strongly) after a candidate list refresh at iteration 9. The final `-ed` for ≺**18A** Stymied: `thwarted`≻ is already evident in the first iteration because, although there is uncertainty about the correct fill, most of the high ranking possibilities are past-tense words.

It is difficult, although interesting, to follow precisely the behavior of PROVERB on any given puzzle. In the next two sections, we examine statistics averaged on sets of puzzles.

5.2. Daily puzzles

We tested the system on puzzles from seven daily sources, listed in Table 1. 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 in Section 3.4 was tuned 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. Experiments were run on two DEC Alphas and approximately 8–10 Sparcstations with about 60M of memory and 2G of shared disk space. PROVERB ran in roughly 15 minutes per puzzle, with about 7 minutes for candidate generation and 8 minutes for grid filling. Table 6 reports both word score (percent of words correct) and letter score (percent of letters correct in the grid). The system achieved 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). On average, the implicit distribution models helped, although this was not always true on a puzzle-by-puzzle basis. PROVERB performed similarly on all seven sources, with two mild outliers: NYT was harder than average and LAT was slightly easier. We leave it to the reader to speculate on the implications of this observation.

In Fig. 12, we plot the scores on each of the 370 daily puzzles attempted by PROVERB, grouped by the source. In addition, we split the NYT puzzles into two groups: Monday through Wednesday (MTW), and Thursday through Sunday (TFSS). As noted earlier, there is an effort made at the NYT to make puzzles increasingly difficult as the week progresses, and with respect to PROVERB's performance they have succeeded.

Table 6
Summary of results from running PROVERB on 370 crossword puzzles from seven different sources, with and without the implicit modules shows average percent words correct

		W/O Implicit			W/ Implicit		
Source	#	Word	Letter	Perfect	Word	Letter	Perfect
NYT	70	86.4	93.6	11.4	89.5	95.3	18.6
LAT	50	97.4	99.1	56.0	98.0	99.3	60.0
USA	50	96.0	98.7	44.0	96.8	98.9	54.0
CS	50	94.4	97.4	54.0	95.2	97.7	54.0
CSS	50	94.6	98.1	30.0	96.3	98.8	50.0
UNI	50	97.1	99.3	46.0	97.5	99.4	46.0
TVG	50	96.0	98.6	52.0	96.2	98.6	52.0
Total	370	94.1	97.6	37.6	95.3	98.1	46.2

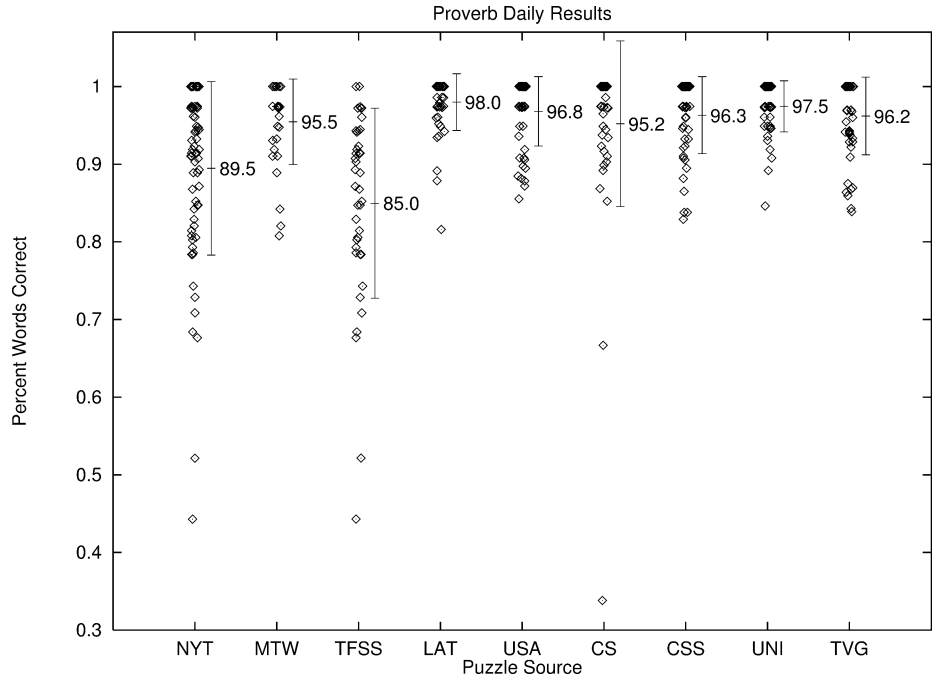


Fig. 12. PROVERB's performance is strong on a variety of daily crossword puzzles.

