Part I

Introduction to the Problem / Field

Cryptic crosswords are widely thought to be at the crossroads of various fields of human endevour considered to be right at the limit of current AI and Machine Learning – featuring wit, slang, allusion, linguistic ambiguity and generally deliberate trickery. Along with this, they possess other characteristics that make brute force solutions difficult, if not impossible: the state space of all possible crossword grids is of the order 10^{90}citation needed(compare, for example, an upper bound on all the possible chess positions is merely 10^{50}), and worse still, a solution to a grid is non-trivial to verify (as the verification process is the nearly same as solving the clue!)

Nevertheless, techniques from combinatorics, compiler design and NLP and AI all have applications that can help elucidate and simplify the problem, along with heuristics adapted from both human solvers and analytical optimisations that can help improve the time taken to arrive at at the correct solution.

I have developed a system which can solve a significant percentage of cryptic crossword clues More here, obvs

> "A provost at Eton once boasted that he could do The Times crossword in the time it took his morning egg to boil, prompting one wag to suggest that the school may have been Eton but the egg almost certainly wasn't." – Citation Needed

Contributions Itemize what the contributions are

Part II
Literature Review

1 Summary of Cryptic Crosswords

1.1 Definition of Cryptic Crosswords

A crossword is a puzzle, usually published in newspapers or magazines. They consist of a grid of squares, often 15 x 15. Some of the squares are white (i.e. blank) and some are blacked out. Any contiguous run of more than one white square, either down (vertically) or across (horizontally, left to right) is a space for a word, to be written. These are marked by numbers in the initial square (the top-leftmost one), and referred to by those numbers, and the direction (e.g. '5 down', '8 across'). Horizontal runs can overlap vertical runs, and at the points at which they do, each of the two words, when written in, must have the same letter in that square. Along with the grid are a set of clues, which the solver can use to determine which word to write in each space (the 'answer' or 'solution'). The aim of the puzzle is to find the set of solution words such that each clue's solution is correct for that clue, and fits in the grid correctly, with respect to the overlapping words.

Grids can be very densely white, with few black squares and most squares shared by two words (usually called American Style) or more sparse, with fewer overlapped clues (called British Style). Clues can also be in two styles. Straight or Quick crossword clues usually provide a single straightforward indicator as to what the correct word might be - often a synonym for the clue ('Joyful' = 'Happy') or a missing word ("Stitch in ____ saves nine" = "Time"). Cryptic clues are less straightforward, appearing on the surface to be a valid syntactic utterance in English, but actually consisting of a definition (as in the Straight clue) and some wordplay which the solver can use to arrive at the same answer as with the definition by apply a series of transformations and operations. Insert some example here TFThe challenge is that the definition and the wordplay are not clearly separated, and that there are multiple ways to apply to the transformations, but with only on yielding the correct answer.

It is the task of determining the correct answer for this type of clue that this report will address.

Definitely do some more examples dotted throughout here

1.2 Cryptic Crosswords in the Literature

While not a topic well covered in scientific literature in general, what few analytical studies around cryptic crosswords there are tend to be classifiable into three main groups

Generation of Cryptic Clues The largest body of work that exists is centered around the generation of cryptic clues, focusing largely around analysis of how string literals from a pre-determined answer can be transformed by set clueing patterns, as well as some work around measures of the quality of generated clues.

Interpreting Clues The next set are the select few who have done prior, similar investigations into interpreting cryptic clues, with some work put into formalizing definitions and notation for the sorts of clue types that appear in the majority of cryptic crosswords, and some attempts at solving based on these interpretations.

Other Work There has also been some work aimed at solving non-cryptic crosswords probabilistically, working on whole-grid solutions rather than individual clues. A number of left-of-field studies have also been undertaken, e.g. statistical studies into errors made during manual solving, and psychological studies into solving. More here TF

1.3 Complexity

A variety of factors make solving cryptic crosswords a difficult problem:

Ambiguity Cryptic crosswords are deliberately ambiguous. Instruction indicators are indistinguishable from string literals, which are identical to words' semantic meanings. Often, the setter will deliberately chose words to give rise to further ambiguities. For example, the *Telegraph* printed

```
Bug starts to move in dark, glowing endlessly (5)
```

cluing for 'MIDGE'. Usually "endlessly" and similar mean "remove the last letter", but here it is one of five consecutive words to form an acronym from, with the words "starts to" as an indicator.

State Space Even with only a few different clue types, the number of different readings of one clue based on those grows exponentially with the length of the clue. This means that unless heuristics are applied, the evaluation time for a whole grid longer clues may be unfeasibly long.

