REGIS FILLBIN: A CROSSWORD PUZZLE SOLVER

Sam Hage

Adviser: Professor Matthew Dickerson

A Thesis

Presented to the Faculty of the Computer Science Department

of Middlebury College

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Arts

May 2016

ABSTRACT

This paper describes a New York Times crossword puzzle solver, Regis FILLbin. The program uses several techniques to search and analyze data sources in order to generate lists of candidate answers for each clue. With these lists, the program determines the most likely possibilities from each candidate list and searches for the configuration of answers that best satisfies all constraints (constraints being the intersecting answers for other clues). Programming and research for this project began in September 2015 and were completed in the Spring of 2016. Regis FILLbin takes its name from Dr. Matt Ginsberg's 2011 paper and corresponding program, "DR.FILL" [2].

ACKNOWLEDGEMENTS

Many thanks to my advisor, Professor Dickerson, who was an incredible help despite claiming not to know anything about the problem I was undertaking; to Will Shortz for agreeing to let me use his data; to Matthew Ginsberg for almost introducing me to Will Shortz; to the many technical interviewers, who, after questioning me about my work, offered legitimate insights that I have shamelessly used to improve this project; to Daniel Scharstein, the indefatigable facilitator and mentor of our department; and to my father, who first kindled my interest in crossword puzzles by enduring years of me looking over his shoulder as he solved them.

TABLE OF CONTENTS

1 Introduction **1**
- 1.1 Problem Introduction and Relevance to Computer Science 1
- 1.2 Previous Work 3
- 1.3 What This Thesis Will Accomplish 4

2 General Techniques **5**
- 2.1 Data 5
- 2.2 Algorithms 7
  - 2.2.1 Binary Search 7
  - 2.2.2 Levenshtein Distance 7

3 Answer Generation **10**
- 3.1 Modified Binary Search 10
- 3.2 Fuzzy Search 11
- 3.3 Pattern Matching 13
- 3.4 Answer Weights and Overall Process 15

4 Solving the Puzzle **16**

5 Conclusions **18**
- 5.1 Results 18
- 5.2 Limitations 19
- 5.3 Future Work 20

A Solved Puzzles **26**

B Code **34**
- B.1 Filling the Puzzle Answers 34

Bibliography **40**

LIST OF FIGURES

1.1 Program’s plaintext input 2
1.2 Puzzle skeleton and plaintext output 3

2.1 Clue-answer pairs 6
2.3 Levenshtein distance table 8
2.2 Levenshtein Distance (pseudocode) 9

3.1 A sample of clues and their candidate answers 11
3.2 Modified Binary Search (pseudocode) 12
3.3 Fuzzy Search (pseudocode) 14
3.4 Three search modules in practice 15

5.1 A plot of my results 19
5.2 A rebus puzzle 21
5.3 A puzzle with a picture in its grid 22
5.4 Pi Day puzzle 23

A.1 Solved Monday puzzle 27
A.2 Solved Tuesday puzzle 28
A.3 Solved Wednesday puzzle 29
A.4 Solved Thursday puzzle 30
A.5 Solved Friday puzzle 31
A.6 Solved Saturday puzzle 32
A.7 Solved Sunday puzzle 33

CHAPTER 1

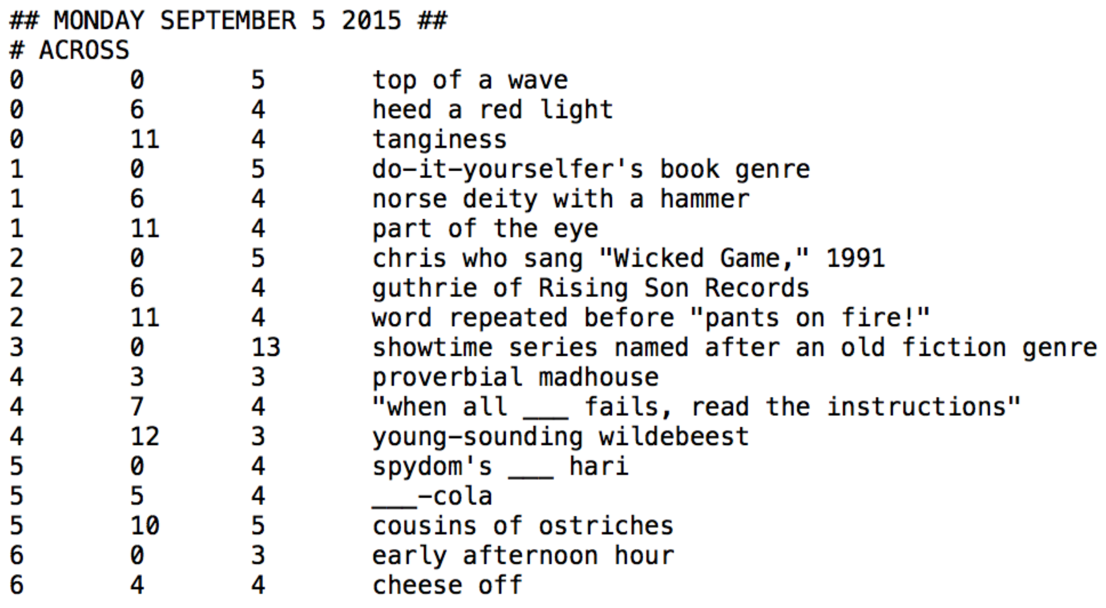
INTRODUCTION

This chapter will introduce the problem and its context in the field of computing, summarize previous work on this and similar problems, and conclude with a brief look at what this project intends to accomplish.

1.1 Problem Introduction and Relevance to Computer Science

Crossword puzzles present several interesting problems from a computer science perspective, because they comprise a few processes at which humans remain superior to computers (or at least draw on skills at which humans naturally excel). Each clue, for example, corresponds to a specific location on the puzzle's grid, which is not definitively described by the clue's single 'number' alone, but needs to be compared to the grid, a trivial task for the human eye, but one for which a computer needs more information (a representation of the puzzle grid). Nothing about the clue itself describes whether it runs 'across' or 'down'; this information comes simply from the header an indeterminate distance above the clue. Even the puzzle grid has a pattern of white (empty) and black squares, which determine the length of the clue's answer. These can generally be described as problems of computer vision, and are circumvented in this program by reading the puzzles in .puz format using a parser I wrote in Ruby to extract necessary plaintext information [Fig. 1.1].

But there are other 'human intelligence' tasks that the program must accomplish, starting with the generation of possible answers from each clue. This is the central feature of a crossword solver, and also the hardest to implement, given that a human solver is much better at interpreting a clue's nuances, understanding wordplay, detecting themes across a puzzle, and much else. Regis FILLbin uses a combination of natural language processing and intelligent data analysis for this piece of the problem. Essentially, it



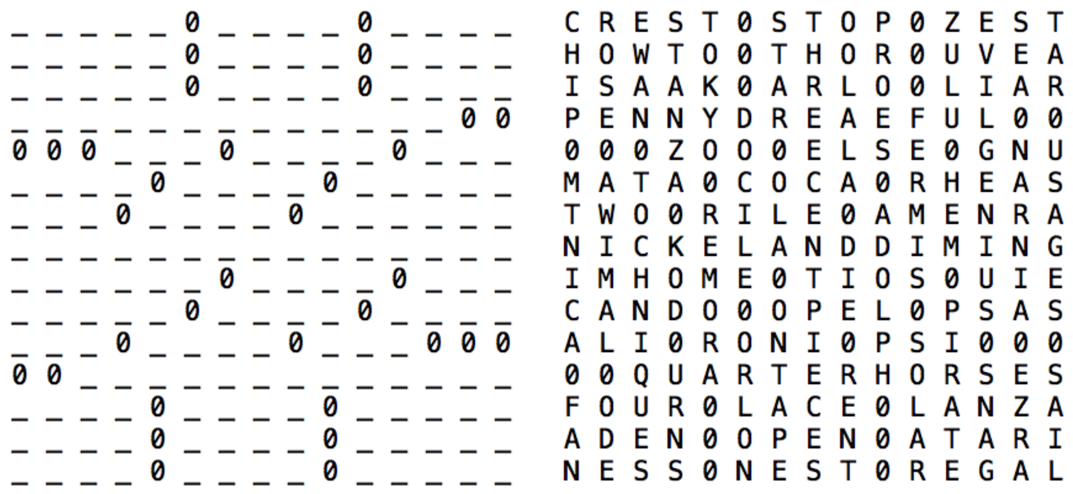
```
## MONDAY SEPTEMBER 5 2015 ##
# ACROSS
0       0       5       top of a wave
0       6       4       heed a red light
0       11      4       tanginess
1       0       5       do-it-yourselfer's book genre
1       6       4       norse deity with a hammer
1       11      4       part of the eye
2       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       spydom's ___ hari
5       5       4       ___-cola
5       10      5       cousins of ostriches
6       0       3       early afternoon hour
6       4       4       cheese off
```

Figure 1.1: The beginning of a text file given as input to the program. The columns represent the i, j indices in the grid, the length of the answer, and the clue itself, respectively.

searches a number of data sources containing potential answers, evaluating them on criteria like whether similar clues have been found in previous puzzles, which keywords are most important to the clue, and the matching of certain letter patterns.

The final step, in contrast, represents a problem at which computers still vastly exceed humans, due to their ability to perform billions of logic operations per second. The reconciling of candidate answer lists is essentially a constraint satisfaction problem, in which the program must try to combine thousands of potential answers in the best way. The approach will be outlined later in this paper.

I chose the New York Times puzzle 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. Sample puzzles are also more readily available than from other sources. Most of the coding was done in Python, with one JavaScript program that generates styled html to display solved puzzles, and a Ruby parser using a Ruby API from GitHub.



```
_ _ _ _ _ 0 _ _ _ _ 0 _ _ _ _     C R E S T 0 S T O P 0 Z E S T
_ _ _ _ _ 0 _ _ _ _ 0 _ _ _ _     H O W T O 0 T H O R 0 U V E A
_ _ _ _ _ 0 _ _ _ _ 0 _ _ _ _     I S A A K 0 A R L O 0 L I A R
_ _ _ _ _ _ _ _ _ _ _ _ _ 0 0     P E N N Y D R E A E F U L 0 0
0 0 0 _ _ _ 0 _ _ _ _ 0 _ _ _     0 0 0 Z O O 0 E L S E 0 G N U
_ _ _ _ 0 _ _ _ _ 0 _ _ _ _ _     M A T A 0 C O C A 0 R H E A S
_ _ _ 0 _ _ _ _ 0 _ _ _ _ _ _     T W O 0 R I L E 0 A M E N R A
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _     N I C K E L A N D D I M I N G
_ _ _ _ _ _ 0 _ _ _ _ 0 _ _ _     I M H O M E 0 T I O S 0 U I E
_ _ _ _ _ 0 _ _ _ _ 0 _ _ _ _     C A N D O 0 O P E L 0 P S A S
_ _ _ 0 _ _ _ _ 0 _ _ _ 0 0 0     A L I 0 R O N I 0 P S I 0 0 0
0 0 _ _ _ _ _ _ _ _ _ _ _ _ _     0 0 Q U A R T E R H O R S E S
_ _ _ _ 0 _ _ _ _ 0 _ _ _ _ _     F O U R 0 L A C E 0 L A N Z A
_ _ _ _ 0 _ _ _ _ 0 _ _ _ _ _     A D E N 0 O P E N 0 A T A R I
_ _ _ _ 0 _ _ _ _ 0 _ _ _ _ _     N E S S 0 N E S T 0 R E G A L
```

Figure 1.2: The plaintext skeleton (left) given to the program, and the puzzle output (right) after solving.

