Solving Crossword Puzzles

Anna Brezhneva, Rahul Mitra, and Michael Stern

1. Introduction

Crossword puzzles have existed since the early 1900s and have only gained popularity since their inception. The most widely respected source of these puzzles is the New York Times, who have roughly five million people complete their puzzle every day. Crosswords also pose interesting artificial intelligence questions as solving one seemingly relies on very “human” reasoning. Matching an answer with any given clue requires difficult semantic analysis and often concerns slang, sounds, proper names, and references from popular culture. Often times the answer to a clue does not even remotely resemble an English word.

However, there are a few aspects of crossword puzzles that make them very appealing to try to solve with a computer. For one, they have a very clear problem definition. Obviously, each answer must have the same length as the number of squares associated with it and when words overlap, they must use the same letter in any given overlap. Beyond that, it is up to the solver to fill in the words such that they match the clues in the way that makes the most sense. However, as any avid solver knows, there are some more subtle rules such as abbreviating any answer that has an abbreviation in the clue or that if a word appears in the clue it cannot appear in the answer. There have also been thousands of puzzles published over the years, which means there would be no shortage of data from which to learn.

For these reasons, we attempted to make our own crossword solver. The solver is able to take any standard crossword as an input and the goal of the solver is to fill in all of the squares such that the clues match the answers in the best way possible, while the constraints detailed above are still satisfied. In order to get a sense for the nature of the problem we were tackling, we looked at a baseline analysis which simply used the CSP to generate possible answers. When using the full dataset, 7,200 possible answers, of which, the correct answer was among them. This also took a considerable amount of time. For an oracle, we had an experienced crossword solver complete the same puzzle, which he was able to complete in 21 minutes with 100% accuracy.

2. Approach

2a. Datasets

We used five data sets to train our algorithm. Note this is a significant increase from our initial use of two data sets The first is data from past NYT crosswords and contains 432205 clues, their associated clue answers, the year in which the crossword puzzle

was featured and the month. The data spans October 1996 through December 2012. After cleaning up the dataset, we get 292712 unique clues to words.[1]

The second dataset is a list of 109582 english words which are stored in our system as words to their length.[2]

The third dataset is a list of the 100 “crosswordiest” words. “Crosswordiest” is determined by a function that increases the “crosswordy” score if the word appears in crossword puzzles often and decreases the “crosswordy” score if the word appears in natural language often.[3] In other words, this is a set of words that it is very specific to crossword puzzles and are therefore, worth highlighting in our algorithm.

The fourth dataset is a list of 672 tupled homophones. That is each line in the text file associated with a combination of homophones includes anywhere from two to five homophones. A pair of homophones are two words that sound alike such as ‘bare’ and ‘bear.’[4]

The fifth dataset is a list of 1521 synonyms.[5]

Together, these words constituted the set of possible answers that an empty word could potentially be filled with. For more information about how this set is used, see sections 2b and 2c.

2b. Constructing the CSP

Though there are perhaps other ways of modeling a crossword puzzle, a very obvious way to approach solving is by casting the puzzles as a constraint satisfaction problem (CSP). As mentioned before, there are fairly obvious constraints in overlaps and word lengths and the potential variables naturally would include something relating to different word fills. Nearly all of the literature that we came across modeled the problem in this way and since it made intuitive sense to us to cast the puzzle as a CSP.

. After deciding to use the CSP, we researched ways in which to model the variables. As far as we could tell, there were two options. The first is a word-based approach, which models every potential answer as a clue. The advantage to this as

[1] https://github.com/donohoe/nyt-crossword
[2] http://www-01.sil.org/linguistics/wordlists/english/wordlist/wordsEn.txt
[3] http://noahveltman.com/crossword/?p=summary
[4] http://kol.coldfront.net/thekolwiki/index.php/File:Homophones.txt
[5] https://sites.google.com/site/kevinbouge/synonyms-lists

suggested by Steinthal is that it is more of a generic CSP problem and for such problems, there are several proven algorithms to exploits those formulations.[6]

The other approach is character-based, where every empty square in the puzzle is a variable. The advantage to this approach is that the variables themselves are very simple and each has a limited domain size of 26. The disadvantage is that it becomes harder to model the constraints and with any reasonably large grid, the small domain size will not help very much because there will be so many possible combinations.

In the previous literature on crossword puzzles, both techniques have been implemented and yielded reasonable results. However, the word-based approach has seen more success as seen in algorithms by Ginsberg and Connor et al.[7][8] Because of the previous success seen using the word-based approach and because we wanted to be able to leverage our knowledge about popular CSP algorithms, we decided to use a word based approach. After we decided the word based approach, designing features was fairly straight forward.

We constructed our CSP as follows: the variables in our CSP are the words that correspond to the fills in the crossword puzzle. Each fill has a length corresponding to it, and the domain of each variable is the set of all words that have the correct length, taken from our database of words. In the semantic analysis section, we discuss methods of limiting the domain such that only words of interest will be considered in the CSP computation.

