Solving Crossword Puzzles as Probabilistic Constraint Satisfaction

Noam M. Shazeer, Michael L. Littman, Greg A. Keim
Department of Computer Science
Duke University, Durham, NC 27708-0129
{noam,mlittman,keim}@cs.duke.edu

Abstract

Crossword puzzle solving is a classic constraint satisfaction problem, but, when solving a real puzzle, the mapping from clues to variable domains is not perfectly crisp. At best, clues induce a probability distribution over viable targets, which must somehow be respected along with the constraints of the puzzle. Motivated by this type of problem, we describe a formal model of constraint satisfaction with probabilistic preferences on variable values. Two natural optimization problems are defined for this model: maximizing the probability of a correct solution, and maximizing the number of correct words (variable values) in the solution. To the latter, we apply an efficient iterative approximation equivalent to turbo decoding and present results on a collection of real and artificial crossword puzzles.

Introduction

Constraint satisfaction is a powerful and general formalism. Crossword puzzles are frequently used as examples of constraint satisfaction problems (CSPs), and search can be used to great effect in crossword-puzzle creation (Ginsberg *et al.* 1990). However, we are not aware of any attempts to apply CSPs to the problem of solving a crossword puzzle from a set of clues. This is due, in part, to the fact that traditional CSPs have no notion of "better" or "worse" solutions, making it difficult to express the fact that we prefer solutions that fill the grid and match the clues to ones that simply fill the grid.

To address this problem, this paper describes a probabilistic extension to CSPs that induces probability distributions over solutions. We study 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 apply an efficient iterative approximation equivalent to Pearl's belief propagation algorithm (Pearl 1988) on a multiply connected network.

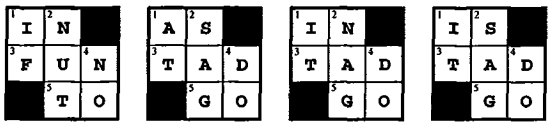
We describe how the two optimization problems and the approximation result in different solutions on a collection of artificial puzzles. We then describe an extension to our solver that has been applied to a collection of real New York Times crossword puzzles. Our system achieves a score of 89.5% words correct on average, up from 51.8% for a more naive approximation.

Constraint Satisfaction Problems

We define a (Boolean) constraint satisfaction problem (Mackworth 1977), or CSP, as a set of variables and constraints on the values of these variables. For example, consider the crossword puzzle in Figure 1. 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 Figure 1 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. The example we consider in this paper 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 [3]" and generate a small set of candidate answers of the appropriate length to be the domain: TAD, JOT, DAB, BIT, for example.

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 automated natural language processing may cause a



	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

Figure 1: 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.

reasonable candidate to be excluded. 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" (Abney 1996). 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 $1 \leq i \leq n$, $\sum_{v \in D_i} p_{x_i}(v) = 1$ and for all $v \in D_i$, $p_{x_i}(v) \geq 0$. This is a special case of probabilistic CSPs (Schiex, Fargier, & Verfaillie 1995). An opportunity for future work is to extend the algorithms described here to general probabilistic CSPs.

In the crossword example, probabilities can be chosen by a statistical analysis of the relation between the clue and the candidate; we have adopted a particular approach to this problem, which we sketch in a later section. Extending the running example, we can annotate the domain of each variable with probabilities, as shown in Figure 1 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 $p_{2\mathrm{D}}(\mathtt{NUT}) = 0.3$.

We next need to describe how preferences on values can be used to induce preferences over complete solutions. We consider the following probability model. 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 Figure 1 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:

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

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

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. In this paper, we consider two optimization problems on CSPs with probabilistic preferences: maximum probability solution and maximum expected overlap solution.

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 Figure 1). It can be found by computing

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

That is, we just need to search for the solution that maximizes the product of the preferences p. This is an NP-complete problem (Garey & Johnson 1979), 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 the previous section, 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.

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 (Shortz 1990). 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 the 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 figure 1 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 $q_{2\mathrm{D}}(\texttt{SAG}) = \Pr(\mathbf{B}) + \Pr(\mathbf{D}) = 0.250 + 0.133 = 0.383$. For the expected overlap Q, we have

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

By the linearity of expectation,

$$\begin{aligned} &\operatorname*{argmax}_{\text{soln}:v_1,\ldots,v_n} Q(v_1,\ldots,v_n) \\ &= \operatorname*{argmax}_{\text{soln}:v_1,\ldots,v_n} \sum_{i=1}^{n} q_{x_i}(v_i), \end{aligned} \quad (2)$$

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 (Roth 1996).