1.2 Previous Work

Regis FILLbin does not break new ground. Crossword solvers have been the focus of several artificial intelligence research projects since the 1990s. I will discuss two in particular that inspired and aided me in my own process.

Proverb: The Probabilistic Cruciverbalist[4][6][7] (1999) was an early crossword solver that took a probabilistic constraint satisfaction approach to the problem, meaning that likely answers are identified for each clue, and solving begins with the most probable answers. This is also my basic approach, and I owe much to the authors for general inspiration, including my use of the concepts of "modules" and "candidates". I suspected early on that I might not achieve *Proverb*'s impressive results (around 90 percent of words correct across all puzzles), but I did hope to beat its solving time of about 15 minutes.

A more recent attempt at the problem came from Dr. Matt Ginsberg in the form of DR. FILL[2], whose name my own program references. DR. FILL uses singly-weighted constraint satisfaction, recursively assigning values to spaces in the puzzle until the lowest-cost solution is found (costs being assessed by a complicated heuristic based on the maximum score given the choice of a certain answer). I got the idea of using Wikipedia article titles from this paper.

1.3 What This Thesis Will Accomplish

The ultimate goal of this project is to create a program that can solve crosswords better than a human can, by the two metrics of speed and accuracy. This means it should near completion on at least early week puzzles (the easier ones), and ideally perform more quickly than any human on any puzzle. I will start by providing an outline of my general approach and data sources, followed by an in-depth look at my algorithms, and finally the results of my program. I will conclude with a discussion of limitations and future direction for the project.

CHAPTER 2

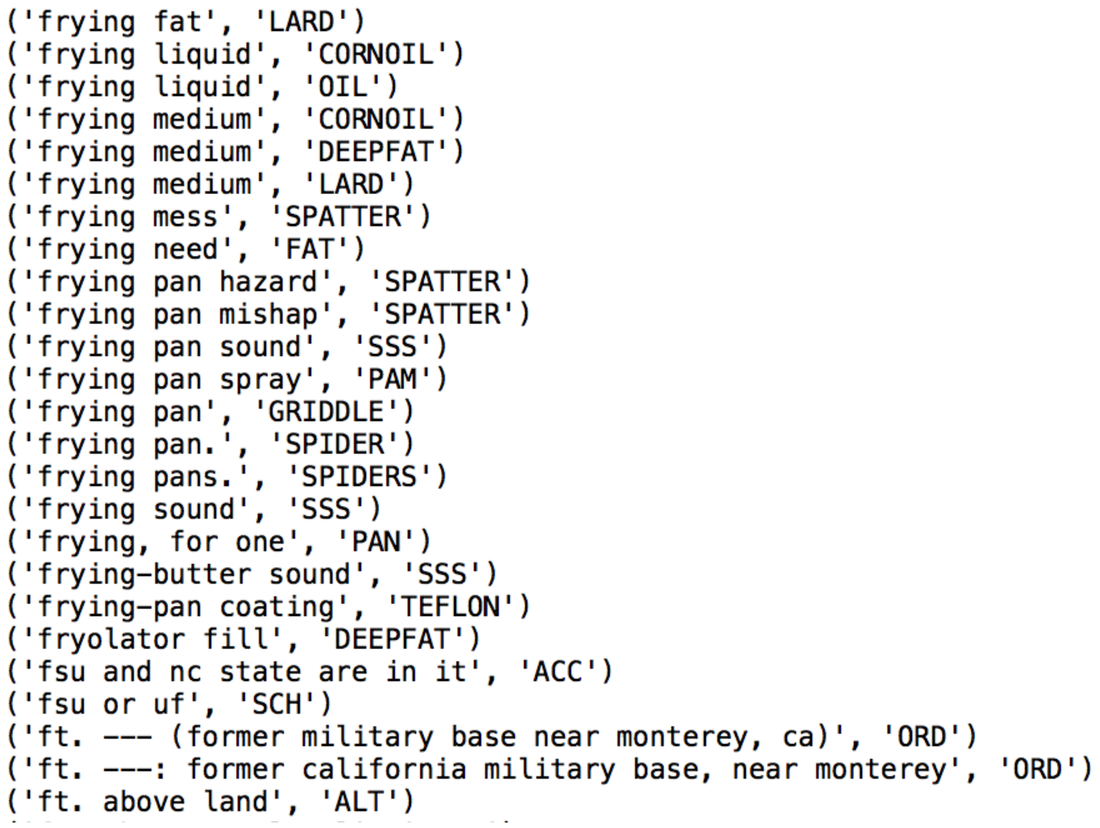
GENERAL TECHNIQUES

2.1 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 [Fig. 2.1]. I wrote a web scraper to gather this data from crossword solving sites, which I then combined with an existing data set. Each element in this set is a tuple of two strings: the lowercase clue followed by the answer. There are many repeated clues with different answers, and many repeated answers with different clues.

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.



```
('frying fat', 'LARD')
('frying liquid', 'CORNOIL')
('frying liquid', 'OIL')
('frying medium', 'CORNOIL')
('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.', 'SPIDER')
('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')
```

Figure 2.1: Some of the pairs in my data set. Each line is a single clue followed by its corresponding answer.

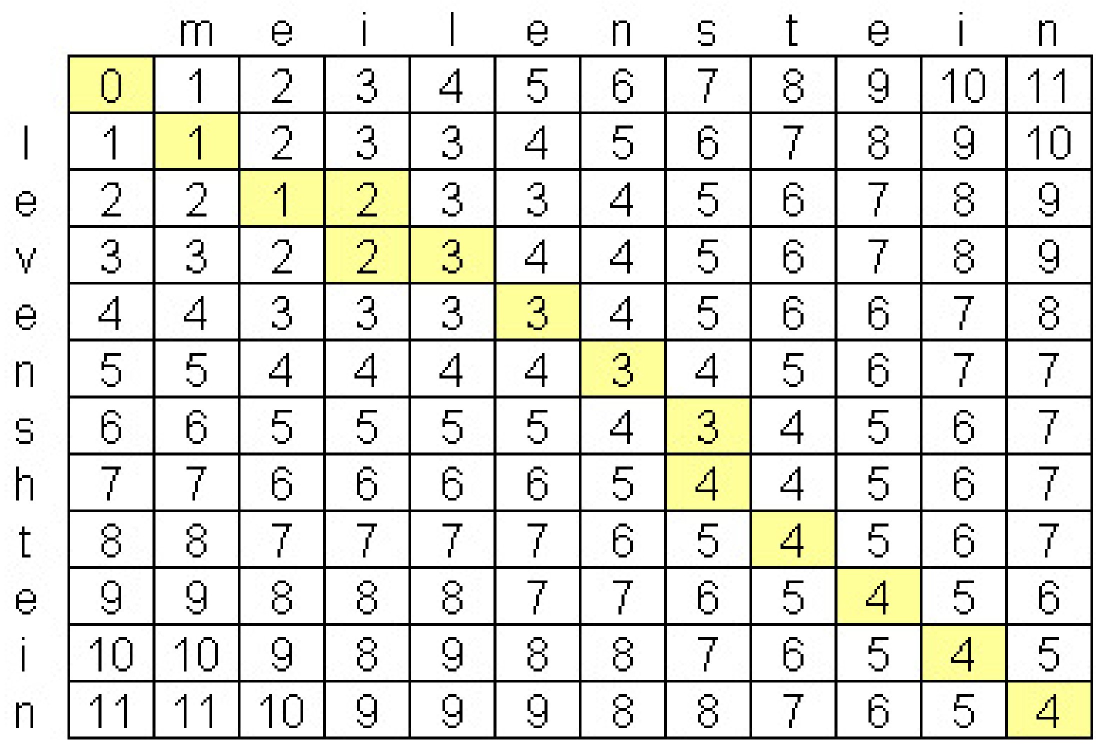
2.2 Algorithms

2.2.1 Binary Search

Binary search is a common divide and conquer algorithm used on sorted data for its simplicity and speed. The algorithm runs in $O(logn)$ time, where n is the number of elements, giving me acceptable performance even on my main data set of several million clue-answer pairs. I use this approach as an initial fast search that can be run on the entire data set [8].

2.2.2 Levenshtein Distance

The Levenshtein distance, also known as minimum edit distance, is a metric for comparing the similarity of two strings. It measures the minimum number of single character edits (insertion, deletion, and substitution) needed to transform one string to another. In general, this algorithm measures the superficial similarity of two strings, and is often used in autocorrect software. I have applied it as a sort of sentiment analysis—a way of determining if two crossword clues mean the same thing. The implementation for this project uses dynamic programming to build up a two-dimensional array, each element (i, j) of which gives the edit distance between the first string up to index i and the second string up to index j [Fig. 2.3]. This runs in $O(n + d^2)$ time, where n is the length of the longer string and d is the minimum edit distance. This is much faster than a recursive divide-and-conquer approach which would take exponential time. Technically this approach requires $O(m * n)$ space for strings of length m and n, respectively, but since I only care about the number of edits, and not actually recreating the transformation, I only need the last two rows of the array at any given step, which reduces the space complexity to $O(n)$ [1].



		m	e	i	l	e	n	s	t	e	i	n
	0	1	2	3	4	5	6	7	8	9	10	11
l	1	1	2	3	3	4	5	6	7	8	9	10
e	2	2	1	2	3	3	4	5	6	7	8	9
v	3	3	2	2	3	4	4	5	6	7	8	9
e	4	4	3	3	3	3	4	5	6	6	7	8
n	5	5	4	4	4	4	3	4	5	6	7	7
s	6	6	5	5	5	5	4	3	4	5	6	7
h	7	7	6	6	6	6	5	4	4	5	6	7
t	8	8	7	7	7	7	6	5	4	5	6	7
e	9	9	8	8	8	7	7	6	5	4	5	6
i	10	10	9	8	9	8	8	7	6	5	4	5
n	11	11	10	9	9	9	8	8	7	6	5	4

Figure 2.3: A visualization of the array giving the minimum edit distance at each pair of indices for the two strings. The final number, 4, in the bottom left, is the minimum number of edits to transform ‘meilenstein’ into ‘levenshtein’. The highlighted boxes represented the optimal sequence of edits [3].

