CPSC 490 - Final Report

Yale University, Department of Computer Science

Solving Cryptic Crossword Puzzles

Josh Rozner

April 28, 2014

Advisor: Prof. Dana Angluin,
Dept. of Computer Science, Yale University
1 Abstract

Powerful computers and linguistic databases and libraries have made artificial manipulation of human language by computers substantially more attainable than in the recent past. Computers are frequently used in mathematical puzzle solving, but language puzzle solving does not lend itself as easily to computation. Cryptic crossword puzzles lie at the intersection between formal rules and linguistic patterns. Clues obey certain rules and have a decently well-defined set of indicator patterns, so we can to some extent formulate solution algorithms. These can in turn be verified and augmented using language databases (dictionaries, thesauruses, etc.) and linguistic manipulation libraries. Here we demonstrate effective solutions to anagram and hidden clue types and limited solutions to reversal and double clue types. We additionally provide output that describes how a solution was reached—essentially the solver’s “thought” process. Many of the solution methods demonstrated here could, with the appropriate resources, be extended to other clue types to yield a full solver.
2 Introduction

2.1 Motivation

Puzzle solving is an interesting application of computer science. Some mathematically based problems are particularly well-handled by computers. Language, on the other hand, is substantially more complex and harder to translate into a machine-solvable problem. Normal crossword puzzle solvers, for example, consist mostly of clue databases, mapping a clue to previously observed solutions.

Cryptic crosswords are an interesting subset of language puzzles. The language problem involves translating a definition or “sense” into a word. This is known as a reverse dictionary lookup (normal dictionaries map words to definitions; reverse dictionaries map definitions to words) [1]. Since cryptic crosswords obey certain rules that we can model somewhat effectively, they are a rule-satisfying subset of the reverse dictionary lookup problem. In some cases these rules can be so straightforwardly implemented that our solver might be more effective than a human, which is ultimately the goal of a good puzzle solver.

Other applications of particularly good reverse dictionaries are areas that involve intent guessing. This is extremely relevant to interpreting web search queries.

2.2 Introduction to cryptic crossword puzzles

Much of the following section has been adopted from the wikipedia page [2]. Cryptic crosswords consist of a grid and clues, much like normal crosswords. However, there is substantially less overlap in the grid, and the clues take a particular, “cryptic” form. But for a few exceptional cases, cryptics generally follow certain rules:

1. A length for the solution is provided
2. A definition for the solution word comes either at the end or the beginning of the clue
3. The solution word can be constructed by obeying a set of cryptic, wordplay rules with the “half” of the clue that is not part of the definition
4. The wordplay rules are often indicated by sets of indicator words

To allow the reader to understand the solution method, we now provide some examples of the forms of cryptic clues, organized alphabetically. Following each clue, we provide a level of difficulty of automated solution, on a scale of 1-5, 1 being easiest and 5 being hardest. This difficulty level is based on the following analysis of solution methods and on experience in building this solver.

2.2.1 Anagrams

Rearrange part of the clue.

*Chaperone shredded corset (6)*

“shredded” is a word that indicates that some part of the clue should be anagrammed (literally shredded) to give a new word that satisfies the definition portion of the clue. Other anagram indicators include “shred,” “stir,” and other semantically similar words. In this case, corset can be rearranged to escort, which is a synonym of chaperone.

Anagrams can be complicated in the following ways:
• An anagram of multiple words is required, e.g.,

No heart gets broken by someone else (7)

is an anagram indicated by “broken.” An anagram of “no heart” is “another” which matches the definition “someone else.”

• An anagram involving initial letters and some words.

• An indirect anagram, where a synonym of a word in the solution should be anagrammed, e.g., in

Chew honeydew fruit (5)

We first take melon as a synonym of honeydew and “chew,” or anagram it to “lemon” to get a fruit.

The above form of anagram solution lends itself easily to a computed answer. A simple set of steps to solution might be:

1. If an indicator word exists
2. Anagram words, combinations of words, and combinations of initial letters and words
3. See if any of the valid anagrams (i.e. real English words) match the definition. From here on, this step is denoted as “Compare to definition”

Complexity: 1 Anagrams have indicators, and computing anagrams should be straightforward, with a high probability of a definite solution

2.2.2 Charades

Combine words to get a solution.

