A Fully Automatic Crossword Generator

Conference Paper · January 2009

DOI: 10.1109/ICMLA.2008.104 · Source: IEEE Xplore

4 authors:

Leonardo Rigutini
Università degli Studi di Siena
30 PUBLICATIONS **283** CITATIONS
SEE PROFILE

Michelangelo Diligenti
Università degli Studi di Siena
72 PUBLICATIONS **1,317** CITATIONS
SEE PROFILE

Marco Maggini
Università degli Studi di Siena
170 PUBLICATIONS **2,086** CITATIONS
SEE PROFILE

Maria Cristina Gori
Department of Neurology and Psichiatry
118 PUBLICATIONS **3,233** CITATIONS
SEE PROFILE

A Fully Automatic Crossword Generator

Leonardo Rigutini Michelangelo Diligenti Marco Maggini
Marco Gori
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Siena,
Via Roma 56, Siena, Italy
{rigutini,diligmic,maggini,marco}@dii.unisi.it

Abstract

This paper presents a software system that is able to generate crosswords with no human intervention including definition generation and crossword compilation. In particular, the proposed system crawls relevant sources of the Web, extracts definitions from the downloaded pages using state-of-the-art Natural Language Processing *(NLP) techniques and, finally, attempts at compiling a crossword schema with the extracted definitions using a* Constrain Satisfaction Programming *(CSP) solver. The crossword generator has relevant applications in entertainment, educational and rehabilitation contexts.*

1. Introduction

This paper introduces a software system that generates crosswords with no human intervention, including definition/clue generation and crossword compilation. The system crawls a predefined set of information sources (wiki pages, dictionaries, etc.) and extracts definitions from the downloaded pages using state-of-the-art *Natural Language Processing* (NLP) techniques. Finally, the system compiles a given crossword layout, positioning the extracted definitions on the crossword slots by using a *Constraint Satisfaction Programming* (CSP) solver [10]. Up to our knowledge, this is the first system that is able to fully automatically generate crosswords. Other works attempted at automatically solving crosswords given a human edited vocabulary of definitions [1]. For example, *Proverb* [7] solves the clues using *Natural Language Processing* (NLP) techniques working over a pre-compiled knowledge base. [4] attempts at overtaking the limited flexibility of Proverb by acquiring his knowledge directly from the Web: each clue is sent to a Web search engine, the results are analyzed using NLP techniques and a list of possible answers are selected. *Constraint Satisfaction Programming* (CSP) techniques are used to refine the selected answers in the attempt of finding a combination of candidates that solves the biggest possible portion of the schema. The outline of the paper is the following: section 2 introduces the general architecture of the system. Some experimental result are shown in section 3. Finally, section 4 draws some conclusions.

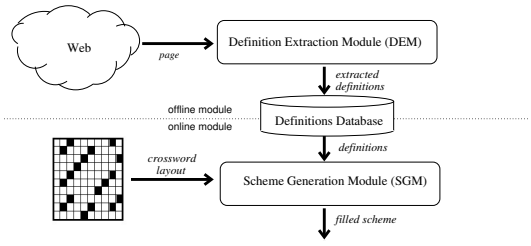
2. System architecture

The architecture of the system is composed by a definition extraction and a scheme generation module. The definition extractor crawls a set of relevant data sources from the Web and discovers new definitions by applying NLP techniques on the downloaded pages. The step is performed offline, and the resulting definitions can be used to generate an arbitrary number of crosswords.

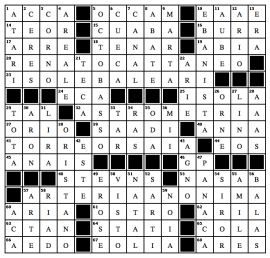
The extracted definitions are then used by the crossword compiler. This module is executed every time that a user requires the generation of a new crossword. Since this module works online, it should provide the result in a reasonably short time. The overall architecture of the system is sketched in fig:1-(a). The system is able to generate high-quality medium sized crosswords with no human interaction. See in fig:1-(b) for an example of a (15×15) crossword generated using our system.

