Crossword Puzzle Generator

Darren Cheng
darren.l.cheng@dartmouth.edu

Nimit Dhulekar
nimit.s.dhulekar@dartmouth.edu

CS-44: Artificial Intelligence

13 March 2009

Table of Contents:

"Crossword puzzles...require of the solver both an extensive knowledge of language, history and popular culture, and a search over possible answers to find a set that fits in the grid. This dual task, of answering natural language questions requiring shallow, broad knowledge, and of searching for an optimal set of answers for the grid, makes these puzzles an interesting challenge for artificial intelligence."
- From PROVERB [1]

1. Abstract

A crossword is a word puzzle that normally takes the form of a square or rectangular grid of black and white squares. Generation of crossword puzzles by hand is a very tedious task, and automating this task through a computer program is also a complex task. This complication arises from the fact that most crosswords include not just extensive vocabulary but also phrases, slang language, abbreviations etc. The goal of this project was to use intelligent techniques to emulate a crossword *setter* or *constructor* and generate or “compile” crossword puzzles.

2. Introduction

Crossword puzzle generation is an NP-complete problem. It has been used by many researchers to test intelligent algorithms and heuristic techniques. The earliest work in the field was by Mazlack [2], who viewed it as a heuristic search problem. However with limited processing power and a small dictionary he was unable to leverage the power of the word-by-word instantiation approach (Section 3) and was relegated to using the letter-by-letter instantiation approach (Section 3). Other researchers such as Ginsberg et al [3], Meehan and Gray [4] followed suit using ever larger dictionaries and higher processing capabilities. The outcome of this research has been that it is now widely understood that Crossword puzzle generation is a fine example of a Constraint Satisfaction Problem. The variables consist of the empty grid positions to be filled up and the values are dictionary words and/or phrases that can be placed in these positions. The board itself presents constraints in the form of the size of the grid, the number of empty grid positions to be filled, the length of word patterns and the intersection between different “across” and “down” word patterns. An important consideration when filling any empty word pattern is that the word to be filled must be a valid English language word. This constraint means that a dictionary lookup is necessary.

3. Approaches to Generate Crossword Puzzles

There are 2 main approaches to generating crossword puzzles:

a) Letter-by-letter instantiation approach
b) Word-by-word instantiation approach

The letter-by-letter instantiation approach proceeds by repeatedly picking an empty square from the grid and instantiating it to some alphabet. The word-by-word instantiation approach is based on repeatedly picking empty word patterns from the grid and filling them up with valid English words and/or phrases. For the purpose of this project, the word-by-word instantiation approach was chosen as this requires lesser number of variables. This allows for faster processing with lesser number of backtracks on each word pattern.

4. Algorithm

The algorithm used to implement Crossword Puzzle Generation is a slight improvement on the algorithm described in Meehan and Gray [4].

The efficiency of the algorithm depends on three factors which need to be considered at every step along the execution:
a) Which uninstantiated variable (empty word pattern) to fill in?
b) Which word to fill into the uninstantiated variables?
c) What point to backtrack to?

The algorithm consists of 3 steps:

a) Fill Strategy –
The Fill Strategy maximises the number of choices for the remaining words on the grid, thus minimizing the number of backtracks to reach a solution.

As mentioned in the project update, there are 2 heuristics that can be applied.
i) Most Constrained / Fail First: checks how many matches each word pattern has and chooses the one that has the least number of matches
ii) Ratio: calculates the ratio of uninstantiated letters in the word pattern to the length of the word pattern

On initial implementation, it was soon discovered that a new heuristic combining both the heuristics would result in much better performance. Thus the “ratio-ed most constrained” heuristic was implemented. This heuristic first applies the ratio heuristic to all partially instantiated (containing at least one letter) word patterns on the grid. This heuristic usually returns multiple word patterns that can be filled. The next step is the application of the most-constrained heuristic which mostly selects a single word pattern to fill. The probability of 2 or more word patterns having the same ratio and also being equally constrained is very low and wasn't observed during testing at all. But in case such a scenario does arise, the first out of these patterns will be chosen for filling.

b) Pick Strategy –
The Pick Strategy chooses a word to fill a pattern with and maximizes the number of choices for the remaining patterns in the grid. The heuristic used was based on the 'first n' possible matches heuristic. Depending on the number of words that are present in the database of a particular sized word, n is varied so that it reflects the number of possible matches. We name this heuristic the "size dependent first n' heuristic. For example there are 1252 words of length 3 whereas there are 32263 words of length 9 in the database. Thus n is chosen to be 20 for 3 letter words and 100 for 9 letter words. Refer Table 1 for full data.