Outlaw leader managing money (7)

Solutions involve taking synonyms of two or more words and joining those synonyms together. In this case, we join “ban,” a synonym of “outlaw” and “king,” a synonym of “leader,” to get “banking,” which can be defined as “managing money.”

Charades do not have indicator words, so possible steps to solution:

1. Find synonyms of words in clue
2. Look for combinations of synonyms that match the clue length and are valid English words
3. Compare word combinations to definition

Complexity: 5 There are no indicators, and pairing lists of synonyms is $O(n^2)$ for two word combinations

2.2.3 Containers

Place some letters inside a word or other set of letters.

Apostle’s friend outside of university (4)

Put “pal,” from “friend,” outside of “U” for “university” to get “paul,” an apostle. Note here we encounter an additional level of complexity: we must find an instance of apostle rather than a definition. Another example of this sort of complication would be Turkey for country.

Containers do have indicator words, so steps to solution:
1. Find indicator word (note here this also clues how to compute the solution, e.g. “outside of”)

2. Get synonyms and initial letters and shuffle according to the clue word. Note that exactly following the clue word is probably more difficult than just computing anagrams.

3. Check definition

*Complexity: 3-4* There are indicators, but we must find synonyms before anagramming.

### 2.2.4 Deletions

Remove a letter from a word (beginning, end, or middle) to get a solution.

*Beheaded celebrity is sailor (3)*

Substitute “star” for “celebrity” and behead “star” to get “tar,” a synonym for “sailor.”

Solution steps:

1. Find indicator

2. Get synonyms and perform beheadment, curtailment, or internal deletion. Note that the first two are computationally easier.

3. Check definition

*Complexity: 3* The steps to solution are very similar to those for Containers, but we avoid having to combine different sets of letters and words.

### 2.2.5 Double definitions

These clues lack wordplay. Instead, the solution will match a definition at the beginning and end of the clue.

*Bird country (6)*

Gives “turkey,” which is both a bird and a country.

These clue types lend themselves to reverse dictionary lookups [1].

*Complexity: 4* Depending on the efficiency of our reverse dictionary and semantic analysis, we might need to split the clue in many positions (total possible split locations is \((\text{number of words in clue}) - 1\)) and attempt \(2 \times (\text{number of split locations}) = 2 \times (\text{words} - 1)\) reverse lookups. Without an efficient reverse dictionary, this is extremely costly.

### 2.2.6 Hidden words

The solution is hidden in the letters of the clue.

*Cook some lunch effectively (4)*

“chef” occurs at lunch effectively and matches “cook.” “Some” functions as a hidden indicator.

Hidden solutions can be complicated by taking initial letters, or odd/even letters.

Solutions steps:

1. Find indicator word

2. Search for valid words in string of text (Note not all strings should be considered, and the possible complications mentioned above must be handled carefully, though in a straightforward computational fashion.)
3. Check definition

Complexity: 1 Indicator words and the presence of the solution word in the letters of the clue lend this solution type to computation.

2.2.7 Other types

- Homophones: Taking a homophone to get a solution. E.g., We hear twins shave (4) gives twins $\rightarrow$ pair $\rightarrow$ pare, matches shave. “Hear” indicates homophone.
- Reversal: Flip a word to get a solution. E.g., Returned beer fit for a king (5) gives beer $\rightarrow$ lager $\rightarrow$ regal, matches king. “Returned” indicates reversal.

2.2.8 More complex types

A discussion of the following clue types can be found on the cryptic crosswords wikipedia page [2]. Combinations involve combinations of different clue types, as the name suggests.