2.1. The Definition Extraction module

The definition extraction module has three components: the information gatherer, the Natural Language analyzer and the definition extractor (see fig:2). The gatherer downloads the pages from the information sources and processes the content of each page discarding the formatting information and retaining the raw text. The text is then passed to the *Natural Language Processing* (NLP) analyzer. The NLP analysis is divided into layers as sketched in the fig:2. Each layer works on the output of the previous layer, with the exception of the first layer that directly works on the input

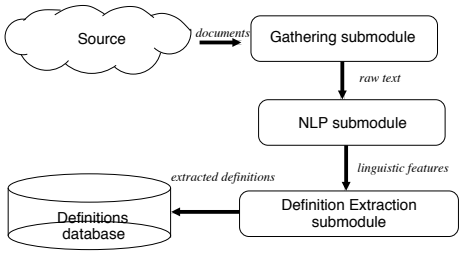


(a)

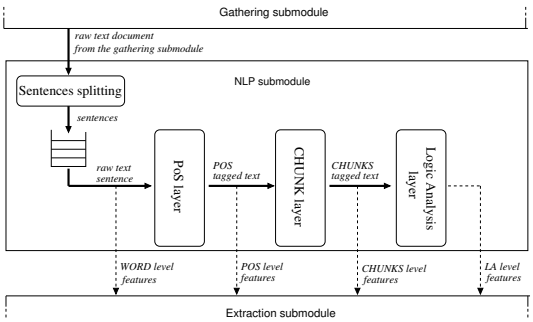


(b)

Figure 1. (a) The system discovers new definitions by processing information sources on the Web. The extracted definitions are used to compile a crossword layout. (b) an example of automatically generated crossword.



(a) Definition extractor



(b) NLP analysis

Figure 2. Architecture of the Definition Extractor and its main module: the NLP analyser.

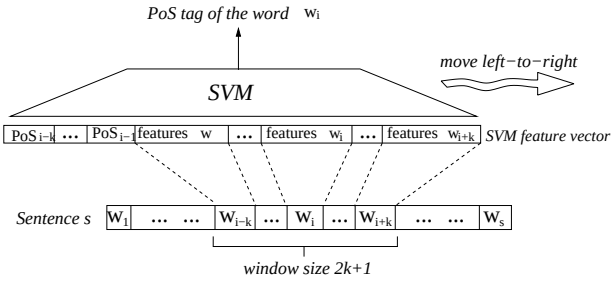


Figure 3. The SVM window-based PoS tagger.

text. Finally, the extractor processes the linguistic features provided by the NLP analyzer and extracts the definitions.

The Part-of-Speech (PoS) layer. The first NLP layer is the *Part-of-Speech* (PoS) tagging A PoS tag summarizes a grammatical property of a word like: noun, proper noun, adjective, or a verb. In the following, $|T_{PoS}|$ indicates the size of the PoS tag list. The PoS tagger processes the input text sentence by sentence and attaches a PoS tag to each word. PoS tagging is a well studied task in the NLP literature. Machine learning approaches are very popular because they can provide state-of-the-art results with little tuning. [3] presents a algorithm to learn tagging rules from a set of examples, while [2] proposes TnT, an efficient statistical PoS tagger. [11] employs a Hidden Markov Model (HMM) to perform tagging, while a stochastic method based on a Maximum Entropy Markov Model (MEMM) is used in [9]. The PoS tagger employed in our system is implemented using a machine learning approach very similar to the model proposed in [5], where the authors propose an effective tagger based on Support Vector Machines (SVM). This SVM-based PoS tagger (see fig:3) scans the input text using a sliding-window with size $2k + 1$ where the tag of the i^{th} word w_i in the sentence is predicted by using the $w_{i-k}, ..., w_i, ...w_{i+k}$ words surrounding w_i. After processing word w_i, the window is moved to the next term. For each word in the window, a set of binary features represents the properties of the raw characters in a word like: "Does the word start with an uppercase letter?", "Is it a number?", etc. Since we use a sliding window approach, the terms preceding the currently processed word w_i are already assigned to a PoS tag. These tags are also included as features of w_i, represented as a set of $|T_{PoS}|$-dimensional binary vectors where the j-th bit is 1 if the word under consideration has been assigned to the j-th PoS tag. Once the feature vector has been constructed for the window under consideration, a set of SVM models tag the input terms. In the training phase, each example is provided to the SVMs, using the target PoS tag as a desired output. When tagging a non previously seen sentence, the feature vector is extracted and processed by the SVM that predicts the tag for the central word of the window.

The performances of the tagger have been measured using the Treebank (TUT) dataset[1], a collection of morpho-

[1] http://www.di.unito.it/∼tuttreeb

	SVM window-based	MEMM
Accuracy	$91.5\% \pm 0.5$	$92.3\% \pm 0.3$
Recall	$91.2\% \pm 0.8$	$92.5\% \pm 0.4$
Precision	$91.4\% \pm 0.3$	$92.0\% \pm 0.3$

Table 1. Results of the SVM and MEMM PoS taggers on the TUT dataset.