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.

Estimating the Posteriors

Constraint satisfaction problems with probabilistic preferences have elements in common with both constraint networks and belief networks (Pearl 1988). So, it is not surprising that, 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 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, Figure 2(a) gives the constraint network form of the crossword puzzle from Figure 1. Figures 2(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

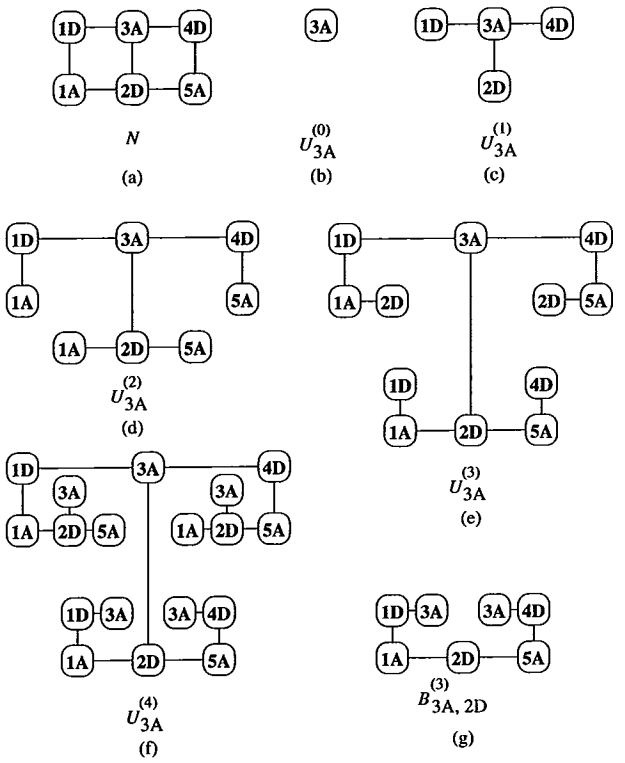


Figure 2: A cyclic constraint network can be approximated by tractable tree-structured constraint networks.

those in the cyclic network. This construction parallels the notion of a universal covering space from topology theory (Munkres 1975).

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 posteriors on unwrapped networks has been shown equivalent to Pearl's belief propagation algorithm (Weiss 1997), which is exact on singly connected networks but only approximate on loopy ones (Pearl 1988).

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 $B_{x,y_i}^{(d)}$ as the y_i-branch of $U_x^{(d+1)}$, or equivalently, $U_{y_i}^{(d)}$ with the x-branch removed (see Figure 2(g)). Let $b_{x,y_i}^{(d)}(w)$ be the posterior probability that y_i takes value w in the network $B_{x,y_i}^{(d)}$. Note that $U_x^{(0)}$ and $B_{x,y_i}^{(0)}$ contain the single variables x and y_i respectively. Thus,

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

For positive d, we view $U_x^{(d)}$ as a tree with root x and branches $B_{x,y_i}^{(d-1)}$. 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 $b_{x,y_i}^{(d-1)}$. 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

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

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[1], the equation for $b_{y_i,x}^{(d)}(v)$ is very similar to the one for $q_x^{(d)}(v)$:

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

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 converges 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 $d \geq k$. 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 infinite. 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 our crossword solver, the candidate lists are very large, so s is enormous. To reduce the value of s, we inserted

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

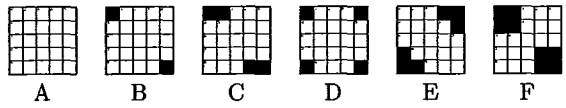


Figure 3: After symmetries have been removed, there are six tournament-legal 5 × 5 crossword grids.

Maximized Qty.	P	Q	$\frac{P}{P(maxP)}$	$\frac{Q}{Q(maxQ)}$
$P \propto \prod p$	.0552	3.433	1.00	.943
$Q = \sum q$	.0476	3.639	.862	1.00
$Q^{(100)} = \sum q^{(100)}$	.0453	3.613	.820	.993

Table 1: 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.

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.

Crossword Results

We applied the iterative approximation method to optimize the expected overlap of a set of artificial and real crossword puzzles.