- Cryptic definition
- Spoonerism
- Literal
- Combinations

3 Implementation

3.1 Classification

With a grasp of cryptic clue structure, we can now perform a classification of clues:

1. Direct computation: Clues that can be solved without going through synonym sets.
   Anagrams, Hiddens

2. Initial synonymization on a single word: Clues that involve performing an action on a synonym of a word in the initial clue.
   Indirect anagrams, Charades, Containers, Deletions, Homophones, Reversals

3. Indicator words: Clues whose solution type is generally denoted by a certain indicator word or phrase.
   Anagrams, Hiddens, Containers, Deletions, Homophones, Reversals

4. Other: Clues that are more complicated than the above.
   Double definitions, Cryptic definitions, Spoonerisms, Literals

5. Combinations: Clues that involve more than one type of the above.

With this in mind, it is evident that anagrams and hidden type clues will be the easiest to solve, so they are the first types we implement. Beyond that, we attempt some of the clue types that have indicator words and involve a single initial synonymization.
3.2 Basic implementation

Barring the consideration of combination clue types, we can follow a relatively straightforward clue solving technique:

```python
for solver in solver set:
    if (clue doesn’t have indicator word):
        continue
    if (solver requires initial synonimization):
        clue = clue.synonimize()
    possible solutions = clue.manipulate()  # anagram, search for hidden, reverse, etc.
    for soln in possible solutions:
        if soln is not valid English:
            continue
        if clue.checkDefinition(soln)  # i.e., does this solution match the definition
            add soln to list of possible solutions
return list of possible solutions
```

Our solver implements the above technique. A more sophisticated, possibly more efficient technique might turn the clue into something akin to a parse tree, where each node is a possible key word that indicates some sort of manipulation (anagram, hidden, put inside, reverse) of a neighboring word before or after synonimization. This method could be more efficient because, for example, we might consider anagrams only of words or sets of words nearby an anagram indicator, rather than any length-valid combination of letters in the clue.

We proceed to implement as follows:

1. Class Clue: An incoming clue string is parsed into a string of terms and a length
2. Class Term: A Clue is made up of Terms that contain a word and a synonym / definition set (syn-def set) for that word. We compute syn-def sets only once per term in the clue.
3. Class Solver: implements method getSolutions(Clue). We have a solver for each clue type that we implement. The anagram solver will compute valid anagrams, the hidden solver will look for valid words hidden in the clue text, etc.
4. Class Solution: A solution object tracks the formation of a given solution word, including the type of Solver, the indicator word, and notes on how the solution word was reached.
5. Algorithm: Deriving syn-def sets. We make use of Python’s Natural Language Toolkit (NLTK) and a thesaurus database we generate per the algorithm detailed below.
6. Algorithm: Syn-def intersection scoring
7. Algorithm: Checking solutions. We use a set similarity algorithm akin to a cross-product to compute the nearness of a solution word to a clue’s probable definition. This is a comparison of multiple syn-def sets.

3.3 Composing syn-def sets

Wordnet is part of the nltk library [3]. It provides synonyms, hypernyms, and hyponym sets, as well as definitions. Our initial implementation derived syn-def sets by iterating over synonyms, hypernyms, hyponyms, and definitions in a breadth first fashion. A syn-set then consisted of a list of sets of related terms at different
depth levels.

The second implementation made use of the moby thesaurus [4]. This is a huge textfile of synonyms. From it we constructed a thesaurus database that maps from a word to its synonyms. A secondary database maps from a word, x, to the words y such that \( x \in \text{syns}(y) \) where \( \text{syn}(y) \) is the function provided by the first database. Scripts to generate such databases from thesaurus input files are provided in the aux/ folder of the codebase. Generation of the “back thesaurus” took less than five minutes.

In the final implementation, both implementations are used, and the higher score from the syn-def intersection algorithm across both methods is used.

### 3.4 Syn-def-set intersection scoring

Given two words and their corresponding syn-def sets, we compute a score for their similarity by taking a weighted set intersection.