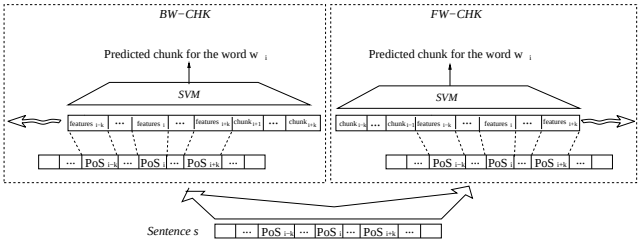


Figure 4. The backward (BW-CHK) and the forward (FW-CHK) chunkers.

logically, syntactically and semantically annotated Italian sentences, used as benchmark at the *EVALITA 2007*[2] evaluation. The set of PoS tags is composed by about 80 tags which can well represent the inflectional richness of Italian. In our implementation, the sliding window has size 5 ($k = 2$) and 20.000 features are extracted for each word. 5-fold cross validation was performed by using 2000 and 1000 tagged sentences as learning set and test set, respectively. The performances of the model have been compared with the MEMM-based tagger described in [9], which reports the state-of-the-art results. As shown in table 1, both models show similar performances on Italian TUT dataset but the SVM-based approach consumes a much smaller amount of memory.

The chunking layer. The second layer is the *chunker* (CHK) which clusters the terms and identifies the base constituents of a sentence (noun phrases, verbs). In grammatical theory, the main constituent types are noun phrase (NP), prepositional phrase (PP) and verbal phrase (VP). The chunker implemented in our system, called TRI-CHK, is an improvement over the model presented in [8], where the authors employed a SVM-based approach using a sliding window over the document. The TRI-CHK is composed by three SVM-based chunkers: a forward chunker which moves its window from left to right (FW-CHK), a backward chunker which moves from right to left (BW-CHK) and a third chunker (R-CHK) which combines the outputs of FW-CHK and BW-CHK. The single FW-CHK and the BW-CHK models are very similar to the PoS tagger described in the previous paragraph, but they uses a different set of features: the output of the PoS layer represented as a $|T_{PoS}|$-dimensional binary vector and, only for the words for which the tag has already been predicted, the chunk tag represented as a binary vector with size $|T_{CHK}|$. The BW-CHK and the FW-CHK chunkers are showed in the fig:4.

[2] http://evalita.itc.it/

	BW-CHK	FW-CHK	TRI-CHK
Accuracy	$88.5\% \pm 0.9$	$89.6\% \pm 0.5$	91.4 ± 0.3
Recall	$86.4\% \pm 0.8$	$88.9\% \pm 0.4$	89.8 ± 0.3
Precision	$90.6\% \pm 0.6$	$90.5\% \pm 0.2$	91.3 ± 0.3

Table 2. The chunking results on TUT dataset.

Table 2 summarizes some experiments we performed to verify the improvement of TRI-CHK compared to BW-CHK and FW-CHK. The results show that TRI-CHK corrects some missed predictions of the single direction chunkers and increases the overall accuracy of chunking by around 2%.

The Logic Analysis layer. In our implementation, the logic-analyzer is implemented as a finite state automata, even if we plan to move toward a machine learning approach in the future. The model processes the chunk sequence predicted by the previous NLP layer and assigns one of the following four syntactic tags to each chunk: subject, nominal predicate, verbal predicate, other. The finite-state automata is also composed by four states, one for each tag. The transitions between the states are activated by the chunk tag under observation, plus a set of boolean predicates representing the composition of the current chunk ("Does the chunk contains a punctuation mark?" etc.) and a set of boolean predicates representing the overall structure of the sentence ("Does the sentence already contain a predicate?", etc.). These features have been defined with the collaboration of the Computational Linguistic group of the University of Siena[3].

The Definition Extractor. The Definition Extractor identifies and extracts the definitions in the sentence, processing as input the linguistic features extracted by the NLP analysis. The Computational Linguistic group of the University of Siena determined that 80% of definitions follow a "nominal structure", which consists of a subject, followed by a nominal predicate and by the complements. The definition extractor is implemented as a finite state automata which observes the sequence of the PoS, chunk and logic analysis tags and decides whether a sentence is a definition. If the phrase is classified as a definition, the pair <subject,definition> is inserted in the database.