5.3. Tournament puzzles

To better gauge the system's performance against humans, we tested PROVERB using puzzles from the 1998 American Crossword Puzzle Tournament (ACPT) [20] (http://www.crosswordtournament.com/). 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 official 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 along with some human scores for comparison are shown in Table 7. 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 could have placed 190 out of 251; its score on Puzzle 5 exceeded that of the median human solver at the contest.

The ACPT puzzles are very challenging. In addition to the standard theme puzzles (Puzzle 7 included twelve targets that were standard phrases written as if pronounced with a Southern accent: ≺What a klutz does at the bakery?: `getshisfootinthedough`≻),

Table 7
PROVERB compared favorably to the 251 elite human contestants at the 1998 championship. Lines preceded by a ▷ indicate the theoretical scores if the solver finished every puzzle in under a minute. PROVERB-I used the implicit distribution modules

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

several included tricks that are "illegal" in crossword puzzles. For example, Puzzle 2 was called *Something Wicked This Way Comes* and included ≺Coca-Cola Co. founder: `asacandler`≻ intersecting at its fourth letter with the first letter of ≺Seagoer of rhyme: `candlestickmaker`≻. To complete this puzzle, the solver must realize that the word "candle" has to be be written within a single cell wherever it appears. Puzzle 5, *Landslides*, required that country names be written one slot away from their clued slot in order for the crossing words to fit. Tricks like these are extremely uncommon in published crosswords, appearing in perhaps fewer than one in 500 puzzles. In the tournament, 3 of 7 puzzles included tricks. In spite of the fact that PROVERB could not produce answers that bend the rules in these ways, it still correctly 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 they ran more slowly.

Throughout the development of PROVERB, we were in touch with Will Shortz. In addition to being the influential editor of the NYT puzzles, he has also ran the ACPT since its inception. He has publicly expressed skepticism in the idea of computers solving challenging crossword puzzles [21], so proving him wrong was one of the forces motivating us.

We showed Will Shortz the results described above early in 1999 and he asked us to "participate" in the 1999 competition. He sent the seven puzzles in electronic form about a week in advance of the competition, which we ran through the system. We wrote a "play-by-play" guide to PROVERB's results on each puzzle, which was made available to the competitors one puzzle at a time during the tournament.

PROVERB got 75% words correct on this batch of puzzles, placing it approximately in 147th place out of 254 competitors (again assuming sub-minute execution time). On one puzzle, PROVERB's score was in the top 20. On another, however, PROVERB answered only 3 clues correctly and came in 249th place.

Quick examination of the problematic puzzle revealed the reason for PROVERB's difficulty: in another crossword trick, all the clues had been spoonerized (initial sounds

transposed). An example is ≺Home is near: alaska≻, since unspoonerized, the clue becomes "Nome is here". PROVERB correctly solved this clue, probably due to its knowledge that the Near Islands lie off the coast of Alaska. (This observation was first made by contestants and then verified by running the Dijkstra module.) PROVERB finds an unexpected answer for ≺Barry or Hess: truman≻ ("Harry or Bess"), namely "eugene" since Eugene Barry and Eugene Hess are both in its actors database. Note that when we "unspoonerized" the clues by hand, PROVERB achieved a perfect score on this puzzle. This reinforces the fact that this puzzle is actually quite easy, even though PROVERB is unable to solve it. We believe that Will Shortz specifically included this puzzle in the tournament to put PROVERB in its place given its impressive performance on standard puzzles. In any event, he retracted his earlier claim that computers would be unlikely to match human performance on tough puzzles in a short essay describing PROVERB [22].

6. 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 found probability theory a potent practical tool for organizing the system and improving performance. It is worth mentioning that these ideas are sufficiently mature that we were able to use them to construct our entire system as a group project in a single semester.

The level of success we achieved would probably not have been possible six 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. These resources make it possible for researchers to address other language-related games [9] as well as other important applications in automated natural language processing.

Although we developed a strong solver in this first attempt, champion level performance seems to require much greater understanding of puzzle themes and tricks. In contrast to search-based games, faster computers alone are not likely to lead to improved performance. Reaching a higher level may depend on new breakthroughs in reasoning algorithms and learning language knowledge from text. We contend that crossword puzzles are an excellent testbed for this type of research and plan to return to them as natural language technology becomes more mature.