1. We consider the intersection of two thesaurus sets (pure synonyms), S and T, where \( S[0] \) is the word, \( S[1] \) is first level synonyms, etc. We take the magnitude of the intersection of \( S[i] \) and \( S[j] \) and scale by the level of the synonyms using the divisor \((1 + i + j)\). We add the 1 to avoid division by zero.

\[
\text{score} = 0 \\
\text{for } i \text{ in range}(0, \text{len}(S)): \\
\text{for } j \text{ in range}(0, \text{len}(T)): \\
\text{score += len}(S[i].\text{intersection}(T[j])) / (1 + i + j) \\
\text{return score}
\]

2. The intersection of wordnet syn-def sets is slightly more complicated. Since definitions can contain many words that individually are unrelated to a given term, we exponentially decrease their weight as we move away from the definition. Here S is a syndef set where \( S.\text{syns}[i] \) gives the synonyms at level i and \( S.\text{defs}[i] \) gives the terms of a definition at level i.

\[
\text{score} = 0 \\
\text{for } i \text{ in range}(0, \text{len}(S)): \\
\text{for } j \text{ in range}(0, \text{len}(T)): \\
\text{score += len}(S.\text{syns}[i].\text{intersection}(T[j].\text{syns})) / (1 + i + j) \\
\text{score += len}(S.\text{defs}[i].\text{intersection}(T.\text{defs}[j].\text{defs})) / (2 \ll (i + j + 1)) \\
\text{# the syn-def intersection requires a couple lines to get syns} \\
\text{# and refs from both S and T and then do a combination of} \\
\text{# exponential and syn weighting.} \\
\text{# here we omit the details} \\
\text{score += syn-def-intersection(S, J, i, j)} \\
\text{return score}
\]

Note that this algorithm is extremely crude. Wordnet provides some other intersection algorithms that work well for two given words but work less well for going from a definition to a word. This is further discussed in the improvements section.
3.5 Checking solutions

Once we have found a number of possible solutions based on the clue type, we attempt to compute their similarity to the definition in the clue. We mark words in the clue as used if they are an indicator word, or part of an anagram or hidden word solution. Then we look at both sides of the clue and walk inward until finding a used term. The words from the end to the first used term are a tentative definition and we perform a syn-set-def intersection over these terms and our solution word. We take whichever side scores higher as the definition and return the syn-set score.

3.6 Anagram Solver: a case study

The anagram solver was the first we implemented. It is arguably slightly harder than the hidden solver since we must find anagrams, which is more difficult than finding words that are simple substrings (how the hidden solver works).

List of cryptic crossword indicator words for this and other clue types were obtained from a cryptic crossword webpage [5]. As the indicator list has less than 1000 phrases, it can be stored as a text file and read into the solver on initialization as a sorted array. A binary search through the indicator list for each word in the clue is not overly costly. Here we use a binary search instead of a hash set to allow partial matches (e.g. shredded vs. shred) that are handled using python’s difflib. A better approach here might be to use a stemmer (wordnet does provide one) on the indicator file and a stemmer on the clue words. Then a hash set could be used, saving time, and avoiding false indicator matches, which do sometimes occur in the current solver.

If an indicator is found in the clue, then a subset sum algorithm is used to find valid combinations of clue words (i.e. word sets whose cumulative length sums to the desired clue length).

The initial implementation then computed anagrams of all valid letter subsets and then used the pycharm library to find the permutations that were valid English words. This worked but was subpar because pycharm’s dictionary is rather incomplete and because permuting words more than eight characters long is particularly time intensive. Certainly for solutions involving words of length 13 or more, the program would run for years.

The solution was to use the linux dictionary present at /usr/share/dict/words. We hash all words in the dictionary by sorting the letters and then hashing. The generation of the anagram database from the text file took less than five seconds. It maps from (sorted set of letters) to a list of valid words containing those letters. The database has since been updated with a few more open source dictionaries as well as with the words included in the moby thesaurus [4]. Files to augment the database are present in the aux/ directory.

From here, we check all resulting valid anagrams against the clue using the definition matching algorithm described above.

4 Results

The anagram and hidden solvers are well-implemented: they succeed everywhere that they should. We did not, however, implement the more complex functionalities of abbreviations or taking only starting letters of words. As predicted, anagram and hidden type clues were easiest to solve as they arguably require the least
artificial intelligence.

Code was written for double and reversal solvers. The latter type of clue is not particularly prevalent in the test crosswords taken from [6]. It does however work on the reversal example given above. The double solver does not work well even in simple cases. It does successfully solve some clues though, including *Draw neckwear* (3) (tie) and *Actors in a shed* (4) (cast). The double solver fails in the case of longer clues because it looks primarily for overlapping synonyms of individual words, rather than matching the sense of longer definitions.