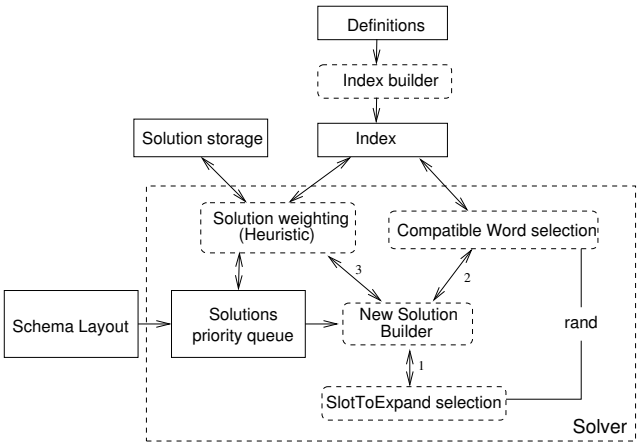


Figure 5. The architecture of the solver.

2.2. Crossword compilation

Crossword compilation from a given vocabulary is a challenging AI search problem, where the branching factor can overtake 10^6 in the early stages of the search and solutions are sparse in the search space For these reasons the heuristics guiding the search are of fundamental importance to converge to a full solution and only small schemas (around 5×5 boxes) can be solved via exhaustive search.

In the following we assume to have available a vocabulary V of definitions (a definition is a word/clue pair). We refer to a *black box* to indicate an element of a crossword that should never be overwritten by a word. A *white box* indicates an element of a crossword that can accommodate a character. A horizontal or vertical consecutive sequence of white boxes forms a *slot*. We will refer to the *layout* of a crossword as the scheleton of the crossword with its boundaries and black boxes. We define a *schema* as a layout partially filled with words.

Crossword solving is a Constraint Satisfaction Programming (CSP) problem where word placement can be aligned either horizontally or vertically. Each placed word is constrained by the layout (*layout constraints*) because a word must start and end adjacent to a crossword boundary or to a black box. Previously inserted words impose additional constraints (*schema constraints*). The layout constraints do not change during the solving process and they simply constrain the length of the words that can be placed in a given position. The schema constraints are instead not present when the schema is empty but they are introduced whenever a new word is inserted. A *compatible* word for a given slot on a schema is a word that respects the actual constraints on the crossword and it could be therefore written on the slot, yielding a new schema that is closer to a full solution.

[3]http://www.ciscl.unisi.it

The core of the solving system is a priority queue which stores the set of partial solutions (schemas). At each step, a new partial solution is popped from the queue and an insertion slot is selected. A set of candidate words are obtained by querying the index to determine the words in the vocabulary that are satisfying the constraints on the selected slot of the schema. Since the index is queried at each iteration, it is essential to make it very fast to avoid it becoming the bottleneck of the search process. As shown in the following of the paper, this task is particularly challenging when the vocabulary of definitions is large.

One-step lookahead is performed to early discard non-promising terms and to compute a more informed score that will be used to rank the candidates in the queue. A certain degree of randomization is allowed both in selecting the insertion slots and in computing the scores used to rank the results in the queue. This allows computing different solutions every time that the solver is ran and to exit from local minimas. The system also keeps a permanent storage of all partial solutions that have been already inserted in the queue at any point of the search. Since there are multiple paths leading to the same partial solution, this avoids loops and wasting computational resources to explore multiple times the same portion of the search space.

The definition index. The basic operation repeated at each step of the search is to obtain the terms in the index that are compatible with the constraints for a slot. First, the layout constraints are used to determine the length of the word that should be inserted in a slot. Secondly, the schema constraints are taken into account whenever there are already characters written on a the slot. We keep a *full-constraint* index where a list of compatible words is stored for any combination of constraints. Querying the index with any set of constraints requires a simple lookup. Unfortunately, the number of possible combinations of g constraints (assuming an alphabet of 26 characters) is 26^g, therefore it is not feasible to create a full index for long terms. In particular, the proposed system employs a full-constraint index to look up terms with length lower than 10. For terms longer than 10 words, our system keeps a *one-constraint* index, where a single constraint is stored as a pair <character, position-in-the-term>. This implementation of the index creates an inverted index associating each constraint to a list of compatible terms. When multiple constraints are imposed by the schema, each single constraint is matched against the index and, then, the returned lists are intersected.

