FIT3036 - Computer Science Project - Final Report

CCS: Cryptic Clue Solver

Author:
Jaroslaw Glowacki

Supervisor:
Dr. Marc Cheong

Abstract

This report outlines the implementation and features of a newly developed tool designed as an aide to solving cryptic crosswords. Capable of deciphering most simple anagram, run, double definition, charade and code clues, narrowing down solutions when provided a word length or known letters, and brute-forcing solutions to clues it cannot handle, its purpose at this stage is not to remove the human solver from the equation, but to provide a means of processing all of the common cryptic clue patterns as the user focuses on the less trivial, lateral thinking approaches, such as puns. The latest iteration of the tool is capable of independently solving 10.40% of all Sydney Morning Herald cryptic crossword clues (of a pool of over 38,000) [1], given only the clue and word length, which increases to 13.21% (of a pool of roughly 6,000) when considering only puzzles set by NS (Nancy Sibtain). Its emphasis on literal interpretation of the clues serves to complement the human approach of reading straight into the underlying subtext. Apart from its practical application, the tool is an experiment into how the complexities of the human language can be organised into a machine-interpretable manner.

Contents

1 Introduction **1**

2 Project Plan **2**

2.1 Overview . . . 2

2.1.1 Functional Requirements . . . 2

2.1.2 Non-Functional Requirements . . . 3

2.1.3 Objectives . . . 3

2.1.4 Constraints . . . 3

2.2 Risk Analysis . . . 3

2.3 Resource Requirements . . . 4

2.3.1 Hardware Requirements . . . 4

2.3.2 Software Requirements . . . 4

2.4 Project Components . . . 5

2.5 Schedule . . . 6

3 External Design **7**

3.1 Production and Deployment . . . 7

3.2 Parameters . . . 8

3.3 User Interface . . . 9

3.3.1 Graphic User Interface . . . 9

3.3.2 Console Interface . . . 10

3.4 Statistics Output . . . 13

3.5 Functionality . . . 13

3.6 Performance . . . 14

4 Internal Design **15**

5 Software Architecture **15**

6 Interpretation and Analysis **16**

6.1 Wordnet Wrapper Module . . . 16

6.2 Wordplay . . . 17

6.2.1 Anagrams . . . 17

6.2.2 Runs . . . 17

6.2.3 Double Definitions . . . 18

6.2.4 Charades . . . 18

6.2.5 Initials and Finals . . . 18

7 Conclusion **19**
7.1 Discussion/limitations 19
7.2 Future work 19

8 References **21**

1 Introduction

A crossword is a lattice-shaped puzzle where words are entered horizontally and vertically, such that they may cross at common letters. It is the crossword solver's task to determine these words, based on a set of clues provided; one clue per word in the puzzle. Supplementary hints to deciphering the clues are gained from determining the letters on word intersections. A variation on this is the 'cryptic crossword'. It challenges one's wit rather than the vastness of their general knowledge (though the latter is still extremely useful), replacing the arguably bland definition/synonym clues of a normal crossword with 'cryptic clues', which require considerably more thought and a degree of lateral thinking to decipher. For example, take:

Petition for phosphorous pea in Alabama. (6)

This clue initially seems to have GMO connotations, but moving past the literal interpretation, we see two parts to it: 'Petition', which serves as the definition for our word, and 'phosphorous pea in Alabama', which comprises the wordplay, with 'for' acting merely as a filler word. The wordplay part employs various tricks to conceal the word. In this case, only the letters of the acronyms for 'phosphorous' (P) and 'Alabama' (AL) are important. The keyword 'in' instructs the solver to take anything immediately preceding it (PPEA) and insert it within what immediately follows (AL), yielding the word APPEAL, which is a synonym for 'petition'. Cryptic clues can take on many forms, though the most common ones incorporate this above definition-wordplay breakdown.

Cryptic Clue Solver (abbreviated to CCS for short), is a tool designed as an aide to solving cryptic crosswords. It can solve most simple clue types independently and comes packaged with a neat interface to simplify its use. Although it is not a pioneering tool in this field, the competition is not fierce, with only a small handful of automated cryptic clue solvers having been published online [2] [3]. Ideally, CCS will bring some new angles to the table.

2 Project Plan

2.1 Overview

The aim is for CCS to be capable of solving at least a subset of existing wordplay patterns. It is to come coupled with a simple yet intuitive user interface for entering the clues, along with various hints from the user, such as known letters, or a suggested wordplay pattern.

2.1.1 Functional Requirements

All clues are assumed to be of a two-part form; a definition and wordplay (order to be determined). The two respective parts are always contiguous and do not overlap, though they may be separated by filler words, as given in the example earlier. Cryptic clues that do not take on this form, such as puns, need not be considered. CCS must be capable of solving the following wordplay patterns with at least a fifty percent success rate:

- Double definitions - These replace the wordplay part with a second definition, though it will often be considerably more abstract; playing on unexpected or rarely used connotations. For example 'flower' might suggest something that flows, rather than colourful flora. [4]
  eg. 'Exuviate garage. (4)' = SHED
- Anagrams - These require shuffling the letters of one or more words in the wordplay to obtain the solution. They are often marked by keywords like 'scramble', 'shuffle' or by less obvious words like 'cook' or even 'up'.
  eg. 'Old man confused nut. (6)' = ALMOND
- Runs - These wordplays contain the solution within them, with the letters already in the correct order and contiguous, spaces excepted. They can be marked by keywords like 'in' or 'has'.
  eg. 'Punch a Rio Tinto official; find vehicle. (7)' = CHARIOT
- Charades - These break up the solution into sub-words, which the wordplay part then defines separately. These 'sub-definitions' are sometimes interjected by keywords like 'then', 'following' or 'preceding'.
  eg. 'House animal after carriage matting. (6)' = CARPET
- Codes - These are similar to runs, but forfeit contiguity for more descriptive keywords, which may instruct to take the first letter of every word, or similar patterns. Codes employ keywords like 'initial', 'ultimate' or 'headless'.
  eg. 'Odd soccer needs result in fusses. (6)' = SCENES

In addition, the interface must provide the user with a means of entering:

- Both clues and their word lengths (if known).
- Additional optional hints such as the wordplay patterns involved or known letters in the answer, which must be adhered to when searching for the solution.

2.1.2 Non-Functional Requirements

In the case of clues requiring brute force to decipher due to unresolved keywords (and by extension ambiguous wordplay patterns), CCS should attempt to fit each wordplay pattern template in the most efficient order, be it by 'hunches' gained from some of the words appearing as though they might be disguised as keywords, or by successively attempting to fit patterns in order of their time-complexity.

2.1.3 Objectives

The primary objective is to develop a piece of software which adheres to all of the minimum requirements outlined above. The idealistic objective is to expand it to handle additional wordplay patterns (such as reversals, abbreviations, and common foreign words), to further improve the user interface, and to also have it solving entire crosswords automatically by taking advantage of the 'known letters' hint functionality, which will have already been implemented as it is a minimum requirement.

