REGIS FILLBIN

Sam Hage ‘16
Middlebury College Computer Science Department

The Problem

How do we get from a list of English clues to a solved crossword puzzle?

Crosswords present several interesting problems from a computer science perspective, because they comprise a few processes at which humans remain superior to computers. Regis Fillbin aims to surmount these problems and solve a New York Times crossword. The program works by generating lists of candidate answers for each clue, then searching for the configuration of answers that best satisfies all constraints (constraints being the intersecting answers for other clues).

I chose the New York Times crossword as the focus of the project because it is the best written and edited crossword, by general consensus among avid solvers. Consequently, the best crossword solving computer program should be able to solve the New York Times puzzle.

```
## MONDAY SEPTEMBER 5 2015 ##
# ACROSS
0       0       5       top of a wave
0       6       4       heed a red light
0       11      4       tanginess
1       0       5       do-it-yourselfer's book genre
1       6       4       norse deity with a hammer
1       11      4       part of the eye
2       0       5       chris who sang "Wicked Game," 1991
2       6       4       guthrie of Rising Son Records
2       11      4       word repeated before "pants on fire!"
3       0       13      showtime series named after an old fiction genre
4       3       3       proverbial madhouse
4       7       4       "when all ___ fails, read the instructions"
4       12      3       young-sounding wildebeest
5       0       4       spydon's ___ hari
5       5       4       ___-cola
5       10      5       cousins of ostriches
6       0       3       early afternoon hour
6       4       4       cheese off
```

Part of the program’s input

```
('frying fat', 'LARD')
('frying liquid', 'CONNOIL')
('frying liquid', 'OIL')
('frying medium', 'CONNOIL')
('frying medium', 'DEEPFAT')
('frying medium', 'LARD')
('frying mess', 'SPATTER')
('frying need', 'FAT')
('frying pan hazard', 'SPATTER')
('frying pan mishap', 'SPATTER')
('frying pan sound', 'SSS')
('frying pan spray', 'PAM')
('frying pan', 'GRIDDLE')
('frying pan,', 'SPIDERS')
('frying pans.', 'SPIDERS')
('frying sound', 'SSS')
('frying, for one', 'PAN')
('frying-butter sound', 'SSS')
('frying-pan coating', 'TEFLON')
('fryolator fill', 'DEEPFAT')
('fsu and nc state are in it', 'ACC')
('fsu or uf', 'SCH')
('ft. ~~~ (former military base near monterey, ca)', 'ORD')
('ft. ~~~: former california military base, near monterey', 'ORD')
('ft. above land', 'ALT')
```

A screenshot of the data used in solving the puzzles

The Data

Finding the right data sources was one of the most important aspects of this project, since my approach relies so heavily on searching through data. My main source is a corpus of close to five million clue-answer pairs, taken from years of previous crosswords. I wrote a web scraper to gather this data from crossword solving sites, which I then combined with an existing data set.

I performed a significant amount of preprocessing on this set, starting with sorting it alphabetically by clue. Because of the size of the collection, and because of the nature of crossword puzzles (certain clues appearing over and over), this resulted in many duplicate clues. I removed all entries that had both an identical clue and an identical answer. The next important step was partitioning the large set into six smaller ones by answer length. I did this to improve performance on my slower search algorithms. Other preprocessing included translating encoded characters to their corresponding plaintext, matching single and double quotes, and removing answers that were too short (under three letters) or too long (over 21 letters).

In addition to the main data set of clues and answers, I used what amounts to a crossword dictionary that I also scraped from the internet, and a list of all Wikipedia article titles. The Wikipedia data is useful because, while the dictionary only contains single word answers, Wikipedia titles provide multiple word answers, acronyms, geographic and historical information, popular culture, etc. I performed similar preprocessing on the Wikipedia titles, including partitioning them by length to increase performance.

Module 1:

___ Beta Kappa (3)” ➡ [“PHI”]

Module 2:

“___ Beta Kappa (3)” ➡ [“PHI”, “TAU”, “ETA”, “RHO”, “CHI”, “PSI”]

Module 3:

“P?I” ➡ [“PPI”, “PLI”, “PSI”, “PTI”, “PCI”, “PAI”, “PHI”, “PRI”, “PII”, “PEI”, “PJI”, “PKI”, “PNI”]

An example of how the three modules would run on a single clue. The first generates the fewest candidates, and the third the most, but the reliability of the answers decreases in the same direction.