Figure 2.2 Levenshtein Distance

Parameters:

s1, s2

```
procedure LEVENSHTEINDISTANCE
    $m \leftarrow$ s1.length
    $n \leftarrow$ s2.length
    for $i$ in 0..$m$ do
        v[$i$][0] $\leftarrow i$
    end for $i$
    for $j$ in 0..$n$ do
        v[0][$j$] $\leftarrow j$
    end for $j$
    for $i$ in 0..$m$ do
        for $j$ in 0..$n$ do
            if (s1[$i$-1] $==$ s2[$j$-1]) then
                v[$i$][$j$] $\leftarrow$ v[$i$-1][$j$-1]
            else
                v[$i$][$j$] $\leftarrow 1 +$ min( v[$i$][$j$-1], v[$i$-1][$j$], v[$i$-1][$j$-1] )
        end for $i$
    end for $j$
    return v[$m$][$n$]
```

CHAPTER 3

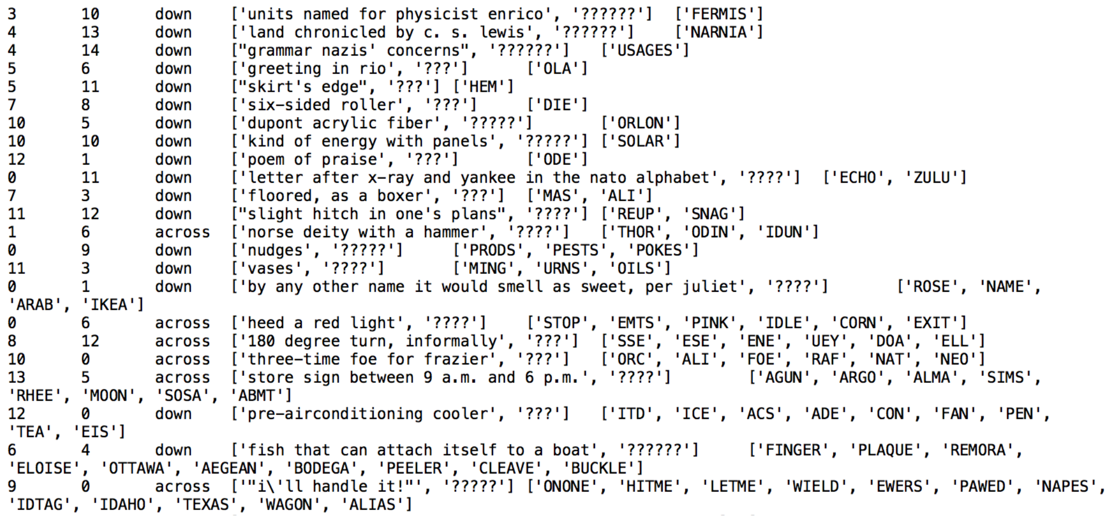
ANSWER GENERATION

The first step in solving every puzzle is generating answer lists from unencoded clues [Fig. 3.1]. 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 of which has the job of generating candidate answers for each clue. They go about this task in different ways, and at different points in the solving process.

3.1 Modified Binary Search

The first module is based on binary search, but instead of matching items exactly, it uses the Levenshtein distance to compare strings. Using binary search, I find the location the desired clue would occupy in the corpus (which, as described above, is sorted alphabetically by clue). 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, their corresponding answer is added to the clue's candidate list. I set this threshold experimentally at ten, which seems to be enough to account for substitution, addition, or removal of a few words, or minor differences in word order.

The theory behind this approach is that across thousands of crosswords certain words, and therefore certain clues—or their near variants—show up in common. These words, used for their advantageous vowel patterns or letter pairings, are commonly considered to constitute 'crosswordese', and we can expect their clues to appear with only minor changes in our data set. Thus by comparing a clue in the puzzle I'm trying to solve to a clue in my data set, I can infer whether or not the clues are likely to mean the same thing based on their edit distance. For early week puzzles, which rely more on common knowledge and simple phrases, and less on wordplay, ambiguity, and long answers like



```
3       10      down    ['units named for physicist enrico', '??????']  ['FERMIS']
4       13      down    ['land chronicled by c. s. lewis', '??????']    ['NARNIA']
4       14      down    ["grammar nazis' concerns", '??????']   ['USAGES']
5       6       down    ['greeting in rio', '???']      ['OLA']
5       11      down    ["skirt's edge", '???'] ['HEM']
7       8       down    ['six-sided roller', '???']     ['DIE']
10      5       down    ['dupont acrylic fiber', '?????']       ['ORLON']
10      10      down    ['kind of energy with panels', '?????'] ['SOLAR']
12      1       down    ['poem of praise', '???']       ['ODE']
0       11      down    ['letter after x-ray and yankee in the nato alphabet', '????']  ['ECHO', 'ZULU']
7       3       down    ['floored, as a boxer', '???']  ['MAS', 'ALI']
11      12      down    ["slight hitch in one's plans", '????'] ['REUP', 'SNAG']
1       6       across  ['norse deity with a hammer', '????']   ['THOR', 'ODIN', 'IDUN']
0       9       down    ['nudges', '?????']     ['PRODS', 'PESTS', 'POKES']
11      3       down    ['vases', '????']       ['MING', 'URNS', 'OILS']
0       1       down    ['by any other name it would smell as sweet, per juliet', '????']       ['ROSE', 'NAME',
'ARAB', 'IKEA']
0       6       across  ['heed a red light', '????']    ['STOP', 'EMTS', 'PINK', 'IDLE', 'CORN', 'EXIT']
8       12      across  ['180 degree turn, informally', '???']  ['SSE', 'ESE', 'ENE', 'UEY', 'DOA', 'ELL']
10      0       across  ['three-time foe for frazier', '???']   ['ORC', 'ALI', 'FOE', 'RAF', 'NAT', 'NEO']
13      5       across  ['store sign between 9 a.m. and 6 p.m.', '????']        ['AGUN', 'ARGO', 'ALMA', 'SIMS',
'RHEE', 'MOON', 'SOSA', 'ABMT']
12      0       down    ['pre-airconditioning cooler', '???']   ['ITD', 'ICE', 'ACS', 'ADE', 'CON', 'FAN', 'PEN',
'TEA', 'EIS']
6       4       down    ['fish that can attach itself to a boat', '??????']     ['FINGER', 'PLAQUE', 'REMORA',
'ELOISE', 'OTTAWA', 'AEGEAN', 'BODEGA', 'PEELER', 'CLEAVE', 'BUCKLE']
9       0       across  ['"i\'ll handle it!"', '?????'] ['ONONE', 'HITME', 'LETME', 'WIELD', 'EWERS', 'PAWED', 'NAPES',
'IDTAG', 'IDAHO', 'TEXAS', 'WAGON', 'ALIAS']
```

Figure 3.1: A sample of the file containing clues and answers. This file is updated as potential answers are eliminated. Clues with fewer possible answers are located toward the top, since the list of clues is sorted based on those with the greatest degree of certainty

puzzles later in the week tend to do, this holds especially true. If this first module generates a single match in the data set for a given clue, subsequent modules no longer need to be used on that clue, since I can say with a great deal of confidence that I have found exactly the right answer.

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. If all or nearly all clues had close matches in the data set, then puzzles could be solved almost instantaneously and with perfect accuracy. A further disadvantage is that although this technique accounts for moderate wording changes, it does not account for when the first word of the clue is changed, as the data is sorted alphabetically.

3.2 Fuzzy Search

The results of the first module are considered very reliable, so the second module is only applied to clues that had multiple matches or no matches from the first. Because performance is not an issue with binary search, the first module is run on the entire

Figure 3.2 Levenshtein Binary Search

Parameters:

clue,	
length	*length of answer*,
pairs	*set of clue answer pairs*,

```
procedure LEVENSHTEINBINARYSEARCH
    floor ← 0
    ceiling ← pairs.length - 1
    candidates ← [ ]
    while floor <= ceiling do
        middle ← (floor + ceiling) // 2
        curClue ← pairs[middle][0]
        curAnswer ← pairs[middle][1]
        if levenshteinDistance(curClue, clue) < MIN_DISTANCE then
            break
        else if curClue > clue then
            ceiling ← middle - 1
        else
            floor ← middle + 1
    i ← middle
    while levenshteinDistance(curClue, clue) < MIN_DISTANCE do
        curClue ← pairs[i][0]
        curAnswer ← pairs[i][1]
        if curAnswer.length == length then
            cadidates << curAnswer
        curAnswer ← pairs[++i]
    i ← middle
    curClue ← pairs[middle][0]
    while levenshteinDistance(curClue, clue) < MIN_DISTANCE do
        curClue ← pairs[i][0]
        curAnswer ← pairs[i][1]
        if curAnswer.length == length then
            cadidates << curAnswer
        curAnswer ← pairs[--i]
    return candidates
```

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 of key words is then compared to each clue in the partitioned data using Python's regex library. If more than half of the key words match, 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. Short clues are also a problem, since a threshold of one half does not translate well to small numbers of key words. A puzzle's clue, for instance, may have three 'key words', when in fact only one of these captures the meaning of the clue. This could result in too *few* accurate matches. This flaw is mainly due to my imperfect key word detection technique.

3.3 Pattern Matching

The final module discards the clue, focusing only on the length of the answer and the letters already known from crossing answers. It then compares these patterns 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

Figure 3.3 Fuzzy Search

Parameters:

clue,

length *length of answer*,

pairs *set of clue answer pairs*,

```
procedure fuzzySearch
    candidates ← [ ]
    keyWords ← splitOnWhitespace(clue)
    for word in keyWords do
        if NON_KEY_WORDS contains word then
            remove word from keyWords
    for pair in pairs do
        numMatches ← 0
        numMisses ← 0
        for word in keyWords do
            if pair contains word then
                numMatches += 1
            else
                numMisses += 1
            if numMissed > keyWords.length // 2 then
                break
        if numMatches / keyWords.length >= .5 then
            candidates << pair[1]
    return candidates
```

```
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"]
```

Figure 3.4: 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.

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 [Fig. 3.4]. Still, on long answers where a majority of the letters are already known, this module can be very effective, and I suspect was responsible for the good performance I saw on puzzles from later in the week. This module is also quite slow, running in linear time on the number of words in the data set, so it is likewise run on data partitioned by answer length.

3.4 Answer Weights and Overall Process

Combining the three modules, the solving process is as follows. Each clue is passed through the first module and any answers generated are added to the clue's candidate list. If applicable, the clue goes through the second module, after which it will have at least one answer in its list, with some very rare exceptions where neither of the first two modules generates any candidates. Before applying the third module at all, clues are evaluated based on the probability that the correct answer is known. Because I did not develop a more sophisticated heuristic, this amounts to how few answers their candidate lists have. The solving of the puzzle now begins, which method I detail in the next section. After answers can no longer be filled beyond a certain level of confidence, the third module is applied to unfilled answers and solving continues.