2.1.4 Constraints

The constraints on this project are:

- The deadline - This limits how much of the aforementioned additional functionality can be implemented.
- Designing CCS to work as an offline (effectively standalone) tool - Internet access would grant considerably more power through having the tool hook into online crossword solver sites in order to better resolve multi-word definitions; however this would incur a significant slowdown in solving time.

2.2 Risk Analysis

Table 1 summarises the risks that were initially identified. An extension of the final deadline was granted, without which packaging of the final tool would have been somewhat rushed, and the report unpolished. Also, although limited vocabulary breadth was not an issue, the connections between word senses proved to not be as comprehensive as had been anticipated, resulting in some unavoidable shortcomings of the tool. For example the words 'dated' and 'obsolete' are not linked at all in the database, other than through an abstract 'root' construct connecting all verbs.

Table 1: Risk analysis matrix

Risk	Probability	Reduction Strategy
Failure to meet deadlines	Low	Devise and adhere to a project timeline using smaller weekly checkpoints that are easier to meet. Incorporate surplus slack time in case of slowdowns due to unforeseen circumstances.
Loss of progress due to fallible storage media	Very Low	Perform regular backups of work to the cloud.
Software compatibility issues due to library updates	Low	Use well-supported libraries and interface with them at a high level (less likely to be affected by change).
Dictionary locality issues	Moderate	Employ a dictionary containing all word spellings.
Limited vocabulary breadth	High	Employ a large database of words (within reason). This comes as a trade-off with speed.
Bugs in code slipping under the radar/insufficient validation	Moderate	Incorporate strict unit and integration testing schemes along with real-user acceptance tests.
Personal injury to eyes, neck, back or hands due to excessive time at computer	Low	Ensure appropriate workstation conditions (desk height, screen height, comfortable seating), distribute the workload evenly over the available time period and take regular breaks.

2.3 Resource Requirements

2.3.1 Hardware Requirements

The following are the recommended minimum system and hardware requirements:

- Microsoft Windows XP or later
- Intel Dual Core 2.50 GHz processor
- 4 GB RAM
- 500 MB disk space

2.3.2 Software Requirements

The following are the software requirements for this project's development:

- Dropbox - For automated real-time backup of all work to the cloud
- Python 3.4 - Programming language [5]
- PyCharm (Community Edition) - Intergrated development environment [6]

- PyQt4 - Set of Python bindings for Qt UI development framework [7]
- QTDesigner - For designing user interface [8]
- cx_Freeze - For building executable [9]
- Wordnet - Library of word associations [10]
- Natural Language Toolkit (NLTK) - Wordnet interface for Python [11]
- Inflect - Inflection library for python [12]
- NumPy - Numeric library, for manipulating 2D arrays [13]
- SMH Cryptic Crossword Archive - For evaluating CCS performance [1]

2.4 Project Components

Table 2 outlines the initial project breakdown. It is not too dissimilar from the actual tasks that ended up being completed, though implementation of entire-crossword solving capability was regrettably omitted due to time constraints.

Table 2: Initial Work Breakdown Structure

ID	Task Name	Predecessors
1	Write Project Specifications	-
2	Setup Python 3.4 and Spyder IDE	-
3	Install dependencies	2
4	Design basic performance testing suite	1, 3
5	Implement all classes as decided in project specifications	1, 3
6	Implement basic console-based UI	5
7	Develop first intergration tests	5
8	Run integration tests and some real-user acceptance tests	4, 6, 7
9	Revise and reorganise class structure if necessary	8
10	Develop unit tests	9
11	Expand upon UI - convert to a more sophisticated windowed GUI	9
12	Expand functionality with additional wordplay recognition	9
13	Implement automated entire-crossword solving capability	9
14	Write more unit and integration tests to cover new functionality	11, 12, 13
15	Perform regression testing - rerun all earlier integration and unit tests	10, 14
16	Run more acceptance testing with real users	11, 12, 13
17	Finalise and polish solution based on test feedback and results	15, 16
18	Write up documentation (user guide, report, presentation)	17
19	Package and release	18

2.5 Schedule

A Gantt chart has been included in the appendix A1, illustrating the project schedule.

3 External Design

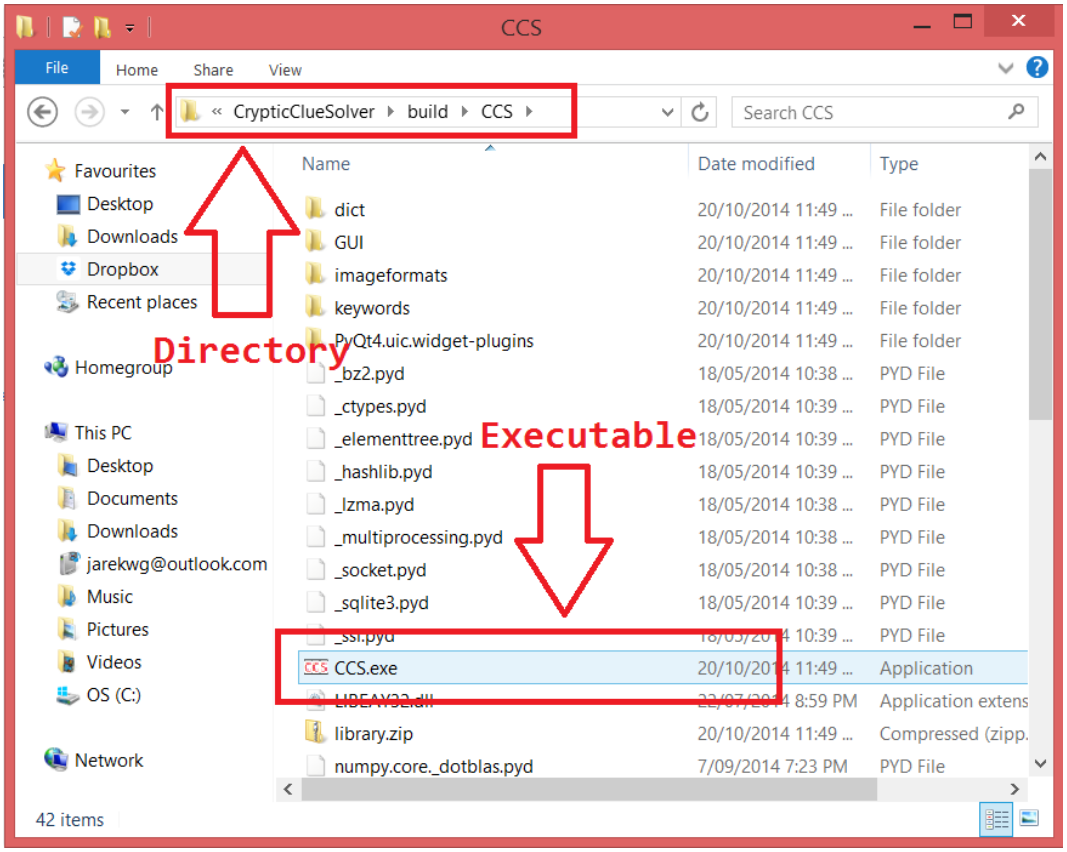
CCS is accessed primarily through its graphic user interface, though the source code may also be utilisied to run the tool from the command line.