1. Getting Answers

The first step in solving every puzzle is generating answer lists for each of the puzzle’s unencoded clues. This is the most important step, since none of the rest is possible without reliable answer candidates. It is also what most of my work programming was devoted to. Treating this step primarily as a data analysis problem, I developed three search ‘modules’, each taking a slightly different approach to generating potential answers.

Modified Binary Search

The first module is based on binary search, but incorporates Levenshtein Distance—a metric for determining the similarity of two strings—instead of exact matching, the usual criterion. Using binary search, I find the location the desired clue would occupy in the corpus. This gives me a small range of neighboring clues, whose edit distance to the desired clue is then computed. If any of these distances falls within a certain threshold, then their corresponding answer is added to the clue's candidate list.

The theory behind this approach is that across thousands of crosswords certain words, and therefore certain clues and answers—or their near variants—show up in common. These words, used for their advantageous vowel patterns or letter pairings, are commonly considered to constitue ‘crosswordese’, and we can expect their clues to appear with only minor changes in our data set. For early week puzzles, which rely more on common knowledge and simple phrases, and less on wordplay, ambiguity, and long answers like puzzles later in the week tend to do, this holds especially true.

This module is extremely fast, but has limited scope. The majority of clues in most puzzles will not match closely enough with any element from the data set to be considered reliable. What's more, this technique accounts for moderate wording changes, but not when the first word of the clue is changed, as the data is sorted alphabetically.

Fuzzy Search

The results of the first module are considered very reliable, so the second module is only applied to clues that had no matching answers from the first. Because performance is not an issue with binary search, the first module is run on the entire data set; the fuzzy search, however, runs in order linear time on *the number of words* in the data set. To compensate for this enormous hit, this module is run only on the set of clue-answer pairs whose answers are the same length as the answer needed for the desired clue. This was not done for the binary search module, because to search the partitioned data, each file must be passed around in memory. Since performance of the algorithm itself was fast enough, I chose to avoid this.

To perform the fuzzy search I identify the clue's key words by eliminating those perceived not to be important to the clue's meaning (a, and, of, to, the, etc.). This remaining set is then compared to the key words of each clue in the partitioned data using Python's regex library. If more than half of the key words , an answer is considered possible.

This module surmounts the first's problem of limited scope, but can overcompensate, producing too many false positives on short clues. If our sought clue has only one or two key words, for instance, then a very long clue in the data set has a good chance of also containing these key words, without actually being related in meaning.

Letter Matching

The final module discards the clue, focusing only on the length of the answer and the letters already known from crossing answers. These patterns are then compared with my crossword dictionary and list of Wikipedia article titles. Depending on the length of the answer and the portion of known letters this module can overcompensate even more than the fuzzy search, since most of the letters need to be known to significantly narrow the possibilities. This becomes an even bigger problem with short answers, when we consider the vast number of three and four letter acronyms with Wikipedia pages. Thus, even `P?I` generates nearly every possible letter combination, even with two of the three letters filled in, as in the example below. This module is also quite slow, and is likewise run on data partitioned by answer length.

2. Filling the Puzzle

Having generated a list of candidate answers for each clue, the next step is to seek the configuration that will result in the fewest conflicting word intersections. This is generally referred to as constraint satisfaction. In a constraint satisfaction problem, or CSP, where the solution space is relatively small and well understood, we can use techniques that essentially start with the assumption that one answer or configuration is correct, and then continue solving the problem under that assumption until a conflict is discovered, at which point the algorithm will backtrack. This works well for, say, sudoku, because each square must contain one of the numbers 0 through 9, and at some point we are guaranteed to find a solution. This is not the case with crossword puzzles, at least not with my program. In order for this to work, I would need to have a guarantee beforehand that each candidate list contains the correct answer, which is not the case.