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[2], as shown in Figure 3. 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 number of solutions are shown in Table 2), from which we derived the exact posteriors q on each slot. We also used the iterative approximation to compute approximate posteriors $q^{(0)}, \ldots, q^{(100)}$. 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 1, 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.

[2]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. We fold out reflections and rotations because candidates are randomly created and are thus symmetric on average.

	#solns	$P(maxP)$	$Q(maxQ)$	$\frac{Q(maxP)}{Q(maxQ)}$	$\frac{Q(maxQ^{(100)})}{Q(maxQ)}$
A:	32846	.004	1.815	.854	.994
B:	7930.8	.014	2.555	.921	.991
C:	2110.2	.033	3.459	.925	.992
D:	2025.4	.034	3.546	.940	.994
E:	520.9	.079	4.567	.961	.992
F:	131.1	.167	5.894	.980	.993

Table 2: 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.

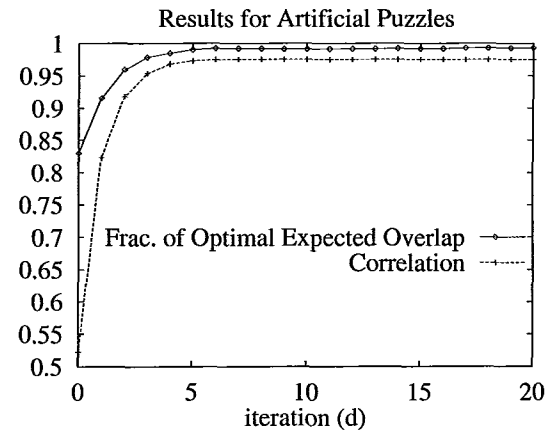


Figure 4: Successive iterations yield better approximations of the posteriors.

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 2). 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 (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 Figure 4 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 approxi-

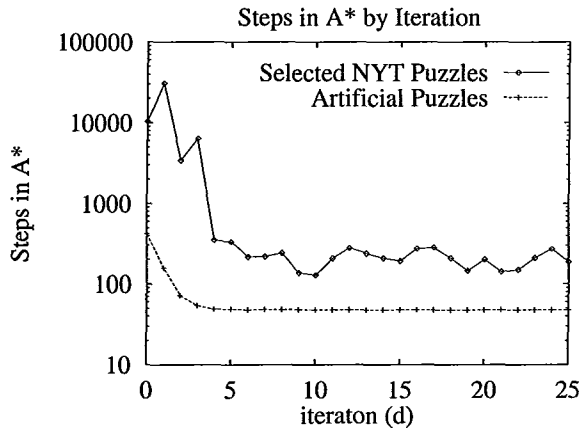


Figure 5: Maximizing the approximate expected overlap with A* tended to get faster with successive iterations of our approximation.

mate posteriors $q^{(d)}$ 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. 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 is 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 Figure 5.

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.

Real Puzzles

We adapted our approach to solve published crossword puzzles. Candidate lists are generated by a set of thirty expert modules using a variety of databases and techniques for information retrieval (Keim *et al.* 1999). Each module returns a weighted list of candidates, and these lists are combined according to a set of parameters trained to optimize the mean log probability assigned to the correct target.

Without returning all possible letter combinations, it is impossible for our expert modules to always return the correct target in their candidate lists; in fact, they miss it about 2.1% of the time. To ensure that solutions exist and that the correct solution is assigned a positive probability, we implicitly represent the probability distribution over all letter strings according to a letter-bigram model. The total probability assigned to this model is learned along with the weights on the expert modules. Because of its simple form, the system is able to manipulate this distribution efficiently to calculate $b^{(d)}$ and $q^{(d)}$ correctly on the explicit candidates. The full solver includes a combination of several of these "implicit distribution modules."

Note that, because of the implicit bigram distribution, all possible patterns of letters have non-zero probability of being a solution. As noted in Table 2, 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.

The solver itself 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, and we switched 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 $Q^{(25)}$. Figure 5 plots the number of steps required by A* for each iteration, averaged over the 23 puzzles where the inadmissible heuristic was unused.

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 posteriors $q^{(d)}$ 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 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.)