3.1 Production and Deployment

CCS comes pre-built and self-contained. A Windows executable and all of its required files and dependencies exist within the 'build/CCS' directory, which can be copied to one's local computer and used separately from the remainder of the source code. This directory size is approximately 122MB, which can be reduced to less than 40MB when compressed, making the tool highly portable.

To run CCS, simply navigate to 'build/CCS/' and execute the file 'CCS.exe', as shown in Figure 1. Nothing else is required. Python need not be installed on the host computer, nor does WordNet.

Figure 1: CCS executable location



The source code has been tested to run both in Linux and Windows, with the following dependencies required:

- Python 3.4 (32-bit version recommended) [5]

- Natural Language Toolkit (NLTK) - Wordnet interface for Python (note that WordNet itself need not be installed as it is included within CCS) [11]
- Inflect - Inflection library for python [12]
- NumPy - Numeric library, for manipulating 2D arrays [13]
- PyQt4 (can be omitted if not running the GUI) [7]

In windows, Python 3.4 and the other library dependencies can all be installed by downloading and running the relevant binaries.

For Linux, Python 3.4 comes pre-installed on most recent distributions (invokable by entering 'python3' from the terminal). If not installed, run 'sudo apt-get install python3.4'. The remaining libraries (PyQt4 excepted) can be installed trivially with 'sudo pip3 install <packagename>' (pip3 should also come pre-installed. If not, download it first with 'sudo apt-get install python3-pip'). Installation of PyQt4 is required only to be able to open the GUI from the command line. Its installation is less trivial, but can be done by following the instructions here: http://ubuntuforums.org/showthread.php?t=1777613 [14]

With the above installed, CCS can be interfaced from the python console via the `ClueParser` class. More on this is discussed in the User Interface section.

A fresh build can be also performed in Windows by executing the 'make_exe.bat' file in the top directory. However to do this, all of the aforementioned packages need to first be installed, along with a patched version of cx_Freeze [9] (downloaded from here: http://www.lfd.uci.edu/ gohlke/pythonlibs/ [15], with the modification as outlined here: https://bitbucket.org/anthony_tuininga/cx_freeze/pull-request/56/catch-error-if-getdependentfiles-is-called/diff [16]). The official version provided on the cx_Freeze website has some bugs, hence these workarounds are required.

3.2 Parameters

CCS has several parameters that it accepts and this customisation is mirrored by the GUI. A screenshot of the GUI is shown in Figure 2 in the next section. The primary field is the input box at the top for entering the clue. The remaining inputs are all optionally settable and include:

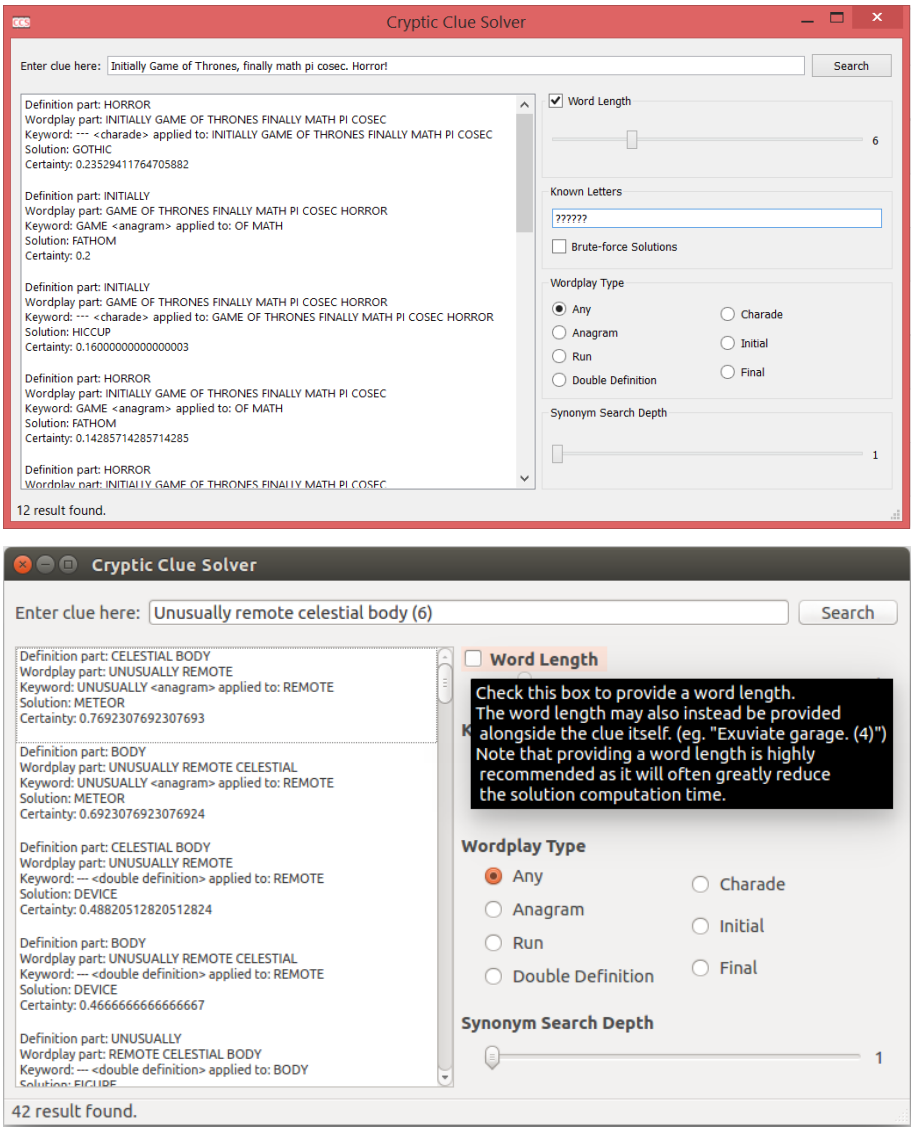
- A slider - for providing the word length. This is limited to the range 3-15, as those are the expected lengths for most cryptic crosswords. However this can still be circumvented if need be (more on this later).
- A check box - for enabling/disabling word length control. When disabled, CCS will assume any word length to be acceptable.
- An input box - for entering known letters. This has no length limit, however the input is restricted to alpha characters and question marks.
- A check box - for toggling brute-force mode. More on this later.

- A set of radio buttons - for selecting a specific wordplay type to be used.
- A slider - to provide the option of deeper synonym searches. This is limited to the range of 1-5, since 0 would provide no synonyms at all apart from the original word, and 6 would yield results so dissimilar to the original word that they would not be worth considering.
- A button - for submitting the clue.