CHAPTER 4

SOLVING 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, change the answer configuration that it believes to be correct, and continue. 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, or solve the puzzle square by square instead of word by word. With just 180 squares (about the minimum allowed in a puzzle), doing constraint satisfaction by letter would leave 26^{180} possibilities, and unlike sudoku, it is not even possible to immediately eliminate letters based on simple constraints as in, say, having two 'A's in the same row. More simply, the number of possible configurations for a given crossword exceeds the number for a sudoku puzzle by more than 200 orders of magnitude.

Instead, Regis FILLbin solves the puzzle the way a human solver would, that is, starting with the most probable answers, and, having filled some of them in, eliminating candidates of other clues based on letters that have to be in certain positions. More specifically, I make a first pass at solving the puzzle, filling only 'singleton' answers, those whose clues have no other candidates. This first pass comprises entirely clues that required only the first module (since it is basically impossible that the second module will return just a single candidate), and thus have a high degree of accuracy. Once the singletons are filled, candidate lists of crossing clues are culled of their conflicting answers, generating more singletons and allowing the process to repeat. Previous answers are not overwritten when new ones are added, since answers chosen earlier are assumed to be more accurate than later ones. Instead, the letters are filled in the gaps as though the whole word were being filled in. This is continued until each iteration no longer updates the puzzle state. As mentioned in the preceding paragraph, not all candidate lists actually contain the right answer at all, so after the repeated culling, some lists are now empty. I now run the third module, which will generate additional candidates with greater or lesser accuracy, as we have seen. In some cases formerly empty candidate lists will now be singleton lists and their answers can be filled repeatedly, following the same technique as before. Most of the time, all singletons will have been filled, while the puzzle remains incomplete. At this point I select answers at random to fill the rest of the puzzle. This is where most of my errors come from.

The described technique allows for a great 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, but I get good results without it. This process is too cumbersome to describe in pseudocode, so I have included the source code in appendix B.1

CHAPTER 5
CONCLUSIONS

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 New York Times puzzles, evenly distributed across the days of the week. The puzzles were generously provided by Matt Ginsberg and Will Shortz. In some cases I removed puzzles because they involved some feature, usually of their themes, that Regis FILLbin cannot solve. For the most part, however, this test set was already curated for puzzle solving software with similar limitations to mine.

5.1 Results

Regis Fillbin can solve Monday through Wednesday puzzles with an average of 94.54% of squares correctly filled, and 85.01% of correct answers, meeting my expectations. The number of correct words is of course lower, because there may be up to two wrong words for every letter. 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 answers, meaning that fuzzy search and pattern matching will work much more effectively than on the three- four- and five-letter answers that dominate earlier puzzles. Friday and Saturday puzzles also tend to be themeless, meaning that their answers comprise mostly trivia, common phrases, and minimal wordplay.

Timing was difficult to determine definitively, because solving time varies so much from puzzle to puzzle. I was unable to accurately time a large number of puzzles because

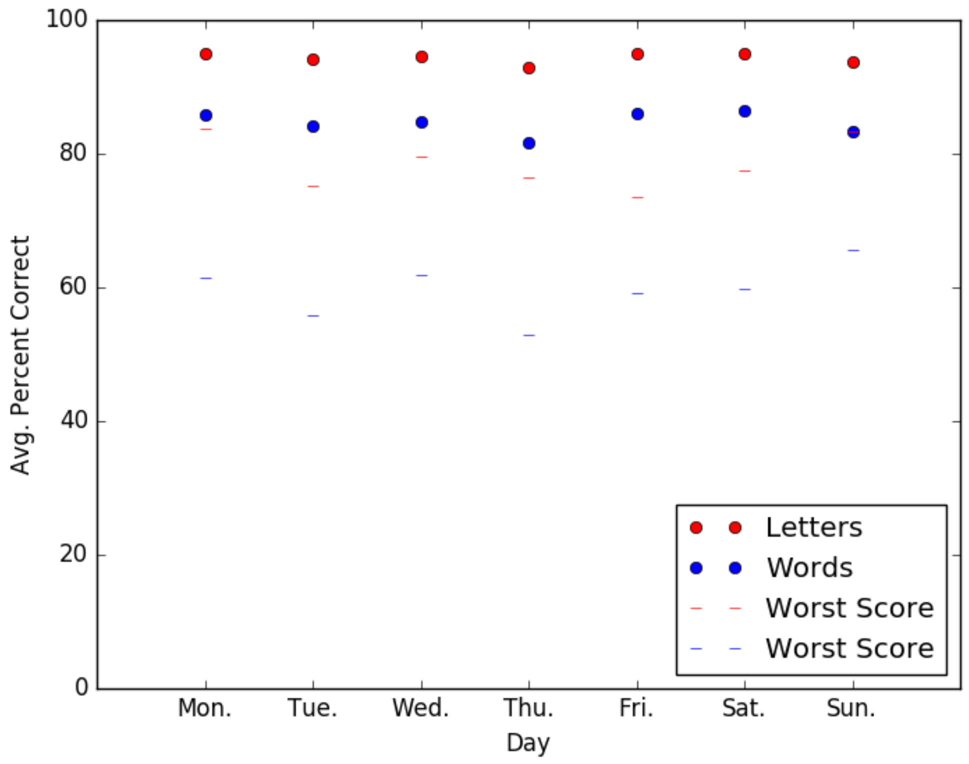


Figure 5.1: A plot of my results on the test data set. Red points indicate the percentage of letters correctly filled, blue dots indicate percentage of words. The dashes represent the lowest score on a puzzle of that day.

I ran the tests on my own machine, and in order to finish them all I often had three terminal processes running three batches of puzzles at once, slowing down execution to some extent. Based on tests of a few individual puzzles, 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.

5.2 Limitations

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.

Another major problem, though with a potentially simple solution, is that most puzzles contain some kind of self reference. For instance, 5-down may read, "With 14-across, lime green insect", while 14-across will read, "See 5-down". The main problem here is not the ambiguity of the clues on their own—although they are often too ambiguous to be helpful—but rather confusion in the data set. Since my data is drawn from crossword clue-answer pairs, it's possible that the binary search module will discover a close match and believe that it is a likely answer, when in reality the clue could have nothing to do with the content of the current puzzle beyond a semantic similarity in its referencing of another clue.

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, whose relation to the theme is left for the solver to decipher.

5.3 Future Work

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 believe) 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 the solving times

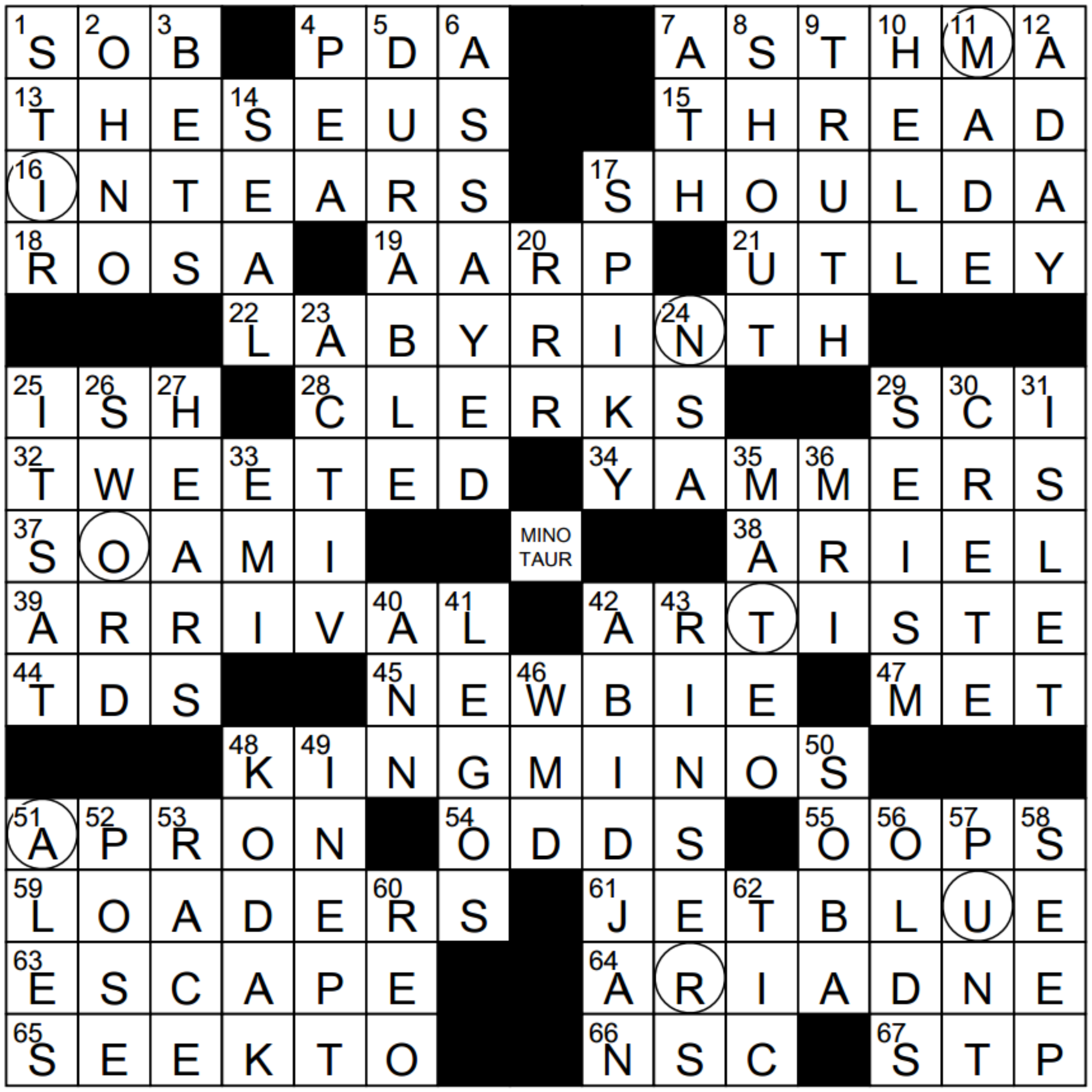


Figure 5.2: This puzzle's theme is Theseus and the Minotaur. Many of the clues reference the enclosed center square of the puzzle, which has to contain the entire word "minotaur". This is beyond the capabilities of my program, which allows only one letter per square.