In puzzles with many anagram and hidden type clues, the solver does well. In one puzzle, 50% solution success (the correct solution was in the top 3 scoring solutions) was observed. In others, solution rates can be as low as 20%.

The aux/ folder contains code for scraping puzzles from the cryptic crossword puzzle site [6]. As we could not find an OCR module for converting crossword solution images to word lists, test files must have solutions manually entered. A script in aux/ is provided to make this process easier.

## 5 Improvements

Finding or implementing a reverse dictionary would be particularly useful, especially in the case of solving double type clues. With a reverse dictionary, it might be optimal to compute possible senses of the clue prior to proceeding with other types of solution analysis. Furthermore, it’s presently difficult to find the exact set of words to use as the definition portion of the clue. The reverse dictionary could improve this.

Our syn-def comparison algorithm is a particularly crude implementation of the reverse dictionary algorithm described in the Shaw paper [1]. Whereas we use a stop indicator in the form of word length, for example, the paper has two levels of stop sets that are trained based on language.

Stemming, to find root words would be particularly useful in multiple stages of solving. Indicator words could be stemmed, as mentioned before to improve matching and reduce false positives. Some clues are not solved because of the presence of plurals or tense modifiers. The removal and readdition, post-solution, of such modifiers could substantially enhance the solver’s performance. Wordnet provides a stemming interface that could be used.

Our current check definition method could be improved via the use of Wordnet’s lowest-common-hypernyms() function, which we became aware of only after writing the initial algorithms. The function could also be used to improve solutions to simple double type clues involving only two to four words.

Presently, finding instances of a word, e.g., Turkey for bird, country, or Paul for apostle, is poorly handled. Wordnet’s common-hypernyms() function might be useful for this, but otherwise another resource could be useful. One solution method might involve executing google searches using the search API [7].

More indicator information could be stored in our text files, e.g., whether the indicator generally points to the preceding or subsequent word. We also don’t handle indicator phrases very well at present. Reversal indicators are also different based on whether the clue runs across or down in the puzzle. Our solver does not accept down
or up as input.

We could also implement a machine learning component. The indicators could have scores associated with how good they are as indicators. In cases where many solutions result, the solver could adjust synonym scoring (this would require keeping scores in the synonym database) based on the correct solution. With a crossword OCR) reader, we could then run large training sets through the solver.

As our code is presently implemented as pure solvers that run independently of each other, we could parallelize the running of different solvers.

6 Running the Code

1. The code has been run using Python 2.7.6. Python 2 was chosen over Python 3, as Wordnet for Python 3 is still in its alpha stage with limited support.

2. Running the code requires the nltk library and corpus downloads.

3. The databases required for the code can be generated from dictionary and thesaurus text files using the python scripts in the aux/ folder. All databases and indicator files should be placed in lib/. The indicator files should be sorted alphabetically, with one term / phrase per line.

Conclusion

Further solutions are not particularly difficult to implement once we have decent synonym and definition lookups. Containers, charades, and deletions are computationally straightforward. Double definitions are easily solved with a reverse dictionary. Homophones would require a special homophone library.

Our solver could still use substantial calibration. In cases where indicators could suggest multiple clue types, the scores from different solver types are not always well-scaled. This is particularly true with double clue types, where the solutions are generated from synonym lookup. When the double solver is enabled, it sometimes pushes out the valid solution from another solver.

By far the most complex modification would be to allow combination solution types. We would parse into something like a tree and be far more discerning in the words we modify and synonimize. For example, in Terribly alert at some time in the future (5), with indicator “terribly,” we would anagram only alert to get the solution “later,” rather than anagramming every five letter combination in the clue.

Lastly, we note that a full solver would need to take as input the crossword grid. It would use a dynamic programming algorithm to find the best solution to each clue type and fill the puzzle appropriately.

References


6. Cryptic crosswords and solutions: [bestforpuzzles.com](http://bestforpuzzles.com)

7. Goldschmidt, David. “Solving Crossword Puzzles Via the Google API”