Lack of Standardization Although all cryptic crossword share some common conventions, there are no fixed rules shared between publications for what can and can't be a clue, indicator etc.. Although most publications have internal guidelines or style-guides, these are not accessible to the solver, and some publications (such as the *Guardian*) have named setters whose styles and self-imposed rulesets differ, even between one publication. Alistair Ferguson

Ricthie, who set for *Listener* for many years, referenced the concept of fairness in his book *Armchair Crosswords* in 1946. He defers the judgement of fairness to a notional rulebook:

> We must expect the composer to play tricks, but we shall insist that he play fair. *The Book of the Crossword* lays this injunction upon him: "You need not mean what you say, but you must say what you mean." This is a superior way of saying that he can't have it both ways. He may attempt to mislead by employing a form of words which can be taken in more than one way, and it is your fault if you take it the wrong way, but it is his fault if you can't logically take it the right way.

Although *The Book of the Crossword* there have been many books written on the subject of what should and should not constitute a valid cryptic crossword clue. One of the most notable and influential was written by *Observer* setter Derrick Somerset Macnutt, both cluing and writing under the name Ximenes, in his book *Ximenes on the Art of the Crossword Puzzle*. The book contains many in-depth guidelines about what a fair clue entails, summed up by his successor Azed (Jonathan Crowther, born 1942):

> A good cryptic clue contains three elements:
>
> 1. a precise definition
> 2. a fair subsidiary indication
> 3. nothing else

A crossword setter following these rules is said to adhere to 'Ximenean principles' and their produced work to be Ximenean. Most mainstream crosswords exist on a continuum between being more closely Ximenean (examples include *The Times*, the *Independent*) to being very libertarian (e.g. *Guardian*). No crossword in a major UK newspaper is 'strictly Ximenean'.

Knowledge Base As well as being made up of encrypted and hidden meanings, cryptic crosswords also draws on a diverse knowledge base of synonyms, abbreviations, facts etc. These can include information as diverse as names of capital cities, common sayings, and the fact that one may carry a wallet in one's pocket.

In order to run a fully working cryptic crossword solver against any arbitrary clue, all of these pieces of information must be encoded, stored and accessible to the solver in a machine readable form. Understandably, this is subject to an entire field of study itself.

1.4 Programming Language Analogues

Much of the current work on interpreting crosswords draws on work by Backus, Naur and Chomsky in creating a specification for the grammar of crosswords. While these frameworks are useful for describing many different languages, interpretations of the grammar of cryptic

crosswords seem to be perversely somewhat closer to mathematical and programming languages than to natural language. In some ways, the cryptic clue as a whole can be thought of as a program that generates the output string as its answer. The wordplay section is analogous to a program, and the definition section of a clue could be thought of as a checksum to verify the final answer.

1.4.1 Lexing, Parsing, Evaluating

The steps for compiling and running a computer program apply also to solving (or 'running') a crossword as a program. Each word in the input string needs to be tokenized, parsed into a relevant structure and then that structure evaluated to produce the final answer. Unusually for a programming language, however, the grammar of a cryptic crossword is highly ambiguous, and requires complex parsing. Firstly, programming languages are only usually required to output the one valid abstract syntax tree, however here we may need to output many thousands in order to evaluate them to see which yields the correct answer. Furthermore, the grammar cannot be expressed without using complex context-sensitive features such as lookbacks, lookaheads and backtracking. Most major programming languages are parsed without these features, allowing information to flow in one direction from the lexer to the parser. To parse a cryptic crossword, lexing and parsing need to take place simultaneously in a process referred to as "Scannerless Parsing".citation TF

1.5 There's No Accounting for Wit

Along with clearly defined and program-like cryptic crossword clues, there exist other clues that rely on humour, imagery and wit, rather than following the regimented classical structure, as set out by Ximenes. Some examples include:

```
Flower of London? (6)
    (= THAMES, flower = that which flows)
In which you can get three couples together and have sex (5)
    (= LATIN, 'sex' is 6 in Latin)
```

Clues such as these, and the question of computerised wit and humour, unfortunately exist out of the scope of this project.

2 Parsing Frameworks and Notation

Some different notations for denoting parsing of cryptic clues have come out of previous work – in order to provide a rigorous analysis of the structures and conventions of cryptic crosswords, it is necessary to analyze and choose a framework in which to do it.