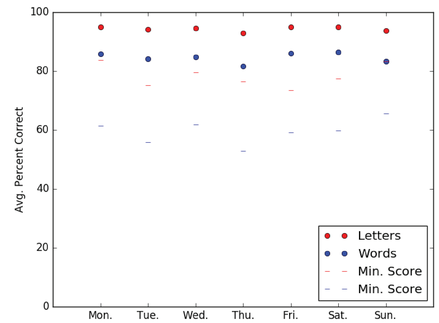
Instead, Regis Fillbin solves the puzzle the way a human solver would, that is, ranking the clues in terms of confidence in their answers, and, having filled some of them in, culling candidates of other clues based letters that have to be in certain positions. This allows for a greater degree of elimination among clues that need it the most (those with the longest answer lists), but it also means that an early mistake can have far-reaching consequences later in the puzzle. Even without relying on what I call the sudoku method, this program could incorporate some element of backtracking, namely by making multiple passes of the puzzle, identifying where perceived conflicts are, and then searching the original answer list to find a more suitable answer. I have not done this, mainly because I did not have time to expand the project so much, and also because I get good results without it.

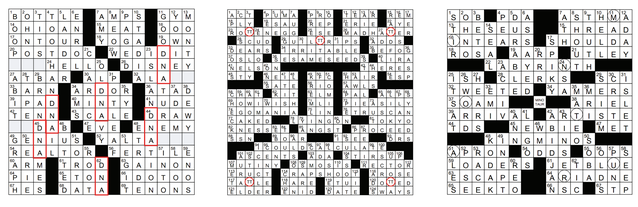
3. Results

My goal at the beginning of this project was to create an AI that was better at solving crosswords than humans. I knew I was unlikely to surpass the very best solvers, since they can solve every single New York Times puzzle fully correctly, every single day. But I was hoping to do better than an average person—at least on the early week puzzles—and solve any puzzle faster than any human. To test these goals I ran my program on 1,828 puzzles, evenly distributed across the days of the week.

As I had hoped, Regis Fillbin can solve Monday through Wednesday puzzles with an average of 94.54% of squares correctly filled, and 85.01% of correct answers. Bizarrely, I saw very consistent performance across all days of the week. In fact, Saturday, supposedly the most difficult puzzle, had the highest average rate of accuracy! I suspect the reason for this is that puzzles later in the week tend to have longer and longer answers, meaning that fuzzy search and pattern matching will work much more effectively than on the three- four- and five-letter answers that dominate early puzzles.

Timing was difficult to accurately determine, because I ran the tests on my own machine, and I often had three terminal processes running three batches of puzzles at once, slowing down execution to some extent. However, Regis seems to solve standard, 15x15 puzzles in around 1.25 minutes, and larger, 21x21 puzzles in the two minute range. This is far better than most, if not all, humans.





Three examples of rebus puzzles

Limitations and Future Work

Many crosswords have themes; related longer answers that appear throughout the grid and are generally tied together with a single “theme clue”. These themes almost always involve wordplay, intentional misspelling, or other humor; famously difficult problems for computers. Regis Fillbin’s natural language processing algorithms will fail to correctly interpret such wordplay.

Richtext, non-alphanumeric characters, and so-called “rebus” puzzles also exceed the program’s capabilities. These puzzles will have a special character, multiple characters, or even an entire word in a single square, generally requiring some level of human intelligence to interpret them correctly. Other puzzles will have a shape described in the black squares of the grid itself.

Clearly, there is room to expand Regis Fillbin’s capabilities in these areas, but some remain essentially impossible. More realistically, I could make large improvements in speed. Regis Fillbin is already faster (I think) than any human solver. Still, computers can usually perform tasks thousands or millions of times faster than people, so some improvement is definitely possible. Specifically, the slowness of the program comes from the size of my data. Based on my tests, if I could cut the data in half, I would see roughly a 25 percent speed up in solving times, which would bring many puzzles under one minute. Obviously there is also potential for improving Regis Fillbin’s accuracy. With more time, I would do this by implementing the backtracking described in the section on filling the puzzle

Puzzles Solved by My Program

Monday

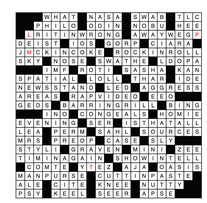
Tuesday

Wednesday

Thursday

Friday

Saturday



Sunday