Length of word	Number of words
3	1252
4	4896
5	9782
6	17266
7	23587
8	29782
9	32263
10	30813
11	25961
12	20460
13	14922
14	9758
15	5921

Table 1: Different word lengths and their counts

c) Arc-consistency –
Arc consistency allows determining whether the grid is consistent or there are word patterns which cannot be instantiated with words. Arc-consistency prevents having to search the entire grid for a word pattern to backtrack to when some word pattern cannot be instantiated. To implement arc-consistency, a hash table was used which has as keys each element belonging to the "across" and "down" word patterns and as values a list consisting of intersecting word patterns.

5. Dictionary and Google Search

For the purpose of the project, the UK Advanced Cryptics Dictionary (UKACD) [5] was used. This is a word list compiled for the crossword community. It is a 234,885-word dictionary. The dictionary contains words, phrases, place names, abbreviations etc. One of the issues with the dictionary was that some of the words were duplicate i.e. they would appear in both lower-case and upper-case or with the first letter capitalized and the rest of the word in lower-case.

The dictionary was divided into subsets based on the length of the words and thus there were 13 subsets ranging from words of length 3 to length 15. For efficient retrieval of the words and also to run “regular expression” type queries (such as get all words of length seven with ‘A’ as the first letter and ‘T’ as the fourth letter) that would fetch words matching the particular expression, the subsets of the dictionary were stored in Oracle Database10g Express Edition. The database approach has been used by complex crossword solving systems such as WebCrow [6] and Proverb.

A novel approach was implemented for retrieving clues for the crossword. Initially we tried to use the dict service (dictionary) installed on UNIX systems. But it was soon realized that the UKACD was far vaster than the vocabulary of the dict service containing abbreviations, place names, phrases etc which were not found in the dict service. To overcome this problem, we downloaded the JSON [7] library which allows to query Google and returns the top result or the most pertaining result. There are of course shortcomings with this approach in the sense that the API used could only return the data present on the Search Results page of Google. Thus only an abstract is returned which may or may not be a significant enough clue to clearly figure out what the word is.

6. Implementation

The grid is initially seeded with the word pattern which can be instantiated with the longest possible word.

6.1 Data Structures

The data structures used are as follows:
a) Tile
The grid is represented as a 2 dimensional matrix of squares. For each square, the state (instantiated or uninstantiated), location, alphabet instantiated to, whether a cross-over point for intersecting word patterns, crossword clue number and applet display properties are stored.

b) Words
Each word pattern’s start and end grid positions are stored in this data structure.

c) openList
This list contains all the words retrieved from the database. When a word needs to be backtracked, it is deleted from this list.

d) InfoStructure (closedList)
closedList is of type InfoStructure. This data structure contains the size of the results for a particular query fired to instantiate a word pattern, the position of the first word (corresponding to the particular query) in the openList, whether the word pattern is an “across” or “down” pattern and its position in the “across” or “down” lists. When a word is deleted from the openList, the size of the closeList related to that word is decremented and all subsequent closedLists decrement their positions.

e) Clue
This data structure stores the word and the clue.

6.2 Major Components

a) BoardPanel
Implements the Fill and Pick Strategies and Arc-consistency. This is the main class which is used for implementing the algorithm. The word pattern which allows the longest word is chosen and is seeded. The “ratio-ed most-constrained” heuristic is applied to the partially instantiated word patterns and the best pattern is chosen to fill. The database is queried depending on the partial assignment of this pattern. The word list returned is stored in openList and the closedList is also initialized with the requisite information. If the grid becomes inconsistent with this assignment, the pattern is filled with the next word in the openList. If all subsequent words in the openList fail, then the grid is inconsistent and we backtrack to the last instantiated intersecting pattern. The algorithm runs till we have a complete consistent assignment for the entire grid.

b) CreateBoard
This class is used to create the board.

c) subsetDict
This class creates tables in the database and inserts records into them.

d) UpdateBoard
The UpdateBoard class contains all the methods for dealing with board updating including resetting the board, filling a word pattern with a word, removing a word and reading the word filled in a word pattern.