3.3 User Interface

3.3.1 Graphic User Interface

Figure 2: CCS User Interface in Windows (above) and Linux (below)



The GUI employs a simplistic layout, keeping options to a minimum, so as to not overwhelm the user. In most cases of potential customisability it was decided that the added options would lead only to confusion and not grant any true benefit to the user. The most customisable aspect is the 'Synonym Search Depth' slider, and even its necessity to exist in the GUI is questionable.

Considerable effort has been put towards making every aspect of this interface as intuitive as possible. Features include:

- Comprehensive tooltips - Hovering the mouse cursor over any input element of the user interface will bring up text explaining the element's functionality and possibly a usage example.
- Window resize support - All form elements maintain sensible aspect ratios regardless of what the window is resized to.
- Automatic enforcement of consistency between form elements - If the user supplies a word length with the slider, this is automatically reflected in the 'known letters' field, which gets populated with the corresponding number of question marks. Similarly, if any known letters are provided, the word length slider automatically adjusts its position to match the number of known letters given.
- Input masking - The 'known letters' field dynamically capitalises any user input and prevents entering of any non-alpha non-question-mark characters.
- Undo support - Using 'ctrl+z' while a text field is in focus will step backwards through the input history of that field.
- Error message boxes - If the user makes an unsound request, such as providing different word lengths via both the clue input field and via the slider, or requesting a brute-force of solutions without populating the 'known letters' field, an informative message dialog will pop up and explain where the user has gone wrong.
- Dynamic runtime feedback - A status bar at the bottom of the form keeps the user informed of which stage processing is at, along with displaying the number of results found once processing completes.
- Informative solutions - The list of results provides information as to how each solution was found, breaking down the clue and explaining wordplays applied, along with providing a certainty score.

3.3.2 Console Interface

CCS can be interfaced from the console by importing the `ClueParser` class from the `clue_parser` module, creating a `ClueParser` instance, then submitting clues via the `ClueParser` instance's `.parseClue()` method. The GUI can also be started up from the console by simply importing the `CCS` module and calling `CCS.run()` The following code illustrates some examples of this console-based functionality.

```
# Import the ClueParser class.
from clue_parser import ClueParser
```

```
# Create a ClueParser instance.
cp = ClueParser()

# Submit a cryptic clue for processing.
cp.parseClue('First male orphan on Io.')

# Note the different formats in which a clue may be submitted.
cp.parseClue('First male orphan on Io. (4)')
cp.parseClue('First male orphan on Io.', 4)
cp.parseClue('First male orphan on Io.', length=4)
cp.parseClue('First male orphan on Io.', typ='charade')
cp.parseClue('First male orphan on Io.', known_letters='m?o?')

# Obviously the more information given, the faster the clue will be processed,
#  and the more accurate the solution pool will be.
# Note that repeated information is unnecessary, but will still be accepted unless
#  there exist contradictions.
cp.parseClue('First male orphan on Io. (4)', 4, known_letters='m?o?') # acceptable
cp.parseClue('First male orphan on Io. (3)', 4) # error
cp.parseClue('First male orphan on Io. (3)', known_letters='m?o?') # error
cp.parseClue('First male orphan on Io.', length=3, known_letters='m?o?') # error

# Another alternative is to create a Clue object first.
from clue import Clue

# Note the use of double quotes this time to escape the apostrophe in the clue.
c = Clue("Zebra 'n friends lost confidence.")

# As with ClueParser.parseClue(), the Clue constructor allows various
#  input formats too.
c = Clue("Zebra 'n friends lost confidence. (6)")
c = Clue("Zebra 'n friends lost confidence.", 6, typ='anagram')
# ..etc

# This can then be followed up with calling .parseClue with the
#  Clue object as first parameter.
cp.parseClue(c)
cp.parseClue(c, 6)
cp.parseClue(c, typ='anagram', known_letters='?ra??n')
# ..etc
```

```
# The output is simply a list of solution objects.
solns = cp.parseClue(c)
print(solns) # This will print all solutions to the console.
print(solns[0:5]) # This will print only the top 5 solutions.

# Lastly, the GUI can be started up by running the following lines.
# Requires PyQt4 installed.
import CCS
CCS.run()
```

The output for console-based requests is automatically formatted neatly when printed. As in the GUI, the methodology behind arriving at each solution is also provided. This is illustrated below.

```
>>> cp.parseClue('Study confuses Marc.')[0:5]
2014-10-21 12:22:48,753 - INFO - clue_parser - Parsing clue: Study confuses Marc.
2014-10-21 12:22:50,997 - INFO - clue_parser - 6533 solution(s) found. \
  Best solution is 'cram' (certainty: 0.800000)
[===============
Definition part: STUDY
Wordplay part: CONFUSES MARC
Keyword: CONFUSES <anagram> applied to: MARC
Solution: CRAM
Certainty: 0.8
===============
, ===============
Definition part: STUDY
Wordplay part: CONFUSES MARC
Keyword: —— <double definition> applied to: CONFUSES
Solution: PIECE
Certainty: 0.6725274725274726
===============
, ===============
Definition part: STUDY
Wordplay part: CONFUSES MARC
Keyword: —— <double definition> applied to: CONFUSES
Solution: BLUR
Certainty: 0.6071428571428571
===============
, ===============
```

```
Definition part: STUDY
Wordplay part: CONFUSES MARC
Keyword: —— <double definition> applied to: CONFUSES
Solution: FOX
Certainty: 0.584375
===============
, ===============
Definition part: STUDY
Wordplay part: CONFUSES MARC
Keyword: —— <double definition> applied to: CONFUSES
Solution: JUMBLE
Certainty: 0.5666666666666667
===============]
```

3.4 Statistics Output

CCS does not deal with any statistical data, other than tallying up the number of solutions generated for a given request. However the test suite does include a performance test which statistically weighs up the solving capability of the tool (more on this in the test report).

3.5 Functionality

The GUI contains some additional, less obvious functionality, which has been intentionally concealed so as not to subtract from the tool's straightforward interface. However this is information is still readily surrendered through the tooltips. The functionality includes:

- Circumventing the word length limits imposed by the word length slider - The slider is purposely set to a range of 3-15, as this is sufficient for most cases. However the user may instead provide a word length directly through the clue input box by placing it in parentheses at the end of the clue, or by entering the corresponding number of characters in the 'known letters' field.
- Generating synonyms - The program can be instructed to print all synonyms of a given word by entering that word twice in the search box, and selecting the 'double definition' wordplay type. This is not an intended feature (else it would be more appropriately supported), however it works due to the way double definitions are processed.

When running the source code from the console, even more functionality is made available through the `wordnet` and `wordplay` modules, including synonym generation, word similarity scoring, abbreviation generation, and word pluralisation. Refer to the source code in wordnet.py for a detailed rundown of the available functions.