After adding a variable for each word to be filled, we then compute the intersections between across fills and down fills, and add a binary potential for each one. Programmatically, the following constraint is added for each intersection between two words, across_str and down_str, and their intersecting indexes, across_intersecting_index, and down_intersecting_index respectively:

```
across_str[across_intersecting_index] == down_str[down_intersecting_index]
```

2c. Designing the algorithm

After designing the model, the next decision that we made was which algorithm to use. We considered designing our own algorithm from scratch, but decided to use an existing one because we wanted to leverage the fact that we had modeled our problem as a generic CSP and there is already a significant body of literature detailing these types of algorithms. Because of this, we decided to use the Backtracking algorithm, as provided to us by the CS221 libraries.

Upon trying to run our CSP algorithm using the english language database to fill our domains (wordsEn.txt), we found that although the CSP contained relatively few

---

[6] Steinthal, *Crossword Puzzles as Constraint Satisfaction Problems*
[7] Ginsberg, *Crosswords and an Implemented Solver for Singly-weighted CSPs*
[8] Connor et al., *Crossword Puzzles and Constraint Satisfaction*

variables (74 for a 15x15), the domains for each of our variables were incredible large - around 30,000 words for 4 letter words and 18,000 for 6 letter words. With such large domains, the algorithm seemed to stall and make no progress towards computing a solution. As such, we implemented heavy modifications to the existing CSP code, in order to leverage multi-processing to speed up the computation.

2d. Speeding up the algorithm using multi-processing

In order to speed up our computation, we first looked for the processing bottlenecks. We observed that the generating a binary potential between two variables was extremely computationally expensive, as it required constructing a 2D table of length `variable1_domain x variable2_domain`, then computing the binary potential value for each and every variable intersection in the 2D table. When finished, it would then follow the same steps outlined above, but with the variables switched. As such, this required an n^2 operation to occur twice, where *n* is the size of the domain for the given word. Considering that four letter words had up to 30,000 possible domain values, these operations were very computationally expensive, and on our computers, the algorithm seemed to be making no progress.

To more effectively execute our computations, we implemented massive multi-processing in our code by generating each of these tables on a separate CPU core. This was achieved by using the python multiprocessing module, which spawns a new process to be run on a new core. In this manner, we can achieve completely parallel processing at the core level, not simply threading. Our implementation was to simultaneously compute these large tables using parallel processing, and then update our CSP binary potentials table data (`self.binaryPotentials`).

After implementing the parallel processing, we experienced massive speed-ups. However, the computations were still too much for our laptops, when run on a 15x15 crossword puzzle. As such, we decided to use the most powerful cloud computer offed by Amazon EC2 web services: the c3.8xlarge “Compute Optimized” machine, which has 32 virtual CPU cores, 60 GiB of RAM, and 640GB of SSD storage.

We found that 15x15 crossword puzzles have approximately 180-200 intersections, thus requiring this many binary potentials. As such, we wrote a jobs scheduler that could spawn off processes to compute a binary potential on each core, and then spawn new processes once cores freed up. We found success in computing 28 binary potentials on each core, which would allow us to compute all of the binary potentials within 7 iterations. Figures 1 shows the command line output of our program, assigning 28 binary potential processes. Figure 2 show the resulting load on the CPU, with each of the 28 of the 32 cores being used.

```
Computing binary potentials  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27] in parallel out of 186 total
Process spawned. Generating binary potential tables of dimensions: 18320 x 18320 = 335622400 potentials...
Process spawned. Generating binary potential tables of dimensions: 18320 x 18320 = 335622400 potentials...
Process spawned. Generating binary potential tables of dimensions: 18320 x 18320 = 335622400 potentials...
Process spawned. Generating binary potential tables of dimensions: 18320 x 18320 = 335622400 potentials...
Process spawned. Generating binary potential tables of dimensions: 18320 x 18320 = 335622400 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 24642 x 24642 = 607228164 potentials...
Process spawned. Generating binary potential tables of dimensions: 24642 x 24642 = 607228164 potentials...
Process spawned. Generating binary potential tables of dimensions: 24642 x 24642 = 607228164 potentials...
Process spawned. Generating binary potential tables of dimensions: 24642 x 24642 = 607228164 potentials...
Process spawned. Generating binary potential tables of dimensions: 24642 x 24642 = 607228164 potentials...
Process spawned. Generating binary potential tables of dimensions: 24642 x 24642 = 607228164 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 29427 x 29427 = 865948329 potentials...
Process spawned. Generating binary potential tables of dimensions: 4916 x 4916 = 24167056 potentials...
```

Figure 1: command line output showing assigning 28 cores to compute binary potentials