6.3 Instructions for using the code

a) For the code to run, Oracle 10g Database Express Edition must be installed. The download link is
http://www.oracle.com/technology/software/products/database/xe/index.html
An Internet connection is also required to run the code.
b) There are 2 JAR files included with the source code ojdbc.jar and json.jar. These JAR files should be stored in the directory where the library files for JRE are present. The code can only be compiled when these 2 JAR files are present.

c) The file which contains the words is “wordlist.txt”. Please change the location of this file in subsetDict.java before running. Run subsetDict.java which creates the tables in the database and loads the records into the table.
d) To run the main program, CrossWords.java needs to be run. An applet will pop up after the execution is complete. To retrieve the list of clues, please click on Get Clues button provided on the applet.

7. Results and Analysis

We tested our algorithm on the 15x15 crossword puzzle as shown in Figure 1.

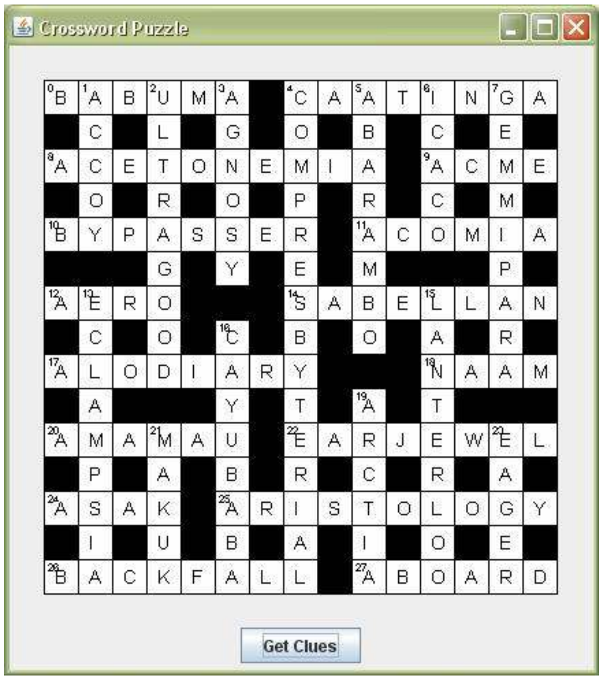


Figure 1

This crossword puzzle grid that we used is the standard 15x15 British-style grid. These grids have a lattice-like structure, with a higher percentage of black squares, leaving up to half the letters in an answer unchecked. These grids also have 180-degree rotational symmetry, so that the pattern appears the same even if turned upside down.

One of the significant issues we faced was “Maximum Open Cursors reached” exception thrown by Oracle when too many backtracks occur. When the number of backtracks is greater than 50, this error is usually returned by Oracle. This problem has a probability of 0.16. When this case arises, the board cannot be solved and it will be displayed in a partial assignment state as in Figure 2.