Logging functionality is also included; refer to the Test Report for details.

3.6 Performance

CCS prefers to trade off space for time where possible. A high speed is attained through pre-compiling certain aspects of the tool, such as a dictionary of all anagrams, and through caching the results to certain functions, such as synonyms of a word. All of this does however result in a huge 300MB-700MB of RAM usage during runtime. Table 3 lists the time and space complexities of certain functions within CCS, along with their average runtime. Times have been measured with a tool from the test suite (more on this in the test report), and complexities will be analysed in more depth in Section 6.

Table 3: Function Complexities and Runtimes

module	function	Time Complexity	Space Complexity	Runtime
wordnet.py	exists	$O(1)$	$O(1)$	$< 1\mu s$
wordnet.py	calcSimilarity	$O(synset\ size^2)$	$O(synset\ size)$	$\approx 0.25ms$[1]
wordnet.py	getSynonyms	$O(synset\ size^{search\ depth})$	$O(synset\ size^{search\ depth})$	$\approx 2\mu s$[1][2]
wordnet.py	getAbbreviations	$O(1)$	$O(1)$	$< 1\mu s$
wordnet.py	getWordsWithPattern	$O(wordlist\ size)$	$O(1)$	$\approx 0.5s$
wordplay.py	getAnagrams[3]	$O(n \log n)$	$O(n)$	$\approx 1\mu s$

[1] - When query result not in cache, else $O(1)$ lookup for both space and time complexities.

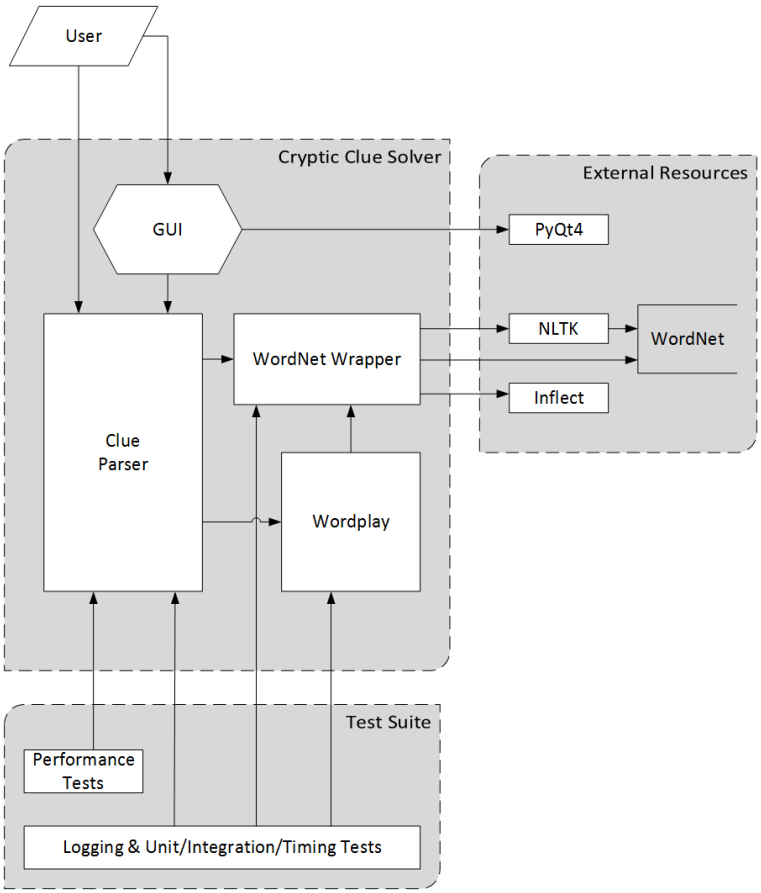
[2] - Running at a search depth of 1 (most common).

[3] - Name is different in module; simplified name used here for clarity.

4 Internal Design

Figure 3 depicts the overarching topology of CCS, its dependencies and test suite.

Figure 3: Abstracted program topology



5 Software Architecture

A UML class diagram has been included in the appendix A2, showing an in-depth topology of the class structure and hierarchy. As is evident from this diagram, CCS is centred around the `ClueParser` class and this is the only class with which the GUI communicates. `ClueParser` oversees the entire workflow, from a clue's submission to the solutions being returned. The other two major components are:

- The `wordnet` module, which wraps around the NLTK interface, providing a clean way of accessing all of the desired functionality, along with optimising efficiency by either caching results of requests made to the WordNet database, or bypassing the database entirely and accessing its resource files directly.
- The `Wordplay` subclasses, which each contain their own set of keywords (if applicable), possible custom wordlist structures for fast lookup, and solving algorithms for handling their respective wordplay types.

Some auxiliary data structures are also used for containing the clues and solutions. These are much cleaner than, say, nested dictionaries, and also provide the option of expanding them with additional functionality. This is the case for the Clue class, whose instances not only store clues, but also dissect them into their constituent parts on init, along with providing the option of checking possible solutions against them.

6 Interpretation and Analysis

This section aims mainly to provide a detailed insight into specific elements of CCS, the research involved in their development, along with possible strengths and shortcomings.

Overall, the research conducted prior to commencing work on CCS primarily involved reading articles on how to solve cryptic clues by hand [4] and downloading a cryptic crossword phone application [17] to gain a better familiarisation with the pattern types. However, little research was conducted into existing automated solvers other than a brief read through Robin Deits' blog post "A Cryptic Clue Solver" [18]. This was because the aim behind CCS was to attempt to possibly bring some new angles to the field; with referring to methodologies of its predecessors, it would have been difficult not to end up walking the same tracks.

6.1 Wordnet Wrapper Module

The WordNet [10] database contains almost every word in the English language, and accepts spellings from all localities. However its most interesting and useful functionality is the organisation of this word pool into collections referred to as synsets, or sets of synonyms. This is the vital functionality that CCS relies on in generating solutions to the definition parts of clues, along with scoring word pair similarities. However, as convenient as WordNet is to use, it does contain numerous pitfalls, which were discovered during the development of CCS, and needed to be remedied for progress to continue:

- Stop words not included: They have no place in WordNet's synset structure. However since the plan was for CCS to use WordNet for looking up word existence, a list of stop words needed to be imported from an external source [19].
- Slow word existence checks: CCS needs to check words for existence in $O(1)$ time. WordNet was far too slow for this task, so a function was written to compile a dedicated word list, bypassing the WordNet interface. This word list is stored in two forms in local memory during program runtime - a set structure, which hashes all elements ensuring constant average lookup time, and a sorted list structure, allowing for fast binary lookup of words starting with a given prefix. Compilation of this word list was non-trivial, as WordNet's resource files contain only the base, uninflected forms of words. Hence the python package Inflect [12] was used to re-generate these inflections, along with a comprehensive word list from an external source (already containing inflected forms) being added to pad out the list [20].
- Lack of commutativity in similarity scorer: As an example, WordNet scores the similarity between 'dated' and 'obsolete' as 0.4, but that of 'obsolete' and 'dated' as 0. This is due to the fact that, as mentioned earlier, WordNet does not have these two words linked in its database, and thus can only

link them through an abstract 'root' node construct, which by default it will use for some words but not others. To remedy this, the similarity request is performed in both directions and the larger score taken.