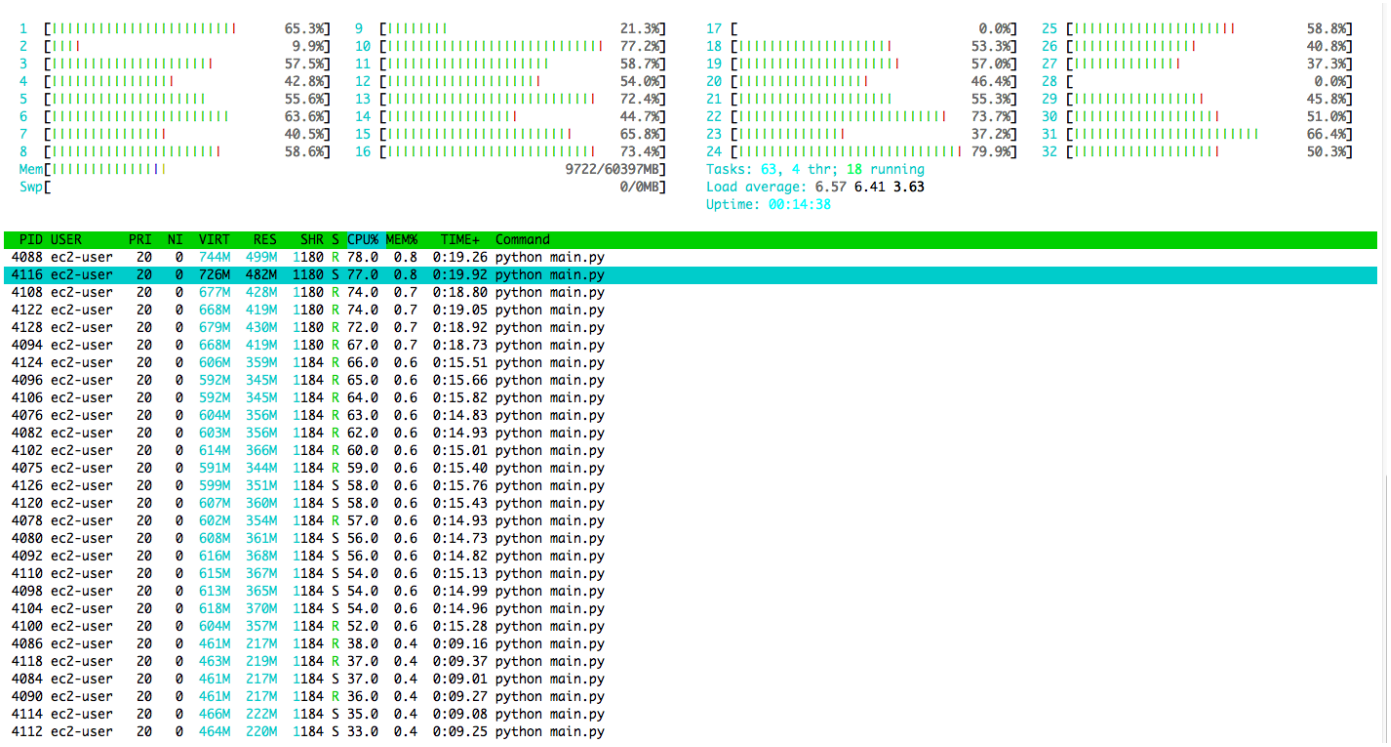

Figure 2: Amazon EC2 c3.8xlarge machine with 32 cores computing binary potentials in parallel (information ascertained by using the htop linux process viewer program)

Overall, adding multi-processing to our code allowed us to begin solving 15x15 CSPs, however, with thousands of variables in the domain, it would still take an entire day on the powerful Amazon EC2 machine to solve the CSP. As such, it became clear to us that solving CSPs with large domains does not scale well, and that simply

throwing more computational power at the problem would not be a fix As such, we turned to the use of semantic analysis to intelligently limit our domain, and ease the computational load. From running tests, we found that all word domains were limited to 2,000 words, we could solve the crossword puzzle within hours, and with 500 words, within five minutes. The challenge would be to limit these domains effectively, such that the correct solution still resides within the domain.

2d. Semantic Analysis

In order to narrow down the domain for each fill when adding each CSP variable, we performed basic semantic analysis on the clue, given its type and associated fill length, to come up with a list of most probable words. We then extended this list of most probable words by words of the english dictionary that satisfied the fill length. The list of most probable words was calculated as follows.

We first specify a list of 'indicator' words that might be present for each type of particular clue. The types of clues that we considered that could be triggered by such 'indicator' words are: reversal clues, hidden clues, palindrome clues, container clues, homophone clues, initialism clues, endalism clues, odd/even clues, deletion clues, foreign language clues, and anagram clues. These types of clues are defined as follows:

a. Reversal clue: The reverse of part of the clue provides the definition.
    Example clue: Sketcher went up to get reward
  Example answer: DRAWER

b. Hidden clue: The word is hidden within the letters of the wordplay.
    Example clue: Delia's pickle contains jelly
    Example answer: ASPIC

c. Palindrome clue: The fill will be a palindrome.
    Example clue: Advance in either direction
  Example answer: PUT UP. A palindrome meaning advance.

d. Container clue: Letters or words are placed inside other words.
    Example clue: Superman retains interest in Painter
    Example answer: TITIAN. TITAN = Superman. I = interest. Put I into TITAN gives the painter TITIAN.

e. Homophone clue: The fill will be a homophone of the potential answer
    Example clue: Not even one sister heard
  Example answer: NONE. Is a homophone of NUN (sister)

f. Initialism clue: First letter or letters provide the solution.
    Example clue: First class pile is really just tacky
    Example answer: CHEAP. C is first letter of 'class', followed by 'heap' which is synonym of pile

g. Endalism clue: Same as initialism clue, but last letters provide the solution.
h. Odd/Even clue: Take the odd or even letters to form the solution.
    Example clue: Observe odd characters in scene
    Example answer: SEE. Odd letters of SCENE
i. Deletion clue: Take letters out of clue to provide solution.
    Example clue: Parker loses a bed
  Example answer: COT. COAT = Parker. COAT loses 'A' for COT (a bed)
j. Foreign language clue: Basic Latin-based languages are sometimes used in clues.
    Example clue: Man on the Spanish foot
    Example answer: HEEL. HE = man. EL = the in Spanish. HE + EL = HEEL (foot).
k. Anagram clue: Indicated by potentially hundreds of words that loosely mean modify or change.
    Example clue: Dress suiting a saint
    Example answer: IGNATIUS. "dress" indicates anagram. letters of 'a suiting' provide IGNATIUS (a saint)