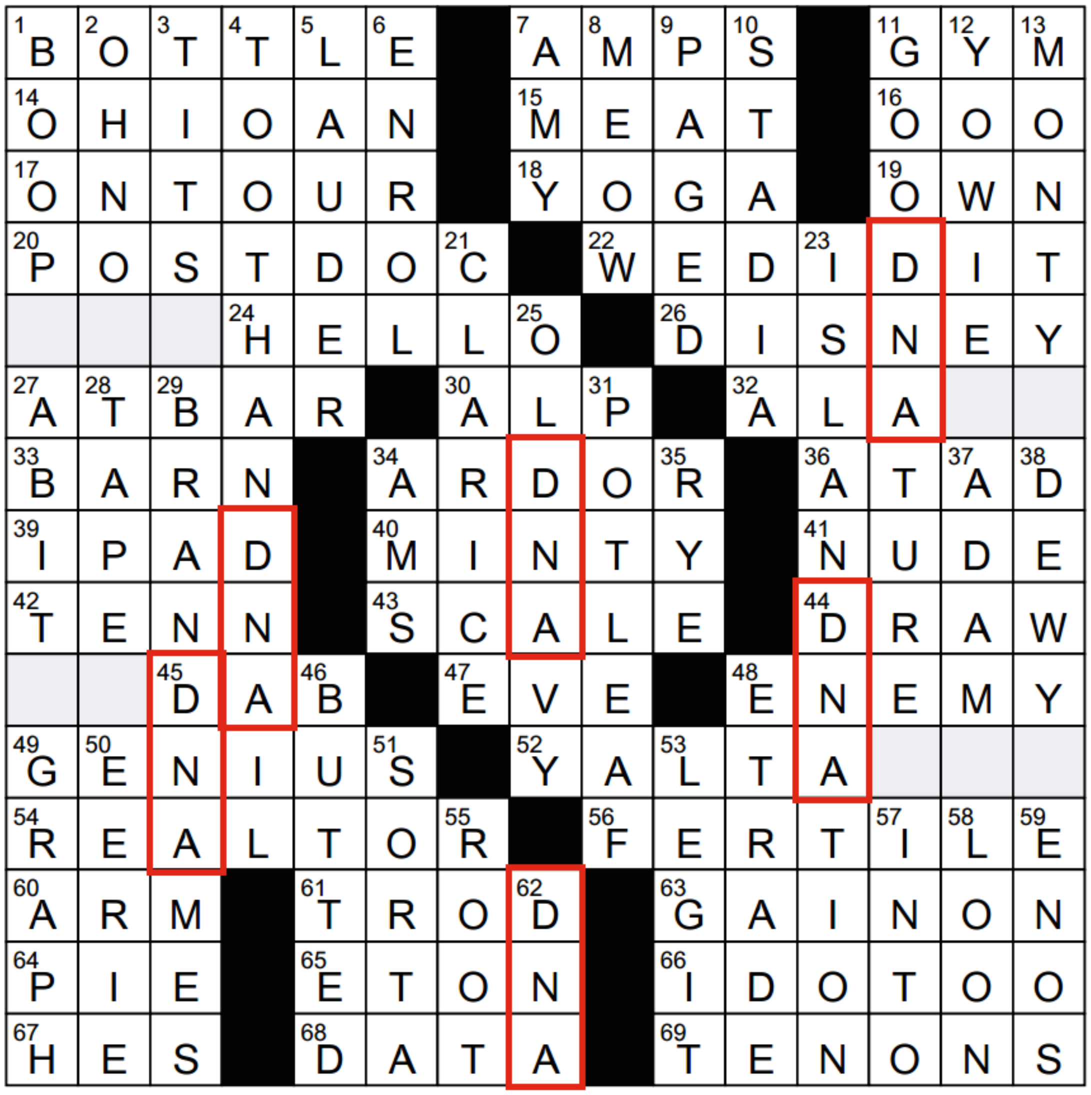


Figure 5.3: The black squares in this puzzle form a DNA helix, and "DNA" appears six times throughout the puzzle's down clues. This wouldn't be impossible for Regis FILLbin to solve, but it won't have any comprehension of the theme, making it harder than for a human.

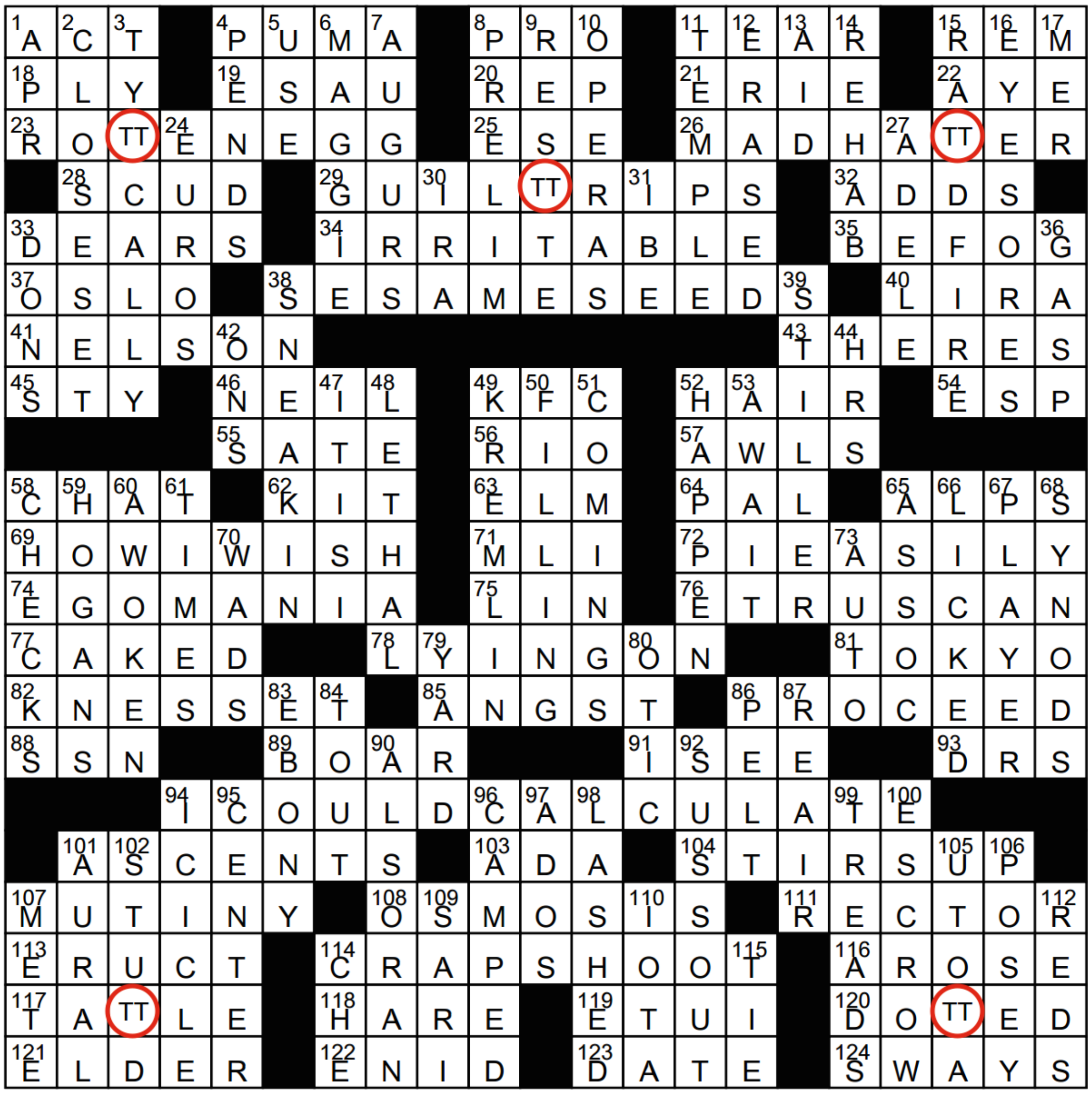


Figure 5.4: A Sunday puzzle done for Pi Day. A human solver is clued into the theme by the letter *pi* described by the middle squares, and will realize that some answers are too long to fit in the grid. This is because the letter *pi* can also be found five times in the theme answers, signifying a double ‘T’ reading across and a ‘PI’ reading down.

of many puzzles under one minute. As mentioned earlier, the binary search module is far faster than the other two, meaning that a higher rate of answer generation from the first module would also improve speed significantly. I imagine this could be achieved by tweaking its parameters or perhaps by reordering the clues of the data set and running the search again to increase the likelihood of getting a correct answer.

There is also potential for improving Regis Fillbin's accuracy. With more time, I would implement some or all of the following improvements:

- Implement the backtracking described in the section on filling the puzzle. Backtracking would allow Regis FILLbin to fill the grid fully, check for conflicts (since letters from previous answers are not replaced, potential conflicts could easily be identified by checking the consistency of the answer Regis FILLbin thinks it fills in and the one actually in the grid), and then return to problematic areas and reconsider the answers that led to what it identifies to be conflicts. More simply, it could find likely conflicts, and then at the very least indicate those as areas of uncertainty.
- Work out a better method of selecting key words from a clue. If done right, this would both reduce the number of false positives generated by the module, and correctly identify clues that it currently misses.
- Going beyond better answer selection, a more sophisticated method of answer weighting might reduce the frequency of early mistakes that without backtracking are difficult to fix.
- Don't just select answers at random from the answer lists after all singletons have already been eliminated. By looking at the individual letter combinations (say bi- and trigrams) that would result from using a certain candidate I could evaluate the likelihood of it being correct. Just looking at the frequency of these patterns in

words would not be entirely effective, as crossword answers can use abbreviations and multiple words, but it would be an improvement over random selection.