- No direct way to compare two words: WordNet only scores the similarity between two word *senses*. Regrettably this means that the full synset of one word must be compared pair-by-pair against the full synset of the other. This results in $O(n^2)$ (where n is the synset size) time complexity, and is the primary cause of slow computation of many clues by CCS.
- No synonym generation function: Although the WordNet database is organised into synsets, these alone will not yield comprehensive synonym lists. Hence CCS uses a combination of synset(), hypernym(), hyponym(), similar_to() and lemma() calls to WordNet to achieve the desired effect. This takes still only $O(n)$ time. If even more synonyms are desired the search depth can be increased, which repeats the process on each result from the previous run, at the cost of raising the time complexity to $O(n^s)$, where s is the synonym search depth.

6.2 Wordplay

Considerable research and experimentation was conducted during development of the Wordplay classes. They run quite efficiently for the most part, however there still remain instances where processing is quite slow. The main strategy used is to have the wordplay part generate all possible words, and then to score each of these against the definition part of the clue. This is optimal for Anagrams, Runs and Initials/Finals, but consumes considerable time for the wordplay types that generate huge pools of possible answers. Currently this issue has been ignored, so as to maintain consistency across all wordplay types.

Each wordaplay class also employs its own scoring mechanism to rate all output solutions based on how well they have used the given wordplay.

6.2.1 Anagrams

The naive approach to anagram generation is permutation of the letters in the input string followed by a check to see whether each permutation is a valid word. This takes $O(n!)$ time, n being the string length. To avoid this lengthy computation, the `AnagramWordplay` class compiles its own version of the word list, grouping words that share the same letters under a key that is the sorted set of those letters. For example:

`'aaegmnt'`: {`'magnate'`, `'magenta'`}

`'aelpst'`: {`'plates'`, `'septal'`, `'pastel'`, `'petals'`, `'staple'`, `'pleats'`, `'tepals'`}

Then, when an anagram of a set of letters is requested, those letters need only be sorted ($O(n \log n)$) and used as a key to lookup ($O(1)$) all corresponding anagrams. This comes at the cost of storing this additional form of the word list in RAM.

Solutions are scored based on how many of words in the wordplay have been used to create the anagram.

6.2.2 Runs

The slow approach here would be to use nested looping, taking $O(n^2)$ time, like this:

```
runs=[]
strlen = len(string)
# For each possible starting point in string.
for i in range(strlen):
        # For each possible finishing point in string.
        for j in range(i, strlen):
                run = string[i:j+1]
                if wordnet.exists(run):
                        runs.append(run)
```

The `RunWordplay` class hence also compiles its own version of the word list, storing all words in a tree structure, with a letter at each node. It then performs the same algorithm as above, but traverses the tree structure in parallel to running the inner loop. This allows for dead-end detection, so that the inner loop can be terminated early. Since words in the word list are of finite length, this reduces the complexity to $O(n)$.

Solutions are scored based on how many constituent words they pass through, of those available in the wordplay.

6.2.3 Double Definitions

These simply use `wordnet` to generate synonyms of words in the wordplay.

Solutions are scored based on how many of the words in the wordplay have been used to create the synonym, with scores multiplied by the similarity between the generated word and its the definition that generated it.

6.2.4 Charades

This is the most complex of the wordplay classes. It first splits the wordplay part into all possible 'halves'. A pool of possible 'solution first halves' are then generated from the first half of the wordplay. These can be synonyms, abbreviations, and even initials and finals (provided there is a keyword that indicates this). Each of these first halves are then run through the wordlist, generating all possible 'solution second halves'. Of these, the ones that are words, or abbreviations/initials/finals of the second half of the wordplay, are found, then the corresponding full solutions returned.

Solution scores are based on the product of the scores of the constituent halves, again using synonym similarity, or in the case of abbreviations/initials/finals, given a maximum score of 1.0.

6.2.5 Initials and Finals

These take the first/last letter of each word or a subset of words in the wordplay, then check if the result exists in the dictionary.

Solutions are scored based on how many words of the wordplay were used.

7 Conclusion

In conclusion this project has been successful, with most goals achieved and CCS exhibiting impressive performance statistics. It has been a great learning experience, and hopefully more can be learned from possible future development of this tool.

7.1 Discussion/limitations

Though CCS shows great promise when exposed to the simpler cryptic clue types, in its current state it is little more than a pattern recognition algorithm mounted atop a word base too vast for it to comprehend. CCS's current success is largely due to its speed in essentially brute-forcing solutions through abbreviation/synonym searches along with looking up runs, anagrams, and codes faster than a human could, with an arguably superior vocabulary. However these are all pre-determined patterns hard-coded into the tool and though they may provide elements of leeway such that they may be applicable to as many variations on a wordplay type as possible, there exists no 'thinking outside the box'. CCS has a limited skill set, and even if more skills were to be added to it in the future, the result would remain a limited skill set. Cryptic clues are by their very nature designed to require 'thinking outside the box' and thus any *true* clue should not be expected to be solvable by anything less than a cognitive computer, like IBM's much revered Jeopardy-solving supercomputer - WATSON [21].

7.2 Future work

The more immediate limitations of this tool that could be addressed in the future include:

- Classification of words in the clue such that their importance can be determined:
  Singling out parts of speech is not of the highest priority when interpreting cryptic clues as it can be potentially detrimental to limit a synonym pool by assuming the sense of a given word; however provided this trap is avoided, such knowledge can still be useful. Currently CCS may consider any word in the clue as a 'filler' word. This leads to bogus solutions when an 'important' word is skipped over, and also means that the wordplay scoring can at times be suboptimal, since all tokens are weighted equally even though stop words should in most cases be omissible without penalty.
  Word classification would also allow synonym generation and similarity scoring to take inflections into account. Currently CCS recognises only plurality.

- Better multi-word and hyphenated-word support:
  CCS's word list already includes all of the multi-word phrases recognised by WordNet, allowing definitions to be comprised of up to three words, along with the tool being able to provide multi-word solutions. However there remain numerous bottlenecks within the tool where only single words are considered. The main two examples of this are wordplay keywords (only single-word keywords are recognised), and allowable word lengths (ie. the user needs to provide the solution length as '(8)' rather than '(5-3)' or '(5,3)').
  There also exists this grey area of hyphenated words, which are currently treated as single words, with the hyphen automatically removed. Proper support for multiple and hyphenated words is thus

certainly an area for improvement, though its implementation will also come with an increased reliance on a comprehensive word list.