Select a slot to fill. Every time that a new partial solution is popped from the queue, the solver selects one or more slots where to try inserting the compatible words. In our implementation the slots are sorted by the number of white boxes left to be filled. Words with more white boxes are selected as the best choices for the next word insertion. This strategy has been directly borrowed from [6], where

the authors notice limits the search space as early as possible in the search by inserting a higher number of constraints. This strategy has been also experimentally proved to provide good results in many other CSP problems. A certain degree of randomization is allowed to be able to generate different solutions for different runs.

Heuristics to sort the schemas. The definition of a heuristic to sort the partial solutions is crucial for the success of the crossword compilation. Every time that a new term is written on a schema, a score is assigned to the schema considering how close to a full solution the schema is (this factor is proportional to the number of boxes that have been filled) and how likely the schema is to be further expansible. This second factor computes the "goodness" of a term because it measures whether the newly inserted constraints are easy to be satisfied at the next steps. In order to assess the goodness score for an insertion, *one step lookahead* is performed to check how many definitions would be compatible with the newly inserted term in the crossing slots at the next search iteration. Without loss of generality let's refer to the insertion of a word in a horizontal slot. Let $p(c)$ be the probability of finding a future match involving a character of the word written in a box. We assume that $p(c)$ is proportional to the number of definitions that would be compatible after the insertion in the vertical slot passing for the same box: $p(c) \sim num\_compatible\_words(c)$. Assuming term independence across different slots[4], the probability of finding a set of future matches involving all the characters of the inserted term is proportional to the product of the probabilities for each symbol: $p(w) \sim \prod_{i=0}^{g} p(c_i)$ where w is a word of length g and c_i is its i-th character. Our experiments have shown that this lookahead-based heuristic is extremely important to focus the search in a promising direction.

3. Experimental Results

The Definition Extractor was ran over the pages in the Italian Wikipedia[5]. The system discovered about 91.000 definitions (removing the definitions including non alphabetic character that they can not be used in a crossword puzzle). Using a pool of human raters, we measured the percentage of correct and crossword-suitable definitions returned by the system. As shown in table 3-(a), the high accuracy of our NLP analysis allowed keeping the percentage of wrong definitions to a very low value. However, there is a higher portion of definitions that are semantically correct but they can not be used in a real crossword puzzle since they are either too long or they contain terms too similar to the solution itself or they are too ambiguous.

correct	wrong	correct but non usable
81%	3%	16%

(a)

	Ambiguous	Punctual
Automatic	52%	48%
Human	55%	45%

(b)

Table 3. (a) percentage of correct and wrong definitions extracted by our system. (b) ambiguity level for the automatic extracted definitions and the definitions in a set of human edited crosswords.

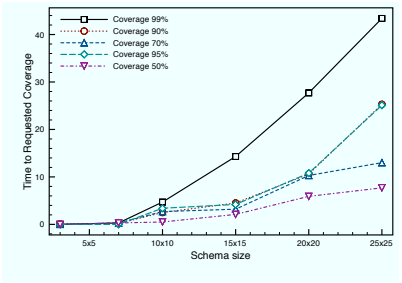
A crossword puzzle should contain a mix of ambiguous (admitting a large set of possible solutions) and punctual definitions. We manually classified as punctual or ambiguous 1000 definitions extracted from crosswords published in various Italian magazines and 1000 definitions automatically extracted by our system. Table 3-(b) compares the obtained ambiguity distributions. Surprisingly, the two distributions look very similar and confirm that our system is able to generate realistic crosswords.

The evaluation of the crossword compilation module has been performed by compiling a set of random layouts using the set of 91000 definitions generated by the definition extractor. The layouts were controlled by two parameters: the vertical/horizontal size N and the percentage of white boxes (defininig the schema difficulty D). Since the horizontal and vertical sizes are always set to same value, all experiments are performed over square-shaped schemas. Obviously, larger values of D and N correspond to schemas that are harder to fill due to the higher number of constraints.

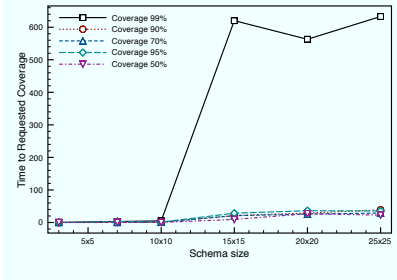
We measured the performance of the crossword compiler as the time needed to discover an $n\%$ filled solution, the percentage of times in which the solver finds a complete solution in a maximum of 10000 iterations and the percentage of completed boxes in the best solution discovered in 10000 iterations. For each selection of the parameters D and N, we averaged the results over 30 different randomly generated schemas. As shown in fig:6, the solver quickly converges to a good partial solution. However, going very close to a full solution becomes more complicated when the number of constraints is high (e.g. schemas bigger than 15×15 with few black boxes). As shown in fig:7-(b), the solver is very likely to discover a full solution for small crosswords or large ones with not too many constraints. When these conditions are not met, many runs can be needed before a full solution is found. However, as shown in fig:7-(a), even if a full solution is not found, the solver goes very close to it, discovering schemas with a very small non-filled portion.