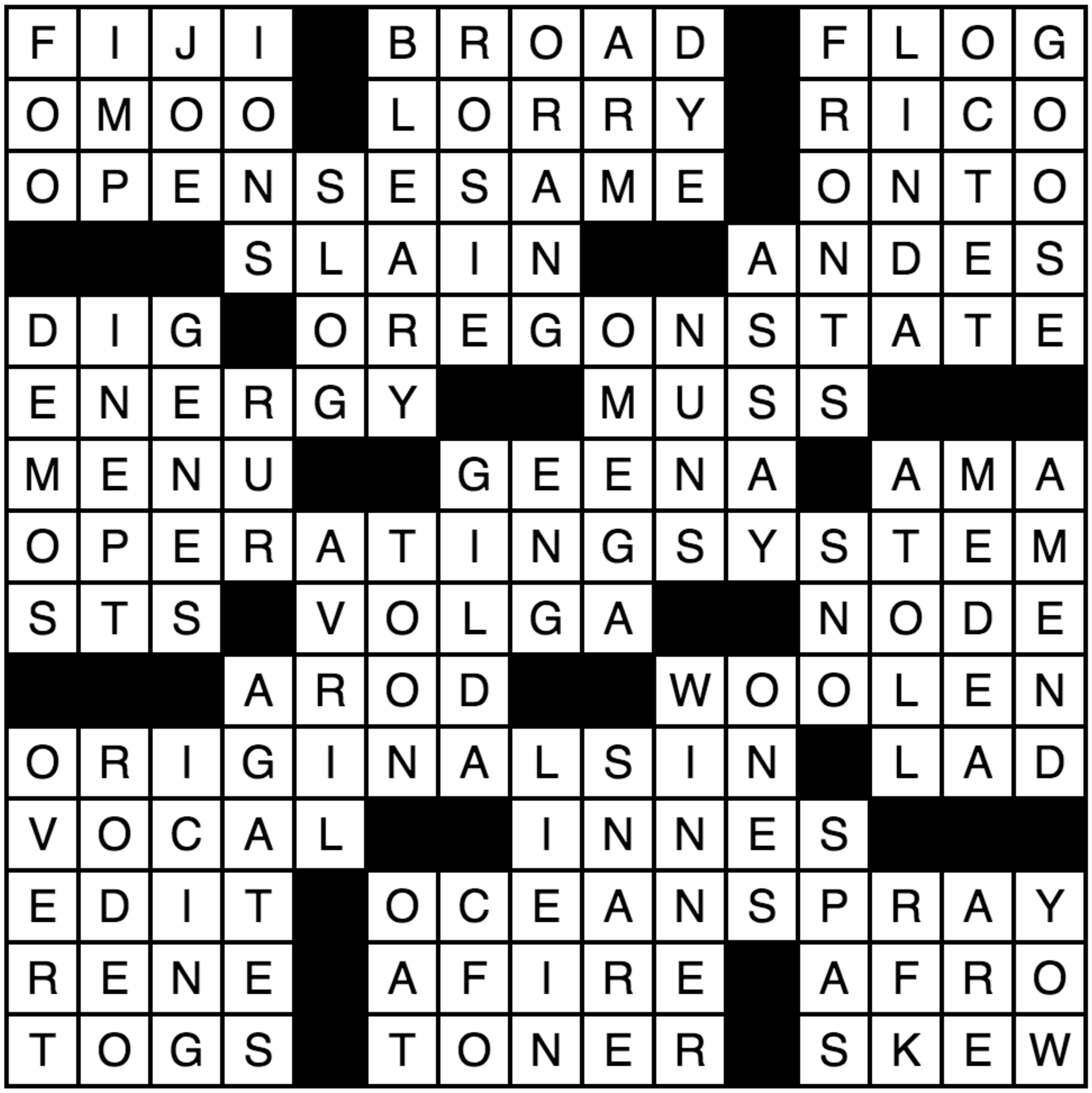
- Recognize when a clue references another clue in the puzzle, then look at that clue for indications about the answers.

Finally, it would be interesting to expand Regis FILLbin's capabilities into scans or pictures of real puzzles, incorporating grid and character recognition to bypass the need for .puz files. This could go as far as finishing a puzzle that had been partially solved by a human on paper, in the same way smartphone sudoku solvers can.

APPENDIX A

SOLVED PUZZLES

The following are seven puzzles solved by Regis FILLbin; a Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday puzzle, though not from the same week. Incorrectly guessed letters are in red.



F	I	J	I	■	B	R	O	A	D	■	F	L	O	G
O	M	O	O	■	L	O	R	R	Y	■	R	I	C	O
O	P	E	N	S	E	S	A	M	E	■	O	N	T	O
■	■	■	S	L	A	I	N	■	■	A	N	D	E	S
D	I	G	■	O	R	E	G	O	N	S	T	A	T	E
E	N	E	R	G	Y	■	■	M	U	S	S	■	■	■
M	E	N	U	■	■	G	E	E	N	A	■	A	M	A
O	P	E	R	A	T	I	N	G	S	Y	S	T	E	M
S	T	S	■	V	O	L	G	A	■	■	N	O	D	E
■	■	■	A	R	O	D	■	■	W	O	O	L	E	N
O	R	I	G	I	N	A	L	S	I	N	■	L	A	D
V	O	C	A	L	■	■	I	N	N	E	S	■	■	■
E	D	I	T	■	O	C	E	A	N	S	P	R	A	Y
R	E	N	E	■	A	F	I	R	E	■	A	F	R	O
T	O	G	S	■	T	O	N	E	R	■	S	K	E	W

Figure A.1: Monday.

H	U	N	T	■	G	L	E	E	■	■	A	T	A	D
O	R	E	O	■	E	A	R	T	H	■	N	A	M	E
B	A	R	R	Y	W	H	I	T	E	■	S	K	I	N
O	L	D	N	A	G	■	C	A	N	T	E	E	N	S
■	■	■	■	W	A	H	■	■	R	E	L	A	T	E
R	O	B	I	N	W	I	L	L	I	A	M	S	■	■
E	D	U	C	E	■	D	O	O	K	■	■	T	A	B
A	I	L	E	D	■	E	G	O	■	T	I	A	R	A
M	E	L	■	■	T	H	I	S	■	R	O	N	I	N
■	■	M	A	U	R	I	C	E	S	E	N	D	A	K
C	R	A	D	L	E	■	■	R	U	B	■	■	■	■
H	A	R	D	T	A	S	K	■	P	L	A	T	T	E
A	N	K	A	■	T	H	E	B	E	E	G	E	E	S
I	T	E	M	■	S	A	P	O	R	■	O	R	E	S
R	O	T	S	■	■	G	I	B	B	■	G	I	N	O

Figure A.2: Tuesday.

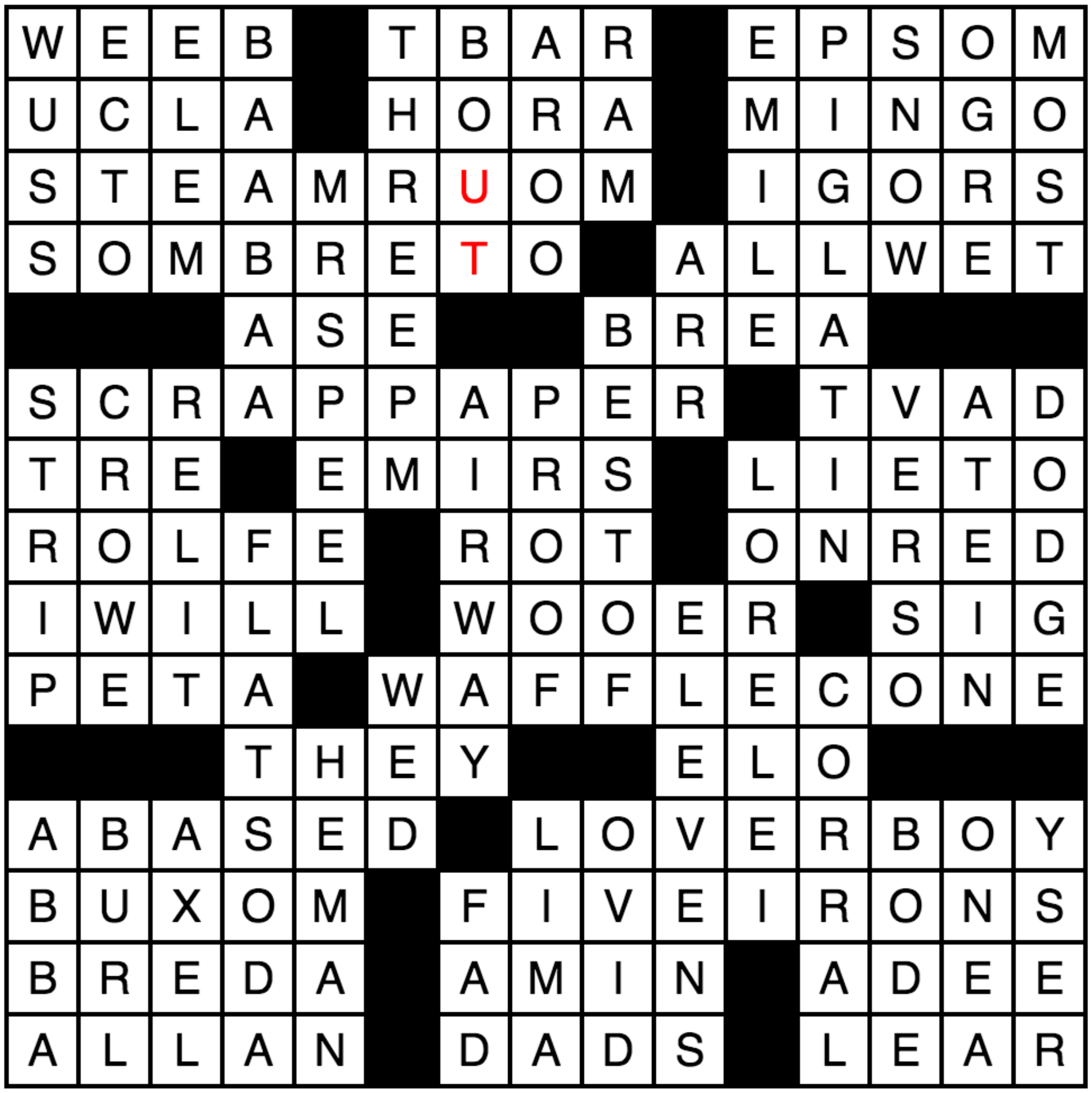


Figure A.3: Wednesday.

P	O	P	S	■	A	T	E	I	T	■	K	I	S	S
S	N	O	W	■	V	O	L	V	O	■	I	R	A	Q
H	A	L	F	N	E	L	S	O	N	■	S	E	A	U
A	H	L	■	E	N	D	E	R	■	A	M	A	R	E
W	O	O	D	C	U	T	■	■	A	M	E	L	I	E
■	P	I	E	C	E	O	F	W	R	I	T	I	N	G
■	■	■	L	O	Q	■	R	A	I	C	■	Z	E	E
A	S	H	E	■	■	P	O	X	■	■	N	E	N	E
S	P	A	■	B	R	O	Z	■	P	S	Y	■	■	■
P	A	R	T	I	A	L	E	C	L	I	P	S	E	■
I	M	D	O	N	E	■	■	H	A	N	D	A	X	E
R	A	C	K	S	■	B	O	O	Y	A	■	F	A	A
A	L	O	E	■	E	E	M	I	F	I	N	A	L	S
T	O	R	N	■	C	I	S	C	O	■	O	R	L	Y
E	T	E	S	■	I	N	K	E	R	■	H	I	Y	A

Figure A.4: Thursday.