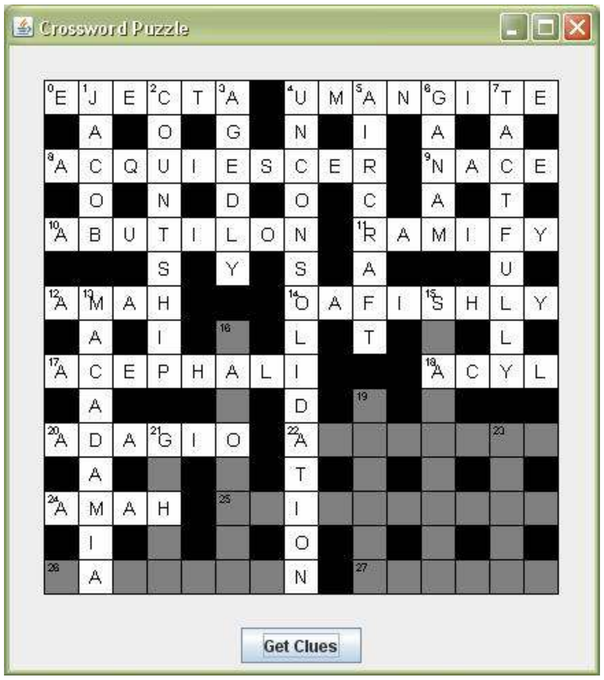
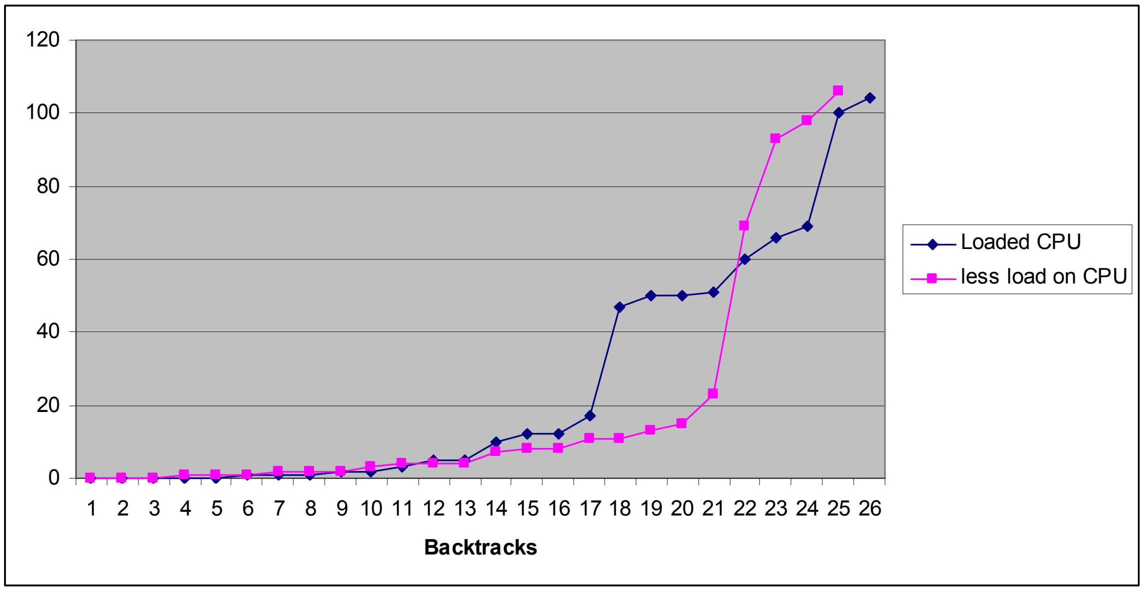


Figure 2

One of the observations we chanced upon was that if there are other applications running at the same time, there is a performance hit. It is our hypothesis that since Oracle requires high processing power but with other applications such as Firefox, Blitzmail etc running, it is unable to get enough CPU resources. Plot 1 displays test runs and the average values are displayed in Table 2. For run 1, we had multiple applications running and for run 2 we only had the code running.



Plot 1

Fail	Backtracks	Time (in ms)
9 out of 26 (loaded CPU)	25.69231	644.8462
4 out of 25 (less load on CPU)	19.44	631.68

Table 2

The Oracle exception caused our algorithm to be less reliable. But we find this to be an agreeable trade-off as compared to storing the wordlist in any of Java's data structures. A success metric that can be seen from the plot is that for most of our test cases the number of backtracks was less than 20. This is a good result for us and we consider it as an indication that the "ratio-ed most-constrained" heuristic and the "size dependent first n" heuristics that we have implemented are suitable candidates for optimal heuristics.

8. Conclusion

In this project implementation, we have shown that there are improvements possible with the Meehan and Gray algorithm. Although our algorithm cannot be considered completely reliable because of the problem of maximum open cursors with Oracle, we consider the use of a database to store our dictionary an important contribution of our work. We believe that the algorithm can be tweaked more by adding a pre-computed probability table for words. This would entail running the algorithm multiple times and calculating the probability of different words appearing in multiples test runs. Also as part of future work, we would like to test the algorithm on denser grids.

9. References

[1] Keim G., Shazeer N., Littman M. L., Agarwal S., Cheves C., Fitzgerald J., Grosland J., Jiang F., Pollard S., and Weinmeister K. *Proverb: The probabilistic cruciverbalist.* http://www.aaai.org/AITopics/pmwiki/pmwiki.php/AITopics/CrosswordPuzzles#prov
[2] Mazlack L. J. *Computer Construction of Crossword Puzzles Using Precedence Relationships*
[3] M. L. Ginsberg, M. Frank, M. P. Halpin, and M. C. Torrance. *Search Lessons Learned from Crossword Puzzles.*
[4] G. Meehan and P. Gray. *Constructing Crossword Grids: Use of Heuristics vs. Constraints.*
[5] Crossword Man – Word Lists. http://crosswordman.com/wordlist.html
[6] WebCrow: solving crosswords using the Web http://webcrow.dii.unisi.it/webpage/index.html
[7] JSON library downloaded from http://www.json.org/java/