Of the clues stated above, reversal, hidden, homophone, initialism, endalism, odd/even, and deletion were implemented. Generating anagrams was too computationally expensive, and we did not find a good source for palindromes. Through research, we found other potential ways for analyzing clues if further work was done on generating probable words given a certain clue. This additional analysis included charade clues (formed by joining individual clues together to create the solution), double definition clues (instead of wordplay clues can have two definitions side by side to give the same solution), visual clues (occasionally objects that look like letters may be used in a clue), and combination clues (clues with more than one type of word play).

In addition, following semantic analysis of the clue itself, we trained our semantic analysis on a list of the "crosswordiest" words. Therefore, if we were to get a particular fill length, we would most often return words from this list that satisfy the given fill length. Additionally, provided the synonyms file, we were able to generate synonyms of words present in the clue as we saw this as a common trend in puzzles.

After returning the list of most probable words, in order to allow the csp variable to have a larger domain of words than what was returned by the semantic analysis algorithm, we also extend the list by english words that satisfy the fill length. After the csp returns several optimal assignments, we assess which solution is most likely the correct solution given another type of semantic analysis.

2e: Results from Semantic Analysis

If we just analyze how well the semantic analysis portion performs on its own, given a certain set of cryptic crossword clues, we see the following statistics, generated using the cryptic-clues-testData.txt file.

Total Number of Clues	**Number of clues with correctly limited domains**	**Percentage Accuracy**
5267	194	3.6%
4835	178	3.7%

Out of 5267 across clues, in 94 cases we had the correct answer be present in the list of most probable words returned by semantic analysis. Out of 4835 down clues, we had 78 cases where the list of most probable words contained the right answer to the clue. Although the accuracy is low, this shows that our semantic analysis serves as a good basis for analyzing cryptic crossword puzzle clues. The cases in which the right answer is not present in the list of most probable words, we likely find the right answer by extending the list of most probable words with words in the english dictionary, as well as answers we have been seen for other crosswords.

After the CSP is solved, we find the optimal solution (semantically) by doing a second round of semantic analysis. In order to do this, we assigned every value in the assignment a score. Two features contributed to the score: semantic analysis and word frequency. Note that it would have been possible to add more features, but with limited time and resources, we thought these two would be enough to modify the domains in significant ways and it turned out that this was the case.

In order to obtain the semantic portion of the score, we first performed analysis on every clue/answer pair. Recall that each value in the domain represents a possible answer. Our intuition was evaluate every answer based on the word in the clues that lead to that answer in previous puzzles. Based on this, we mapped every answer to words that had appeared in associated clues in previous puzzles. For example, if the answer in question was “BAT” and there were two clues, “HALLOWEEN ANIMAL” and “BLIND ANIMAL” that lead to “BAT” in previous puzzles, the value associated with the key “BAT” would be the set: {ANIMAL, BLIND, BAT}. The semantic analysis portion of the score came from seeing how many words in the clue in question showed up in this set and normalizing that number based on how many words were in the clue.

We also believed it was valuable to take into consideration that some words are more likely to show up in crossword puzzles than others. For example, the string “SSN”

appears in crossword puzzles very frequently, so it seems like we would want to give a higher weight answers like that. Note that this is similar to the human approach as experienced solvers likely give more weight to words they see often. The value of this feature was simply how many times the answer had shown up in previous puzzles.

The large domain size also forced us to examine the forward checking portion of an algorithm. In a normal situation, we would use AC-3 to reduce domain size but in this case, because there are so many possible domain values, we thought that it only made sense to check one word in advance. More specifically, we would only restrict the domains that directly intersected with the answer in question. We based this design decision on a few things. First, when we tried to use AC-3, we could not solve a 15x15 puzzle without the algorithm timing out. Second, only checking one variable ahead is similar to the way humans solve a crossword puzzle. Generally, when people go through the puzzle, they will solve sections at a time because finding an answer to one clue drastically affects your thinking about the potential nearby answers. Third, Ginsberg used this approach in his Dr. Fill algorithm, which also implemented backtracking and performed as well as some of the top solvers in the world.[9]

3. Experiments and Data

We ran our first set of experiments on 4x4 crossword puzzles, before tackling the more difficult 15x15 crossword puzzles. The results for the 4x4 crossword puzzles are shown in table 1. Furthermore, the program output from running the program on a 4x4 crossword puzzle in which the puzzle was solved with 100% accuracy is included in the appendix. Note that the output is heavily truncated, as many solutions were found.