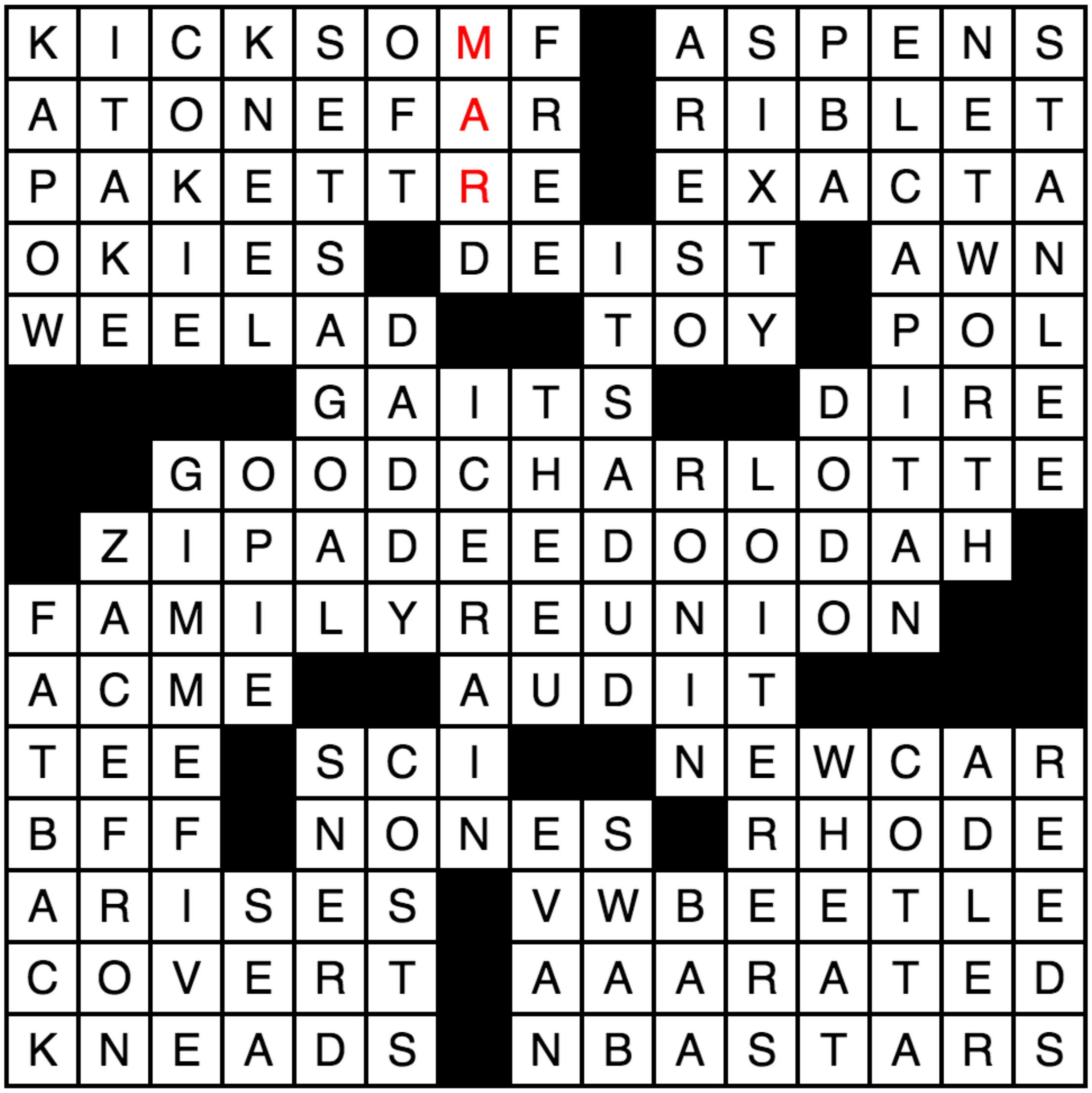
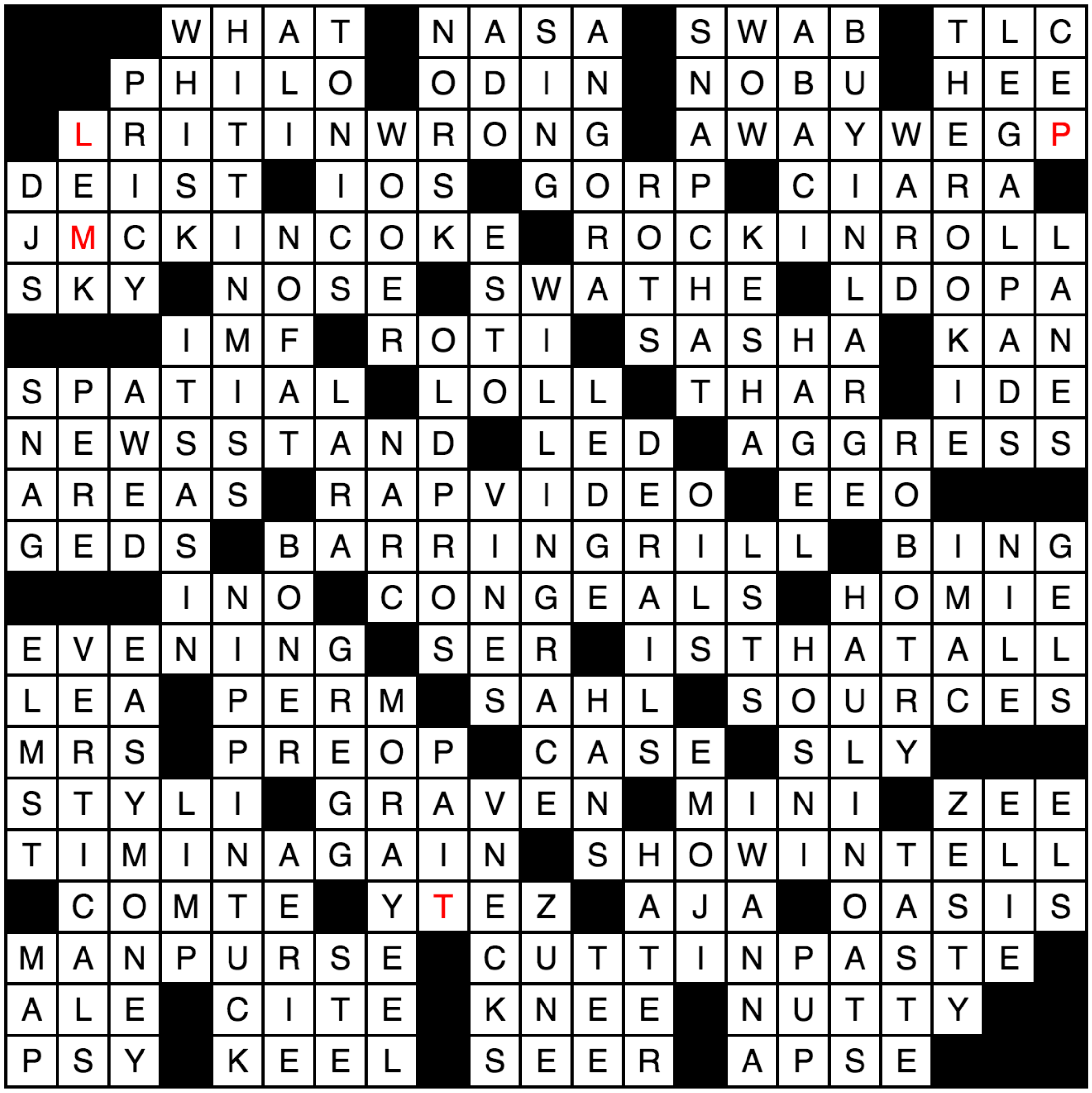


Figure A.5: Friday.

B	I	G	M	A	C	■	S	A	D	S	O	N	G	S
O	G	L	A	L	A	■	O	U	T	T	H	E	R	E
D	O	A	J	I	G	■	F	T	S	U	M	T	E	R
Y	T	D	■	T	E	N	T	O	■	N	I	T	T	I
C	O	S	I	■	S	E	T	H	S	■	C	L	E	F
A	R	O	A	R	■	L	O	A	T	H	■	E	L	S
S	I	M	I	A	N	■	P	R	O	A	M	■	■	■
T	O	E	S	O	O	P	■	P	O	L	E	M	I	C
■	■	■	S	U	M	A	C	■	D	O	S	I	D	O
O	R	S	■	L	A	P	A	Z	■	S	A	L	E	M
N	A	T	L	■	N	A	P	E	S	■	S	L	A	B
S	C	R	A	M	■	B	I	D	E	T	■	P	L	O
T	E	A	C	A	K	E	S	■	T	O	T	O	I	V
A	M	N	E	S	I	A	C	■	B	R	O	N	Z	E
R	E	D	S	C	A	R	E	■	Y	O	N	D	E	R

Figure A.6: Saturday.



■	■	■	W	H	A	T	■	N	A	S	A	■	S	W	A	B	■	T	L	C
■	■	P	H	I	L	O	■	O	D	I	N	■	N	O	B	U	■	H	E	E
■	L	R	I	T	I	N	W	R	O	N	G	■	A	W	A	Y	W	E	G	P
D	E	I	S	T	■	I	O	S	■	G	O	R	P	■	C	I	A	R	A	■
J	M	C	K	I	N	C	O	K	E	■	R	O	C	K	I	N	R	O	L	L
S	K	Y	■	N	O	S	E	■	S	W	A	T	H	E	■	L	D	O	P	A
■	■	■	I	M	F	■	R	O	T	I	■	S	A	S	H	A	■	K	A	N
S	P	A	T	I	A	L	■	L	O	L	L	■	T	H	A	R	■	I	D	E
N	E	W	S	S	T	A	N	D	■	L	E	D	■	A	G	G	R	E	S	S
A	R	E	A	S	■	R	A	P	V	I	D	E	O	■	E	E	O	■	■	■
G	E	D	S	■	B	A	R	R	I	N	G	R	I	L	L	■	B	I	N	G
■	■	■	I	N	O	■	C	O	N	G	E	A	L	S	■	H	O	M	I	E
E	V	E	N	I	N	G	■	S	E	R	■	I	S	T	H	A	T	A	L	L
L	E	A	■	P	E	R	M	■	S	A	H	L	■	S	O	U	R	C	E	S
M	R	S	■	P	R	E	O	P	■	C	A	S	E	■	S	L	Y	■	■	■
S	T	Y	L	I	■	G	R	A	V	E	N	■	M	I	N	I	■	Z	E	E
T	I	M	I	N	A	G	A	I	N	■	S	H	O	W	I	N	T	E	L	L
■	C	O	M	T	E	■	Y	T	E	Z	■	A	J	A	■	O	A	S	I	S
M	A	N	P	U	R	S	E	■	C	U	T	T	I	N	P	A	S	T	E	■
A	L	E	■	C	I	T	E	■	K	N	E	E	■	N	U	T	T	Y	■	■
P	S	Y	■	K	E	E	L	■	S	E	E	R	■	A	P	S	E	■	■	■

Figure A.7: Sunday.