Figure 6 shows the fraction of words correct for the solutions that maximized $Q^{(0)}$ through $Q^{(25)}$. Performance increased substantially, from 51.8% words correct at iteration zero to 89.5% before iteration 25. Figure 6 also shows the fraction of slots for which the candidate with the maximum $q^{(d)}$ is the correct target. This

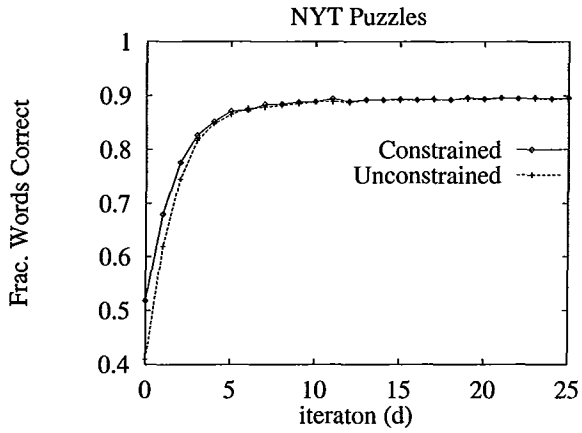


Figure 6: Average number of words correct on a sample of 70 New York Times puzzles increases with the number of iterations. This graph shows two measures, one for solutions constrained to fit the grid, and one unconstrained.

would be our score if our solution did not need to satisfy the constraints. Note that the same candidate lists are used throughout—the improvement in performance is due to better grid filling and not to improved clue solving. We have also run the solver on puzzles less challenging than those published in the New York Times and achieved even better results; the average score on 50 LA Times puzzles, was 98.0% words correct.

Relationship to Turbo Codes

To perform our approximate inference, we use Pearl's belief propagation algorithm on loopy networks. This approximation is best known for its success in decoding turbo codes (McEliece, MacKay, & Cheng 1998), 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.

Conclusions

Faced with the problem of solving real crossword puzzles, we applied an extension of CSPs that includes probabilistic preferences on variable values. The problem of maximizing the number of correct words in a puzzle was formalized as the problem of finding the maximum expected overlap in the CSP. We applied an iterative approximation algorithm for this problem and showed that it is accurate on a collection of artificial puzzles. As a happy side effect, the proposed iterative approximation algorithm speeds optimization. After extending the resulting algorithm to handle real puzzles with implicitly defined candidate lists, the solver scored 89.5% words correct on a sample of challenging New York Times crossword puzzles.

Having identified the importance of maximum overlap score in the crossword domain, we believe that this measure could be useful in other problems. For example, in machine vision, we might be interested in a consistent interpretation for a scene that is expected to have as much in common with the true scene as possible; this could be formalized in a manner similar to our crossword puzzle problem.

All in all, this work suggests that combinations of probability theory and constraint satisfaction hold promise for attacking a wide array of problems.

Acknowledgments. Thanks to Rina Dechter, Moises Goldszmidt, Martin Mundhenk, Mark Peot, Will Shortz, and Yair Weiss for feedback and suggestions.

References

Abney, S. 1996. Statistical methods and linguistics. In Klavans, J., and Resnik, P., eds., *The Balancing Act.* Cambridge, MA: The MIT Press. chapter 1, 2–26.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-completeness.* San Francisco, CA: Freeman.

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.

Keim, G. A.; Shazeer, N.; Littman, M. L.; Agarwal, S.; Cheves, C. M.; Fitzgerald, J.; Grosland, J.; Jiang, F.; Pollard, S.; and Weinmeister, K. 1999. Proverb: The probabilistic cruciverbalist. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence.*

Mackworth, A. K. 1977. Consistency in networks of relations. *Artificial Intelligence* 8(1):99–118.

McEliece, R.; MacKay, D.; and Cheng, J. 1998. Turbo decoding as an instance of Pearl's 'belief propagation' algorithm. *IEEE Journal on Selected Areas in Communication* 16(2):140–152.

Munkres, J. R. 1975. *Topology, A First Course.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems.* San Mateo, CA: Morgan Kaufmann, 2nd edition.

Roth, D. 1996. On the hardness of approximate reasoning. *Artificial Intelligence* 82(1–2):273–302.

Schiex, T.; Fargier, H.; and Verfaillie, G. 1995. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, 631–637.

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

Weiss, Y. 1997. Belief propagation and revision in networks with loops. Technical Report Technical Report 1616, MIT AI lab.