Data Set(s) of possible words to fill	Type of puzzle	**Number of satisfiable answers excluding semantic analysis**	**Number of satisfiable answers after semantic analysis**
English words lexicon + “crosswordiest” + past answers	4x4	7200	1
English words with limited domain	4x4	343	1

Table 1: Results when run on 4x4

[9] Ginsberg, *Crosswords and an Implemented Solver for Singly-weighted CSPs*

As can be seen, when we ran our program on the 4x4, the CSP is able to generate several possible answers without the semantic analysis added in. This is because there are many answers to the puzzle, even if one does not take into account the clues. in particular, there are many 4 letter words which can satisfy the constraints of the crossword puzzle. However, after adding the clues, it is able to break the ties and give us the one possible answer for every case that we tested it on. In the case for the table shown above, the optimal solution was found when run on both datasets. The truncated output of this program is included in Appendix 1.

Note that the semantic analysis is able to differentiate the correct answer by a fairly large margin. As shown in Table 2 below, the score associated with the best answer is several standard deviations above the mean of all of the scores. This suggests that the semantic analysis is performing reliably when it comes to picking out the best answer out of all the possible answers. To see the list of scores that this data is generated from, please see Appendix 2.

Average score	**Standard Deviation**	**Best Score (score chosen as answer)**
1.335	0.4497	4.653

Table 2: Semantic analysis summary scores for a sample 4x4 puzzle

For this crossword puzzle, the accuracy was 100%, as the semantic analysis was able to discern the best possible set of clues out of the 7200 and 343 total words, when run on the two datasets respectively:

Best score = 4.65277777778
{(3, 'down', 'bad'): 'evil',
(2, 'down', 'rest'): 'ease',
(4, 'down', 'which is not fake'): 'real',
(1, 'across', 'Some drink it when they want to have fun'): 'beer',
(1, 'down', 'The loud cry of a mule or donkey'): 'bray',
(6, 'across', 'continent in the east of europe'): 'asia',
(7, 'across', 'to screem'): 'yell',
(5, 'across', 'special kind of music'): 'rave'}

We ran our program on ten 4x4 puzzles, and received the following data:

Number of fills	Number of fills solved correctly	Accuracy
8	7	87.5%
8	8	100%
8	7	87.5%
8	7	87.5%
8	8	100%
8	6	75%
8	7	87.5%
8	8	100%
8	7	87.5%
8	8	100%

Table 3: Results

Finally, when we ran our program with the semantic analysis domain-limiting optimizations on the 15x15, no solutions were found. This was due to the presence of long words in the puzzle, for which the domain did not contain adequate solutions for the variables. In particular, we found that the 15 character words in a 15x15 were very specific, and thus it was impossible, or at least highly unlikely, to generate an optimal assignment such that all 15 constraints on each character in the word was correctly assigned. In order to combat this, we added in the solutions for words larger than 6 letters, and ran on the dataset to determine whether the crossword would solve. With domain limiting of 1500, the Amazon EC2 machine took 19 hours to solve, and yielded 6 total solutions. Out of 74 total assignments, the highest accuracy was 11/74 words correctly identified, of which 21 solution words with length 6 or higher were added to the dataset. Thus, the total accuracy was approximately 14.8%.

Conclusion

Overall, it is clear that solving a crossword puzzle is not a trivial task. We experienced a great deal of difficulty in ensuring assignments in the CSP could be made, and although we had great results in the 4x4 trials, we were not able to solve the CSP in the 15x15 unless we added the solutions for words that were length 6 or longer

in the crossword puzzle. Furthermore, because it took so long to compute the solution for a 15x15, we were only able to solve 1 puzzle of this size.

For future work, we can do a great deal to tune our semantic analysis for the clues. Without a doubt, semantic analysis is the most effective tool in solving the crossword puzzles, as simply adding words to the domain for variables either makes the complexity unwieldy, or yields incorrect fill answers. As a result, this would be the best area for future work on this project.

Appendix 1:

Program output when run on a 4x4:

```
Reading crossword puzzle JSON data and forming CSP...
Reading clue data and training feature vector...
Adding CSP Variables...
currently testing: hidden type of clue
Added CSP Variable: (1 across) with domain length 4
currently testing: hidden type of clue
Added CSP Variable: (5 across) with domain length 4
currently testing: hidden type of clue
currently testing: container type of clue
Added CSP Variable: (6 across) with domain length 4
Added CSP Variable: (7 across) with domain length 4
Added CSP Variable: (1 down) with domain length 4
Added CSP Variable: (2 down) with domain length 4
Added CSP Variable: (3 down) with domain length 4
Added CSP Variable: (4 down) with domain length 4
Computing binary potentials  [0, 1, 2, 3, 4] in parallel out of 15 total
Process spawned. Generating binary potential tables of dimensions: 477 x 477 = 227529 potentials...
Process spawned. Generating binary potential tables of dimensions: 477 x 477 = 227529 potentials...
Process spawned. Generating binary potential tables of dimensions: 477 x 477 = 227529 potentials...
Process spawned. Generating binary potential tables of dimensions: 477 x 477 = 227529 potentials...
Process spawned. Generating binary potential tables of dimensions: 463 x 463 = 214369 potentials...
Added CSP Binary Potential between (5, 'across', 'special kind of music') and (2, 'down', 'rest') Added
CSP Binary Potential:
        (5, 'across', 'special kind of music') char 1 intersects (2, 'down', 'rest') char 1
Added CSP Binary Potential between (1, 'across', 'Some drink it when they want to have fun') and (4,
'down', 'which is not fake') Added CSP Binary Potential:
        (1, 'across', 'Some drink it when they want to have fun') char 3 intersects (4, 'down', 'which is
not fake') char 0
Added CSP Binary Potential between (1, 'across', 'Some drink it when they want to have fun') and (3,
'down', 'bad') Added CSP Binary Potential:
        (1, 'across', 'Some drink it when they want to have fun') char 2 intersects (3, 'down', 'bad')
char 0
Added CSP Binary Potential between (1, 'across', 'Some drink it when they want to have fun') and (1,
'down', 'The loud cry of a mule or donkey') Added CSP Binary Potential:
        (1, 'across', 'Some drink it when they want to have fun') char 0 intersects (1, 'down', 'The loud
cry of a mule or donkey') char 0
Added CSP Binary Potential between (1, 'across', 'Some drink it when they want to have fun') and (2,
'down', 'rest') Added CSP Binary Potential:
        (1, 'across', 'Some drink it when they want to have fun') char 1 intersects (2, 'down', 'rest')
char 0
Computing binary potentials  [5, 6, 7, 8, 9] in parallel out of 15 total
Process spawned. Generating binary potential tables of dimensions: 463 x 463 = 214369 potentials...
Process spawned. Generating binary potential tables of dimensions: 463 x 463 = 214369 potentials...
Process spawned. Generating binary potential tables of dimensions: 463 x 463 = 214369 potentials...
Process spawned. Generating binary potential tables of dimensions: 471 x 471 = 221841 potentials...
Process spawned. Generating binary potential tables of dimensions: 471 x 471 = 221841 potentials...
Added CSP Binary Potential between (5, 'across', 'special kind of music') and (1, 'down', 'The loud cry of
a mule or donkey') Added CSP Binary Potential:
        (5, 'across', 'special kind of music') char 0 intersects (1, 'down', 'The loud cry of a mule or
donkey') char 1
Added CSP Binary Potential between (5, 'across', 'special kind of music') and (3, 'down', 'bad') Added CSP
Binary Potential:
        (5, 'across', 'special kind of music') char 2 intersects (3, 'down', 'bad') char 1
```

```
Added CSP Binary Potential between (5, 'across', 'special kind of music') and (4, 'down', 'which is not
fake') Added CSP Binary Potential:
        (5, 'across', 'special kind of music') char 3 intersects (4, 'down', 'which is not fake') char 1
Added CSP Binary Potential between (6, 'across', 'continent in the east of europe') and (2, 'down',
'rest') Added CSP Binary Potential:
        (6, 'across', 'continent in the east of europe') char 1 intersects (2, 'down', 'rest') char 2
Added CSP Binary Potential between (6, 'across', 'continent in the east of europe') and (1, 'down', 'The
loud cry of a mule or donkey') Added CSP Binary Potential:
        (6, 'across', 'continent in the east of europe') char 0 intersects (1, 'down', 'The loud cry of a
mule or donkey') char 2
Computing binary potentials  [10, 11, 12, 13, 14] in parallel out of 15 total
Process spawned. Generating binary potential tables of dimensions: 471 x 471 = 221841 potentials...
Process spawned. Generating binary potential tables of dimensions: 471 x 471 = 221841 potentials...
Process spawned. Generating binary potential tables of dimensions: 449 x 449 = 201601 potentials...
Process spawned. Generating binary potential tables of dimensions: 449 x 449 = 201601 potentials...
Process spawned. Generating binary potential tables of dimensions: 449 x 449 = 201601 potentials...
Added CSP Binary Potential between (7, 'across', 'to screem') and (1, 'down', 'The loud cry of a mule or
donkey') Added CSP Binary Potential:
        (7, 'across', 'to screem') char 0 intersects (1, 'down', 'The loud cry of a mule or donkey') char
3
Added CSP Binary Potential between (7, 'across', 'to screem') and (2, 'down', 'rest') Added CSP Binary
Potential:
        (7, 'across', 'to screem') char 1 intersects (2, 'down', 'rest') char 3
Added CSP Binary Potential between (7, 'across', 'to screem') and (4, 'down', 'which is not fake') Added
CSP Binary Potential:
        (7, 'across', 'to screem') char 3 intersects (4, 'down', 'which is not fake') char 3
Added CSP Binary Potential between (6, 'across', 'continent in the east of europe') and (4, 'down', 'which
is not fake') Added CSP Binary Potential:
        (6, 'across', 'continent in the east of europe') char 3 intersects (4, 'down', 'which is not
fake') char 2
Added CSP Binary Potential between (6, 'across', 'continent in the east of europe') and (3, 'down', 'bad')
Added CSP Binary Potential:
        (6, 'across', 'continent in the east of europe') char 2 intersects (3, 'down', 'bad') char 2
Computing binary potentials  [15] in parallel out of 15 total
Process spawned. Generating binary potential tables of dimensions: 449 x 449 = 201601 potentials…

assigned 1/8 variables
assigned 2/8 variables
assigned 3/8 variables
assigned 4/8 variables
assigned 5/8 variables
assigned 6/8 variables
assigned 7/8 variables
assigned 8/8 variables

SOLUTION FOUND:
3 down: odin
2 down: once
4 down: nose
1 across: moon
1 down: ming
6 across: ncis
7 across: gene
5 across: indo

assigned 1/8 variables
assigned 2/8 variables
```

```
assigned 3/8 variables
assigned 4/8 variables
assigned 5/8 variables
assigned 6/8 variables
assigned 7/8 variables
assigned 8/8 variables

SOLUTION FOUND:
3 down: moon
2 down: alee
4 down: yens
1 across: gamy
1 down: gang
6 across: neon
7 across: gens
5 across: aloe
```