Acknowledgements

System development was carried out at Duke University as part of a graduate seminar. Sushant Agarwal, Catherine M. Cheves, Joseph Fitzgerald, Jason Grosland, Fan Jiang,

Shannon Pollard, Karl Weinmeister helped develop PROVERB and contributed text to the original conference paper [7]. We also 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, Nabil H. Mustafa, Alan Biermann, Donald Loveland, Gert Webelhuth, Robert Vila, Sam Dwarakanath, Will Portnoy, Michail Lagoudakis, Steve Majercik, Syam Gadde. Thanks to Rina Dechter, Moises Goldszmidt, Martin Mundhenk, Mark Peot, and Yair Weiss for feedback and suggestions on the grid-filling work. Fred Piscop, Merl Reagle, Will Shortz, and William Tunstall-Pedoe and made considerable contributions. We thank Jonathan Schaeffer and the University of Alberta GAMES group for support and encouragement and our anonymous reviewers for extremely helpful comments.

References

[1] S. Abney, Statistical methods and linguistics, in: J. Klavans, P. Resnik (Eds.), The Balancing Act, MIT Press, Cambridge, MA, 1996, Chapter 1, pp. 2–26.
[2] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, R.A. Harshman, Indexing by latent semantic analysis, J. Amer. Soc. Inform. Sci. 41 (6) (1990) 391–407.
[3] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-completeness, Freeman, San Francisco, CA, 1979.
[4] M.L. Ginsberg, M. Frank, M.P. Halpin, M.C. Torrance, Search lessons learned from crossword puzzles, in: Proc. AAAI-90, Boston, MA, 1990, pp. 210–215.
[5] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Systems Sci. Cybernet. SSC-4 (2) (1968) 100–107.
[6] B.J. Jansen, A. Spink, J. Bateman, T. Saracevic, Real life information retrieval: A study of user queries on the web, SIGIR Forum 32 (1) (1998) 5–17.
[7] G.A. Keim, N. Shazeer, M.L. Littman, S. Agarwal, C.M. Cheves, J. Fitzgerald, J. Grosland, F. Jiang, S. Pollard, K. Weinmeister, Proverb: The probabilistic cruciverbalist, in: Proc. AAAI-99, Orlando, FL, 1999, pp. 710–717.
[8] T.K. Landauer, S.T. Dumais, A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction and representation of knowledge, Psychological Review 104 (2) (1997) 211–240.
[9] M.L. Littman, Review: Computer language games, in: T.A. Marsland, I. Frank (Eds.), Computers and Games 2000, Lecture Notes in Computer Science, Vol. 2063, Springer, New York, 2001.
[10] A.K. Mackworth, Constraint satisfaction, in: S.C. Shapiro (Ed.), Encyclopedia of Artificial Intelligence, Vol. 1, 2nd Edition, Wiley, New York, 1992, pp. 285–293.
[11] A.K. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1) (1977) 99–118.
[12] R. McEliece, D. MacKay, J. Cheng, Turbo decoding as an instance of Pearl's 'belief propagation' algorithm, IEEE J. Selected Areas in Communication 16 (2) (1998) 140–152.
[13] G.R. Miller, C. Beckwith, C. Fellbaum, D. Gross, K. Miller, Introduction to WordNet: An one-line lexical database, Internat. J. Lexicography 3 (4) (1990) 235–244.
[14] J.R. Munkres, Topology, A First Course, Prentice-Hall, Englewood Cliffs, NJ, 1975.
[15] J. Pearl, Probabilistic Reasoning in Intelligent Systems, 2nd Edition, Morgan Kaufmann, San Mateo, CA, 1988.
[16] D. Roth, On the hardness of approximate reasoning, Artificial Intelligence 82 (1–2) (1996) 273–302.
[17] G. Salton, M.J. McGill, Introduction to Modern Information Retrieval, McGraw-Hill, New York, 1983.
[18] T. Schiex, H. Fargier, G. Verfaillie, Valued constraint satisfaction problems: Hard and easy problems, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 631–637.
[19] N.M. Shazeer, M.L. Littman, G.A. Keim, Solving crossword puzzles as probabilistic constraint satisfaction, in: Proc. AAAI-99, Orlando, FL, 1999, pp. 156–162.
[20] W. Shortz (Ed.), American Championship Crosswords, Fawcett Columbine, 1990.

[21] W. Shortz (Ed.), Introduction to The New York Times Daily Crossword Puzzles, Vol. 47, Random House, New York, 1997.
[22] W. Shortz (Ed.), Introduction to The New York Times Daily Crossword Puzzles, Vol. 53, Random House, New York, 1999.
[23] C. Silverstein, M. Henzinger, H. Marais, M. Moricz, Analysis of a very large AltaVista query log, SRC Technical Note 1998-014, 1998.
[24] Y. Weiss, Belief propagation and revision in networks with loops, Technical Report 1616, MIT AI Lab, Cambridge, MA, 1997.