[4]The independence assumption is incorrect since the terms correlate through the constraints. However, this assumption leads to precise estimates of the schema goodness, similarly to what happens for Naive Bayes classifiers in text categorization tasks.

[5]http://it.wikipedia.org.

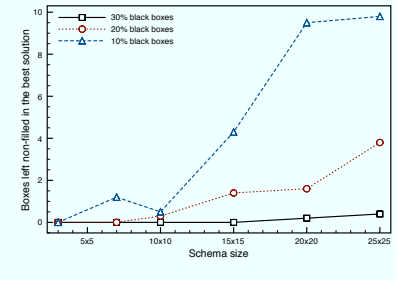


(a)

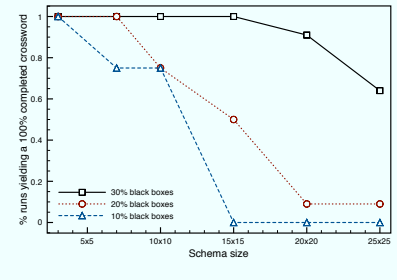


(b)

Figure 6. Average time needed to complete 95% boxes for each schema size for a crossword with 20% and 10% black boxes (respectively in (a) and (b)).



(a)



(b)

Figure 7. (a) Average number of non-filled boxes left on the best discovered solution. (b) Percentage of runs returnng a 100% completed crossword.

4. Conclusions

This paper presents a system that is able to automatically generate crosswords, including automatic definition generation. The proposed system crawls relevant portions of the Web, extracts the definitions from the downloaded pages using state-of-the-art NLP techniques and, finally, compiles a crossword layout using using a CSP solver. Further improvements to our definition extraction module will allow growing the number of available definitions and, thus, being able to fill larger crosswords. As a future development, we plan to estimate and control the difficulty of the generated crosswords to allow using this system in educational contexts: for example as teaching tool for children at different ages. Finally, we plan to classify each extracted definition into a predefined set of topics to be able to automatically generate single topic (or focused) crosswords.

References

[1] A. Aherne and C. Vogel. Wordnet enhanced automatic crossword generation. In *Proceedings of the 3rd International Wordnet Conference*, pages 139–145, Seogwipo, Korea, 2006.

[2] T. Brants. Tnt – a statistical part-of-speech tagger. In *Proceedings of the 6th Applied NLP Conference (ANLP)*, Seattle (WA), 2000.

[3] E. Brill. A simple rule-based part-of-speech tagger. In *Proceedings of the 3rd Conference on Applied Natural Language Processing (ANLP)*, pages 152–155, 1992.

[4] M. Ernandes and M. Gori. WebCrow: a WEB-based system for CROssWord solving. In *American Conference on Artificial Intelligence (AAAI-05)*, July 2005.

[5] J. Gimenez and L. Marquez. Fast and accurate part-of-speech tagging: The svm approach revisited. In *Proceedings of RANLP*, 2003.

[6] M. L. Ginsberg, M. Frank, M. P. Halpin, and M. C. Torrance. Search lessons learned from crossword puzzles. In *AAAI*, pages 210–215, 1990.

[7] G. A. Keim, N. M. Shazeer, M. L. Littman, S. Agarwal, C. M. Cheves, J. Fitzgerald, J. Grosland, F. Jiang, S. Pollard, and K. Weinmeister. PROVERB: The probabilistic cruciverbalist. In *AAAI/IAAI*, pages 710–717. JOHN WILEY, 1999.

[8] T. Kudo and Y. Matsumoto. Chunking with support vector machines. In *Proceedings of the NAACL-2001*, 2001.

[9] A. Ratnaparkhi. A maximum entropy model for part-of-speech tagging. In E. Brill and K. Church, editors, *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 133–142. Association for Computational Linguistics, Somerset, New Jersey, 1996.

[10] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[11] S. Thede and M. Harper. A second-order hidden markov model for part-of-speech tagging. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, pages 175–182, 1999.