[truncated, hundreds of solutions found]

```
-------Solution Score:  0.833333333333 ---------------
clue:  (3, 'down', 'bad')
answer:  mint
0.0

clue:  (2, 'down', 'rest')
answer:  oles
0.0

clue:  (4, 'down', 'which is not fake')
answer:  sous
0.0

clue:  (1, 'across', 'Some drink it when they want to have fun')
answer:  noms
0.0

clue:  (1, 'down', 'The loud cry of a mule or donkey')
answer:  nome
0.25

clue:  (6, 'across', 'continent in the east of europe')
answer:  menu
0.333333333333

clue:  (7, 'across', 'to screem')
answer:  ests
0.0

clue:  (5, 'across', 'special kind of music')
answer:  olio
0.25

-------Solution Score:  0.833333333333 ---------------
clue:  (3, 'down', 'bad')
answer:  mist
0.0
```

```
clue:  (2, 'down', 'rest')
answer:  oles
0.0

clue:  (4, 'down', 'which is not fake')
answer:  etas
0.0

clue:  (1, 'across', 'Some drink it when they want to have fun')
answer:  nome
0.0

clue:  (1, 'down', 'The loud cry of a mule or donkey')
answer:  name
0.5

clue:  (6, 'across', 'continent in the east of europe')
answer:  mesa
0.333333333333

clue:  (7, 'across', 'to screem')
answer:  ests
0.0

clue:  (5, 'across', 'special kind of music')
answer:  alit
0.0

[truncated, hundreds of solutions computed for best score]

Best score =  4.65277777778
(3, 'down', 'bad')
(2, 'down', 'rest')
(4, 'down', 'which is not fake')
(1, 'across', 'Some drink it when they want to have fun')
(1, 'down', 'The loud cry of a mule or donkey')
(6, 'across', 'continent in the east of europe')
(7, 'across', 'to screem')
(5, 'across', 'special kind of music')
Total words:  8
Percent correct:  100%
```

Appendix 2:Output of list of scores generated from “after the fact” semantic analysis