- More pre-computation to increase speed:
  Much like CCS uses custom anagram and run word lists, there need to exist additional custom structures for faster generation of synonyms and computation of word similarities. The problem with WordNet's database structure is that it can only return synonyms and compute similarities for given senses of words. Hence CCS is forced to loop through each word sense to assimilate its required data. Devising a custom word network more applicable to this context would allow for much faster lookup times, and possibly not even be at the cost of significantly increased memory usage.

- Improvement of existing wordplay types:
  The existing wordplay classes can solve many clues of their respective type, however there is still plenty of room for improvement. This is especially true for charades, which currently only consider 2-word splits of the solution, rather than arbitrary numbers of splits.

- Additional wordplay types:
  The intention was for reversals to be included within this release, but due to time constraints, the only additions above the minimum requirements are abbreviations and nested wordplays. Most time was spent optimising existing wordplay parts. Implementing wordplays like reversals, containers, deletions and homophones would complete the set of patterns that an automated solver could look for, resulting in greatly increased clue-solving success rates.

- Expansion of word list with a larger word-relation database:
  Foreign words and names of well known people/places would also be a good addition for CCS. As was mentioned earlier, WordNet would not recognise these, however such an expansion could include doing away with WordNet to opt for a larger, perhaps custom-built, word-relation database.

- Entire crossword solving capability:
  Since the 'known letters' functionality already exists, ramping this tool up to the stage of solving entire crosswords would not be difficult, and would provide a nice addition. The main requirements here would be to decide on the input format(s), along with expanding the GUI to be able to visually present the problem and provide maximal interactivity. This functionality was initially planned, but again, due to time constraints, ended up not being realised.

- Access to auxiliary functions from the GUI:
  CCS includes a wealth of useful auxiliary methods, whose functionality may occasionally be desired. These include: word existence checking, synonym generation, anagram generation, pluralisation, abbreviation lookup, and word pair similarity computation. Making these accessible from the GUI via a file menu of some sort (so as to not complicate the main interface) would be a useful and simple addition.

8 References

[1] The DA Trippers. *The Motherlode: Crosswords from Way Back*. 2010. URL: http://datrippers.com/2010/01/22/the-motherlode-crosswords-from-way-back/ (visited on 08/28/2014).

[2] Robin Deits. *Cryptic Crossword Clue Solver*. Dec. 26, 2013. URL: http://cryptic-solver.appspot.com/ (visited on 10/25/2014).

[3] Crossword Tools. *Clue Solver*. 2013. URL: http://www.crosswordtools.com/cm/ (visited on 10/25/2014).

[4] Denise Sutherland. "Diving Into the world of Cryptic Crosswords". In: *Solving Cryptic Crosswords For Dummies*. Wiley, 2012, pp. 12–15.

[5] Python Software Foundation. *Python 3.4.1 documentation*. 2014. URL: https://docs.python.org/3/ (visited on 08/25/2014).

[6] JetBrains. *PyCharm*. 2014. URL: https://www.jetbrains.com/pycharm/ (visited on 10/21/2014).

[7] Riverbank Computing. *PyQt4*. Sept. 11, 2014. URL: http://www.riverbankcomputing.com/software/pyqt/intro (visited on 10/21/2014).

[8] Digia. *QtDesigner*. 2014. URL: http://www.qt.io/product/ (visited on 10/21/2014).

[9] Anthony Tuininga. *cx_Freeze*. 2014. URL: http://cx-freeze.readthedocs.org/en/latest/ (visited on 10/21/2014).

[10] Christiane Fellbaum. "WordNet and wordnets". In: *Encyclopedia of Language and Linguistics*. Ed. by Keith Brown. 2nd. Elsevier.

[11] Steven Bird, Edward Loper, and Ewan Klein. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

[12] Paul Dyson and Damian Conway. *inflect.py*. June 9, 2013. URL: https://pypi.python.org/pypi/inflect (visited on 10/20/2014).

[13] Stefan Van der Walt, S. Chris Colbert, and Gael Varoquaux. *The NumPy Array: A Structure for Efficient Numerical Computation*. IEEE Computer Society, 2011.

[14] neurobot. *How to set up PyQt4 for python 3.2 in Ubuntu 11.04*. URL: http://ubuntuforums.org/showthread.php?t=1777613 (visited on 10/26/2014).

[15] Christoph Gohlke. *Unofficial Windows Binaries for Python Extension Packages*. 2014. URL: http://www.lfd.uci.edu/~gohlke/pythonlibs/ (visited on 10/21/2014).

[16] Thomas Kluyver. *Catch error if GetDependentFiles is called on a non-library*. June 20, 2014. URL: https://bitbucket.org/anthony_tuininga/cx_freeze/pull-request/56/catch-error-if-getdependentfiles-is-called/diff~ (visited on 10/21/2014).

[17] Teazel Ltd. *Crossword Cryptic Lite*. Oct. 23, 2014. URL: https://play.google.com/store/apps/details?id=com.teazel.crossword.cryptic.lite&hl=en (visited on 10/26/2014).

[18] Robin Deits. *A Cryptic Crossword Clue Solver*. 2013. URL: http://blog.robindeits.com/2013/02/11/a-cryptic-crossword-clue-solver/ (visited on 08/26/2014).

[19] Ted Pedersen. *A WordNet Stop List*. URL: http://www.d.umn.edu/~tpederse/Group01/WordNet/wordnet-stoplist.html.

[20] Sil International. *English Wordlists*. 2014. URL: http://www-01.sil.org/linguistics/wordlists/english/wordlist/wordsEn.txt (visited on 10/15/2014).

[21] IBM. *What is Watson?* 2014. URL: http://www.ibm.com/smarterplanet/us/en/ibmwatson/what-is-watson.html (visited on 10/26/2014).