APPENDIX B

CODE

B.1 Filling the Puzzle Answers

```
'''
Sam Hage
Thesis
Solves the puzzle
12/2015
'''

import ast
import sys
import time
import clue_scraper
import puzzle_structure

TIME_DELAY = .1
WORDS_FILE = 'assets/words-answers.txt'
WIKI_PATH_3 = 'assets/wiki-3.txt'
WIKI_PATH_4 = 'assets/wiki-4.txt'
WIKI_PATH_5 = 'assets/wiki-5.txt'
WIKI_PATH_6 = 'assets/wiki-6.txt'
WIKI_PATH_7 = 'assets/wiki-7+.txt'

def fill( puzzle_name ):
        """
        *****************************************************************
        This does all the work solving the puzzle, starting with reading the raw
        input, then generating all candidate answers and filling the grid

        @param: {string} puzzle_name
        """
        ## read the raw puzzle input ##
        puzzle_structure.read_raw_puzzle( puzzle_name )
        ## create a blank skeleton ##
        puzzle = puzzle_structure.create_puzzle( puzzle_name, 'skeleton' )
        ## scrape all the clues ##
        clue_scraper.lookup_all_clues( puzzle_name )

        ## ask for input to keep solving ##
        # print( 'Answers computed. Proceed with solving? ' )
        # cont = sys.stdin.readline()

        ## keep track of the puzzle at the previous iteration to control ##
        ## how long to solve at each step ##
        previous_puzzle_state = []

        puzzle_structure.sort_answers( puzzle_name )
        f = open( 'puzzles/' + puzzle_name + '/' + puzzle_name + '-answers.txt', 'r' )
        answers = f.read().splitlines()[3:]
```

```

	## extract answer data into list ##
	for i in range( len( answers ) ):
		answers[i] = answers[i].strip().split( '\t' )
		answers[i][0] = int( answers[i][0] )
		answers[i][1] = int( answers[i][1] )
		answers[i][3] = ast.literal_eval( answers[i][3] )
		answers[i][4] = ast.literal_eval( answers[i][4] )

	## fill singleton answers and update the candidates until this no longer ##
	## yields any change in the puzzle ##
	while puzzle != previous_puzzle_state:
		## update previous state ##
		previous_puzzle_state = [ row[:] for row in puzzle ]
		## fill all singletons, update possibilites, and resort ##
		answers, puzzle = fill_all_singletons( answers, puzzle, True )
		answers = update_candidates( answers, puzzle )
		answers = sorted( answers, cmp=puzzle_structure.compare_answers )
		answers = refactor_answers( answers )


	previous_puzzle_state = []
	## update answers with single word and wikipedia title searches ##
	## then fill singletons until this no longer yields any change ##
	while puzzle != previous_puzzle_state:
		## update previous state ##
		previous_puzzle_state = [ row[:] for row in puzzle ]
		answers = search_dictionaries( answers )
		answers = sorted( answers, cmp=puzzle_structure.compare_answers )
		answers = refactor_answers( answers )
		## fill the singletons ##
		answers, puzzle = fill_all_singletons( answers, puzzle, False )
		answers = update_candidates( answers, puzzle )


	## fill based on the first candidate. this is pretty arbitrary ##
	for answer in answers:
		if len( answer[4] ) > 0:
			puzzle = fill_answer( answer[4][0], puzzle, answer[0], answer[1], answer[2] )
		## clear console using escape sequence ##
		print( chr( 27 ) + '[2J' )
		## print the puzzle to stdout ##
		puzzle_structure.print_puzzle( puzzle )
		time.sleep( TIME_DELAY )

	## fill the rest of the squares with 'E'. this is extremely arbitrary ##
	puzzle = fill_empty_squares( puzzle )

	## write the final output for evaluation ##
	puzzle_structure.write_puzzle( puzzle_name, puzzle )


def fill_all_singletons( answers, puzzle, ignore_longs ):
	"""
	******************************************************************
```

```
        Fills the grid with all singleton answers, i.e. answers that are the only
        candidate. Remove filled answers once filled

        @param: {string[][]} answers The list of information about each clue
        @param: {string[][]} puzzle The current grid representation
        @param: {boolean} ignore_longs When true, ignore long answers, as they are
                                    more likely to have generated a false positive
        @return: {string[][], string[][]} The updated answers and puzzle state
        """
        to_remove = []
        for info in answers:

                ## clear console using escape sequence ##
                print( chr( 27 ) + '[2J' )
                ## print the puzzle to stdout ##
                puzzle_structure.print_puzzle( puzzle )
                time.sleep( TIME_DELAY )

                candidates = info[4]
                if len( candidates ) != 1:
                        break
                answer = candidates[0]
                ## ignore long answers ##
                if len( answer ) > 6 and ignore_longs:
                        continue
                row = info[0]
                col = info[1]
                direction = info[2]
                ## fill the answer ##
                fill_answer( answer, puzzle, row, col, direction )
                to_remove.append( info )

        ## remove all filled answers ##
        for info in to_remove:
                answers.remove( info )

        return answers, puzzle


def update_candidates( answers, puzzle ):
        """
        ******************************************************************
        Update the candidate answer lists for all clues based on patterns on the
        board. Also update the pattern for empty candidate lists so they can be
        searched in dictionaries later.

        @param: {string[][]} answers The list of information about each clue
        @param: {string[][]} puzzle The current grid representation
        @return: {string[][]} Updated answers
        """
        for i in range( len( answers ) ):
                ## get the current pattern in the grid ##
                row = answers[i][0]
                col = answers[i][1]
                direction = answers[i][2]
```

```
			pattern = ''
			for j in range( len( answers[i][3][1] ) ):
				pattern += puzzle[row][col]
				if direction == 'across':
					col += 1
				else:
					row += 1

			## check all candidates and remove conflicting ones ##
			updated_candidates = []
			for candidate in answers[i][4]:
				include = True
				for k in range( len( candidate ) ):
					if candidate[k] != pattern[k] and pattern[k] != ' ':
						include = False
						break
				if include:
					updated_candidates.append( candidate )

			answers[i][4] = updated_candidates

			## update the pattern ##
			answers[i][3][1] = pattern.replace( ' ', '?' )

	return answers


def search_dictionaries( answers ):
	"""
	*****************************************************************
	Search clues in word lists and wikipedia list

	@param: {string[][]} answers The list of information about each clue
	@return: {string[][]} Updated answers
	"""
	words = clue_scraper.load_clues( WORDS_FILE )
	wiki_3 = clue_scraper.load_clues( WIKI_PATH_3 )
	wiki_4 = clue_scraper.load_clues( WIKI_PATH_4 )
	wiki_5 = clue_scraper.load_clues( WIKI_PATH_5 )
	wiki_6 = clue_scraper.load_clues( WIKI_PATH_6 )
	# wiki_7 = clue_scraper.load_clues( WIKI_PATH_7 )
	wiki_titles = [ wiki_3, wiki_4, wiki_5, wiki_6 ]

	message = 'Searching dictionaries'
	j = 0
	for i in range( len( answers ) ):
		sys.stdout.write( '\r                                ' )
		sys.stdout.write( '\r' + message + ( j % 4 )*'.' + ( 5 - j % 4 )*' ' )
		sys.stdout.flush()
		j += 1
		pattern = answers[i][3][1]
		if len( answers[i][4] ) == 0:
			answers[i][4] = clue_scraper.single_word_match( pattern, words )
			# print( answers[i][4] )
			if len( answers[i][4] ) == 0 and len( pattern ) < 7:
```

```
                                # length = len( pattern )
                                # if length > 7:
                                #         length = 7
                                answers[i][4] = clue_scraper.wiki_title_match( pattern, \
                                    wiki_titles[ len( pattern ) - 3 ] )

        print( 'Done.' )
        answers = sorted( answers, cmp=puzzle_structure.compare_answers )
        num_answers = len( answers )

        i = 0
        print( num_answers )
        print( answers )
        for i in range( num_answers ):
                ## find first non-empty answer list ##
                if len( answers[i][4] ) != 0:
                        break

        ## move all empties to the back ##
        answers = answers[i:] + answers[:i]
        return answers


def refactor_answers( answers ):
        """
        *****************************************************************
        Move the clues with empty candidate lists to the end since after the sort
        they will be at the top.

        @param: {string[][]} answers The list of information about each clue
        @return: {string[][]} Updated answers
        """
        num_answers = len( answers )

        i = 0
        for i in range( num_answers ):
                ## find first non-empty answer list ##
                if len( answers[i][4] ) != 0:
                        break

        ## move all empties to the back ##
        answers = answers[i:] + answers[:i]
        return answers


def fill_empty_squares( puzzle ):
        """
        *****************************************************************
        Fill any remaining squares with 'E' since it's the most common letter

        @param: {string[][]} puzzle The current puzzle
        @param: {string[][]} The updated puzzle
        """
        height = len( puzzle )
        width = len( puzzle[0] )

        ## find empty squares ##
        for i in range ( height ):
                for j in range ( width ):
                        if puzzle[i][j] == ' ':
                                puzzle[i][j] = 'E'
                                ## clear console using escape sequence ##
                                print( chr( 27 ) + '[2J' )
                                ## print the puzzle to stdout ##
                                puzzle_structure.print_puzzle( puzzle )
                                time.sleep( TIME_DELAY )

        return puzzle


def fill_answer( answer, puzzle, row, col, direction ):
        """
        *****************************************************************
        Fill a single answer in the puzzle

        @param: {string[][]} answers The list of information about each clue
        @param: {string[][]} puzzle The current grid representation
        @param: {int} row The answer row
        @param: {int} col The answer column
        @param: {string} direction The direction of the clue
        @return: {string[][]} The updated puzzle
        """
        for i in range( len( answer ) ):
                if puzzle[row][col] == ' ':
                        puzzle[row][col] = answer[i]
                if direction == 'across':
                        col += 1
                else:
                        row += 1
        return puzzle
```

BIBLIOGRAPHY

[1] Hal Berghel and David Roach. An extension of ukkonen's enhanced dynamic programming asm algorithm. *ACM Trans. Inf. Syst.*, 14(1):94–106, January 1996.

[2] Matthew L. Ginsberg. Dr.fill: Crosswords and an implemented solver for singly weighted csps. *CoRR*, abs/1401.4597, 2014.

[3] Jason (http://stackoverflow.com/users/353137/jason). Levenshtein distance algorithm better than o(n*m)? http://stackoverflow.com/q/4057513/.

[4] Michael L. Littman, Greg A. Keim, and Noam Shazeer. A probabalistic approach to solving crossword puzzles. *Artificial Intelligence*, 134(1-2):23–55, 2002.

[5] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[6] Noam M. Shazeer, Michael L. Littman, and Greg A. Keim. Solving crossword puzzles as probabilistic constraint satisfaction. In *National Conference on Artificial Intelligence*, pages 156–162, 1999.

[7] Noam M. Shazeer, Michael L. Littman, and Greg A. Keim. Solving crosswords with proverb. In *National Conference on Artificial Intelligence*, pages 914–915, 1999.

[8] Eric W. Weisstein. Binary search. From MathWorld—A Wolfram Web Resource.