Solution Scores: [2.0, 1.6666666666666667, 2.305555555555556, 2.305555555555556, 2.305555555555556,
0.1111111111111111, 1.4444444444444444, 0.9166666666666666, 0.6944444444444444, 0.5833333333333333,
0.8333333333333333, 0.8333333333333333, 0.8333333333333333, 1.3333333333333333, 1.6944444444444444,
1.6111111111111112, 1.6111111111111112, 1.8333333333333333, 1.2083333333333333, 1.2222222222222223,
3.333333333333333, 2.5555555555555554, 2.5555555555555554, 2.125, 1.1666666666666665, 1.3333333333333333,
1.3333333333333333, 1.3333333333333333, 1.3333333333333333, 0.75, 0.75, 0.25, 0.9861111111111112,
0.9861111111111112, 0.9861111111111112, 0.9861111111111112, 1.1527777777777777, 1.6527777777777777,
1.4861111111111112, 1.1527777777777777, 1.3194444444444444, 1.3194444444444444, 1.4027777777777777,
1.7361111111111112, 1.7361111111111112, 1.0138888888888888, 1.2638888888888888, 1.2638888888888888,
1.0972222222222223, 1.2638888888888888, 1.2638888888888888, 1.0972222222222223, 1.4305555555555556,
1.0138888888888888, 1.4305555555555556, 1.4305555555555556, 1.5138888888888888, 1.5138888888888888,
1.8472222222222223, 1.6805555555555556, 1.8472222222222223, 2.8472222222222223, 1.5138888888888888,
1.3472222222222223, 1.4305555555555556, 1.4305555555555556, 1.25, 1.75, 1.5833333333333333, 1.25,
1.4166666666666665, 1.4166666666666665, 1.8333333333333333, 1.6666666666666665, 1.6666666666666665,
1.6111111111111112, 1.6111111111111112, 1.5, 1.6111111111111112, 1.6111111111111112, 1.2638888888888888,
1.2638888888888888, 1.0138888888888888, 1.1527777777777777, 1.5972222222222223, 1.8472222222222223,
1.8472222222222223, 1.4305555555555556, 1.4305555555555556, 1.8472222222222223, 1.8472222222222223,
1.4305555555555556, 1.4305555555555556, 1.4166666666666665, 1.1388888888888888, 1.3055555555555556,
0.8611111111111112, 1.1111111111111112, 0.9722222222222222, 1.4027777777777777, 1.4722222222222223,
2.111111111111111, 1.75, 2.111111111111111, 1.75, 1.75, 1.1388888888888888, 1.1388888888888888,
1.1111111111111112, 1.1111111111111112, 1.8611111111111112, 1.3333333333333333, 1.4027777777777777,
1.5694444444444444, 1.0, 0.9166666666666666, 0.75, 1.1666666666666665, 1.5, 1.0833333333333333,
0.9166666666666666, 1.375, 1.3333333333333333, 1.3333333333333333, 1.1666666666666665, 1.0,
1.1666666666666665, 1.3333333333333333, 1.2361111111111112, 1.2361111111111112, 1.2361111111111112,
1.2361111111111112, 0.9861111111111112, 1.0416666666666665, 1.0416666666666665, 0.8472222222222222,
0.8472222222222222, 1.0972222222222223, 1.3333333333333333, 1.3333333333333333, 1.3333333333333333,
1.3333333333333333, 1.4166666666666665, 1.5833333333333333, 1.0416666666666665, 1.0416666666666665,
1.4166666666666665, 0.9166666666666666, 1.2361111111111112, 1.4166666666666665, 4.652777777777779, 0.625,
0.5, 1.125, 0.9583333333333333, 0.9583333333333333, 0.625, 0.7916666666666666, 0.9583333333333333, 1.375,
0.9166666666666666, 1.0277777777777777, 1.0277777777777777, 1.6388888888888888, 1.1388888888888888,
1.5277777777777777, 1.0277777777777777, 1.0277777777777777, 1.25, 1.4166666666666665, 1.25,
1.4166666666666665, 1.25, 1.25, 1.25, 1.25, 0.9166666666666666, 1.1388888888888888, 1.4722222222222223,
1.4722222222222223, 1.1388888888888888, 1.1388888888888888, 1.25, 1.4166666666666665, 0.9166666666666666,
1.0277777777777777, 1.0833333333333333, 0.75, 1.3611111111111112, 1.3611111111111112, 1.0277777777777777,
1.0277777777777777, 1.25, 1.25, 0.9166666666666666, 1.7222222222222223, 1.5833333333333333,
1.1388888888888888, 1.25, 1.25, 1.3888888888888888, 1.7361111111111112, 0.9027777777777778,
1.4722222222222223, 1.5, 1.3333333333333333, 2.111111111111111, 1.3611111111111112, 1.3611111111111112,
1.3611111111111112, 1.3611111111111112, 1.1666666666666665, 1.6111111111111112, 1.6111111111111112,
1.8333333333333333, 1.6388888888888888, 1.6388888888888888, 1.4166666666666665, 1.125, 1.125,
2.0138888888888893, 1.5972222222222223, 1.5972222222222223, 0.6111111111111112, 1.1527777777777777,
1.6527777777777777, 1.4861111111111112, 1.1527777777777777, 1.3194444444444444, 1.3194444444444444,
1.4027777777777777, 1.7361111111111112, 1.7361111111111112, 1.2638888888888888, 1.2638888888888888,
1.0972222222222223, 1.2638888888888888, 1.2638888888888888, 1.0972222222222223, 1.4305555555555556,
1.0138888888888888, 1.0138888888888888, 1.5138888888888888, 1.5138888888888888, 1.8472222222222223,
1.6805555555555556, 1.8472222222222223, 2.8472222222222223, 1.5138888888888888, 1.3472222222222223,
1.4305555555555556, 1.4305555555555556, 1.4305555555555556, 1.4305555555555556, 1.25, 1.75,
1.5833333333333333, 1.25, 1.4166666666666665, 1.4166666666666665, 1.8333333333333333, 1.6666666666666665,
1.6666666666666665, 1.5138888888888888, 1.7638888888888888, 1.7638888888888888, 1.6111111111111112,
1.2638888888888888, 1.2638888888888888, 1.0138888888888888, 1.1527777777777777, 1.5972222222222223,
1.8472222222222223, 1.8472222222222223, 1.4305555555555556, 1.4305555555555556, 1.8472222222222223,
1.8472222222222223, 1.4305555555555556, 1.4305555555555556, 1.4166666666666665, 1.2361111111111112,
1.2361111111111112, 1.2361111111111112, 1.2361111111111112, 0.9861111111111112, 1.0416666666666665,
1.0416666666666665, 0.8472222222222222, 0.8472222222222222, 1.0972222222222223, 1.3333333333333333,

1.3333333333333333, 1.3333333333333333, 1.3333333333333333, 1.4166666666666665, 1.5833333333333333,
1.0416666666666665, 1.0416666666666665, 0.875, 0.875, 1.4166666666666665, 1.3333333333333333,
1.4305555555555556, 2.5972222222222223, 0.5833333333333333, 1.5, 1.5, 1.5, 1.5, 1.5, 2.5,
1.3333333333333333, 1.125, 1.125, 0.5833333333333333, 0.5833333333333333, 0.41666666666666663,
0.41666666666666663, 0.8333333333333333, 0.8333333333333333, 0.5833333333333333, 0.5833333333333333, 1.5,
1.5, 1.5, 1.5, 1.5, 1.25, 0.9166666666666666, 0.6111111111111112, 0.5, 0.5, 0.5833333333333333,
0.5833333333333333, 1.5833333333333333, 1.8055555555555556, 1.8055555555555556, 0.41666666666666663,
0.8333333333333333, 0.8333333333333333, 0.8333333333333333, 0.8333333333333333]

Appendix 3: Answers to the sample puzzle that was used to generate the data from Appendix 2 as well as other data

B	E	E	R
R	A	V	E
A	S	I	A
Y	E	L	L