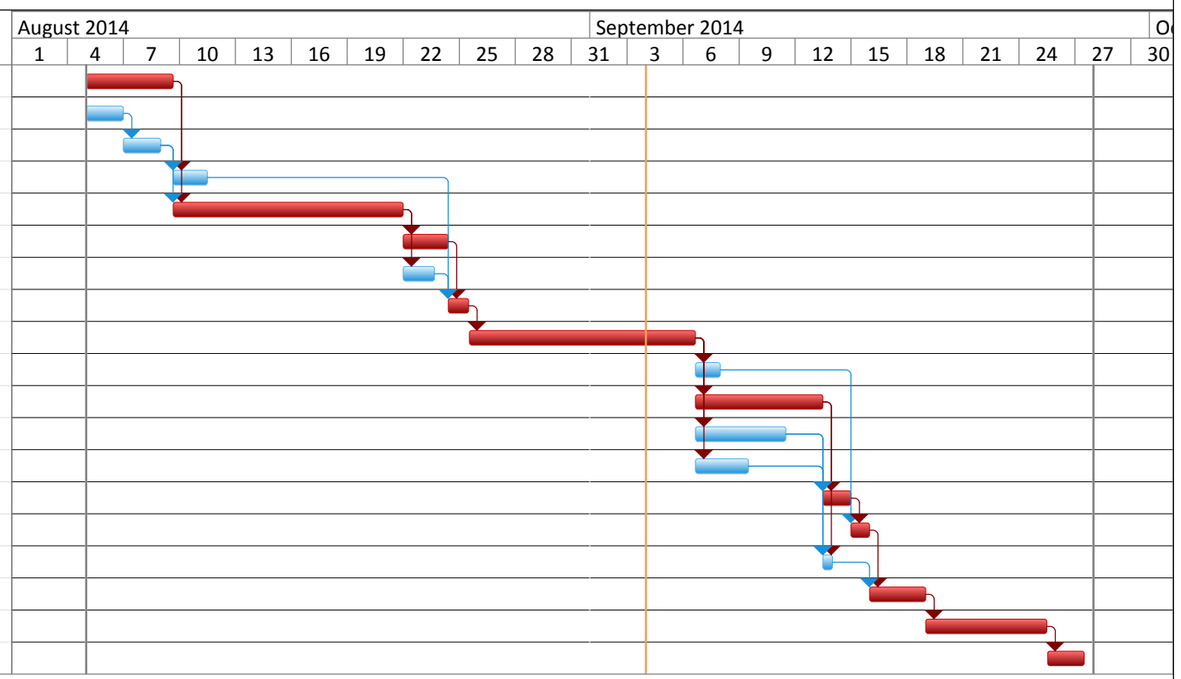
[22] David Hardcastle. *Riddle posed by computer (6): The Computer Generation of Cryptic Crossword Clues*. Tech. rep. Birkbeck, London University, 2007.

[23] P. W. Williams and D. Woodhead. "Computer assisted analysis of cryptic crosswords". In: *The Computer Journal* 22 (1977), pp. 67–70.

A1 - Gantt Chart

Jarek Glowacki - 23392754

ID	Task Name	Duration	Start	Finish	Predecessors
1	Write project specifications	5 days	5/08/14	9/08/14	
2	Setup Python 3.4 and Spyder IDE	2 days	5/08/14	6/08/14	
3	Install depenendencies (NLTK, PyEnchant, NumPy)	2 days	7/08/14	8/08/14	2
4	Design basic performance testing suite	3 days	9/08/14	11/08/14	3,1
5	Implement all classes as decided in project specifications	15 days	9/08/14	21/08/14	3,1
6	Implement basic console-based UI	3 days	22/08/14	24/08/14	5
7	Develop first integration tests	2 days	22/08/14	23/08/14	5
8	Run integration tests and some real-user acceptance tests	2 days	24/08/14	25/08/14	6,7,4
9	Revise and reorganise class structure if necessary	14 days	25/08/14	6/09/14	8
10	Develop unit tests	2 days	6/09/14	7/09/14	9
11	Expand upon UI - convert to a more sophisticated windowed GUI	8 days	6/09/14	13/09/14	9
12	Expand functionality with additional wordplay pattern recognition	6 days	6/09/14	11/09/14	9
13	Implement automated entire-crossword-solving capability	4 days	6/09/14	9/09/14	9
14	Write more unit and integration tests to cover new functionality	3 days	13/09/14	14/09/14	21,11,31
15	Perform regression testing - rerun all earlier integration and unit tests	1 day	15/09/14	15/09/14	01,41
16	Run more acceptance testing with real users	1 day	13/09/14	13/09/14	21,11,31
17	Finalise and polish solution based on test feedback and results	3 days	16/09/14	18/09/14	51,61
18	Write up documentation (user guide, report, presentation)	8 days	19/09/14	25/09/14	71
19	Package and release	2 days	25/09/14	27/09/14	81



Project: Cryptic Crossword Clue Solver
Date: 4/09/2014

Task Critical

A2 – Uml Class Diagram

GUI

CCSMain
-window
-cp : ClueParser
-ccsParseClue : CCSParseClue
-ccsInit : CCSInit
+__init__()
+parseClue()
+updateStatus()
+printSolutions()

CCSThread
-statusUpdate
+__init__()
+updateStatus()

CCSInit
+run()

CCSParseClue
-solutionsToDisplay
+run()
+halt()

CCS

wordnet
-stemmer
-lemmatiser
-pluraliser
-logger : Logger
-WORDLIST : set[string]
-WORDLIST_SORTED : list[string]
-_SIM_CACHE : dict[string]
-_SYN_CACHE : dict[string]
-_ABBREVIATION_LIST : dict[string]
+recompileWordList()
+exists()
+isPlural()
+pluralise()
+literalStem()
+calcSimilarity()
+getSynonyms()
+getAbbreviations()
+getWordsWithPattern()

ClueParser
-clue : Clue
-wordnet : wordnet
-wordplays : list[Wordplay]
-gui_thread : CCSParseClue
-logger : Logger
+__init__()
+parseClue()
+recompileDictionaries()
+recompileWordlist()
-interpretClue()

Clue
-clue : string
-length : int
-typ : string
-known_letters : string
-regex : string
-tokens : list[string]
-token_set : set[string]
-logger
+__init__()
+checkSolution()
+__repr__()

Solution
-solution
-definition
-wordplay
-typ
-applied_to
-certainty
+__init__()
+__str__()
+__repr__()

AnaramWordplay
-hasCustomDict : bool = True
-usesKeywords : bool = True
-__typ__ : string = 'anagram'
+__init__()
+check()
-calcCertainty()
-getPlay()
-addToDictionary()

RunWordplay
-hasCustomDict : bool = True
-usesKeywords : bool = True
-__typ__ : string = 'run'
+__init__()
+check()
-calcCertainty()
-getPlay()
-findTokensUsed()
-addToDictionary()

DoubleDefinitionWordplay
-__typ__ : string = 'double definition'
+__init__()
+check()
-calcCertainty()
-getContiguousCombinations()

CharadeWordplay
-__typ__ : string = 'charade'
+__init__()
+check()
+getPlay()
+getPossibleRights()

InitialWordplay
-usesKeywords : bool = True
-__typ__ : string = 'initial'
+__init__()
+check()
-calcCertainty()
-getPlay()

FinalWordplay
-usesKeywords : bool = True
-__typ__ : string = 'final'
+__init__()
+check()
-calcCertainty()
-getPlay()

Wordplay
-hasCustomDict : bool = False
-usesKeywords : bool = False
-__typ__ : string
-keywords : dict[string] = {}
-logger : Logger
+__init__()
+check()
+recompileDictionary()
-getCombinations()
-getPlay()
-addToDictionary()

WordplaySolution
-solution
-typ
-applied_to
-certainty
+__init__()
+__repr__()

DEBUGGING SUITE

ccs_integration_tests
+run()

ccs_unit_tests
+testClue()
+testWordnet()
+testWordplay()

ccs_performance_tests
+run()

ccs_timing_tests
+run()

Logger
-handler
+debug()
+info()
+warn()
+error()
+critical()

Key
is a subclass of
creates insances of