2.1 LACROSS

William and Woodhead produced language called LACROSS, which forms a sort of calculus for describing crossword clues. They also provide a BNF definition of this grammar. Their clues are of the general form

```
Clue := Δ = G | G = Δ
```

the orientation of which corresponds to the order in which we find the definition (Δ) and the wordplay (G) in the clue. The wordplay may be further expanded out – the wordplay section of the clue is expressed as a sequential annotation for the constituent parts, either as 'text' (t), 'shortening' (S) (etc.) or as placeholders for the operators (*), which are detailed afterwards, including a reference to the substituted indicator. So for instance:

```
Get in odd bit of colour (5) [= tinge]
t* = Δ, a (odd, a)
```

There are several issues with this grammar. Firstly, all unitary operators are treated the same, as are all binary operators, and there is some issue with binding and precedence which they address with an underlining notation, in addition to brackets. Secondly, the grammar attempts to include both the structure of the parsed cluing and how that structure relates to the original sequence of words at the same time. As a result, we end up with complex grammar that does not aid human parsing of the solution well, nor does it lend itself easily to computer or mathematical manipulation

Regardless, the paper provides a basis for future work, and begins a basic enumeration of clue types.

2.2 Simple Clue Markup Language

Proposed by Hall and Rapanotti, Simple Clue Markup Language (SCML) attempts to notate the structure of the solution directly onto the clue.

Double underlining is used to denote the definition, underlining denotes an operator, with its class as an optional subscript, with scope provided by brackets and concatenation (and definition/wordplay separation) given by a semi-colon. The following example is taken from their paper:

```
Note the shuddering appliance Bill regularly installed, noisy thing (6,7)
Note;(the)shuddering_a;(appliance,(Bill)regularly_t)installed_e;noisy thing
```

Note' often indicates a musical note, resolving to one of 'a' to 'g', 'do', 're', 'mi', etc;

‘the shuddering’ may be an anagram indicator applied to ‘the’;

the ‘regularly’ of ‘Bill regularly’ may indicate alternate letters (‘t’); i.e., ‘bl’ or ‘il’; and

‘installed’ suggests the embedding (‘e’) of those letters within something meaning ‘ appliance’.

In this, we have no markup differentiation for literal strings (‘Bill’) against words with their semantic context (‘appliance’), and we also take certain words that reduce to abbreviations (‘Note’) to be non-deterministic nullary operators. With some changes and additions (tagging of string vs. semantic word, for example), this markup serves as a good way to represent a parsing of a clue in a human readable way. It even has the advantage that a printed clue could be annotated (carefully) by hand, as a teaching aid, for example. Unfortunately, the language as it stands is not expressible as a BNF grammar, nor is it a particularly good format for representing the clue and its parsings internally in a program (as it would need to be re-parsed to use!)

2.3 Clue-answer notation

There are several emergent solutions within online cryptic crossword communities for notation to explain solutions derived from clues. From http://cryptics.wikia.com:

> Consider the down clue A message from the setter, hauled up with broken arm after heroin withdrawal (8) yielding the answer TELEGRAM. The corresponding wordplay, having the prolix and possibly ambiguous explanation THE next to LEG reversed next to an anagram of ARM, all with H (heroin) removed could be concisely represented in clue-answer notation simply as T[h]E,GEL<=,(ARM)*.

These meanings are not fixed, but some definitions are given here:

ABC<= or ABC (rev.) ABC reversed. The (rev.) notation is most commonly used when the wordplay consists of a single reversal.

[abc] or -abc or (abc) Letters abc removed, as in [c]OUNT to represent ’count’ with c removed; the convention is to use lower case for the removed letters.

(ABC) Letters placed inside others, as inC(AND)ID to mean ’and’ inside ’cid’.

"ABC" Homophone of ABC.

(ABC)* Anagram of ABC.

A+B or A,B A concatenated with B. Sometimes both notations are used together where ambiguities may arise.

aBcDeF Alternate letters of ABCDEF (shorthand for[a]B[c]D[e]F).

2.4 PICCUP

Hart and Davies define what is currently the most satisfying proposal for a formal syntactical definition of cryptic crossword syntax, in a BNF-like grammar. Theirs is the only current definition that closely resembles a usable formally defined language.

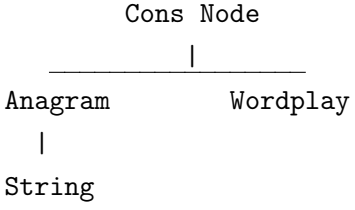
Their interpretation only specifies the grammar in terms of building an abstract syntax tree, rather than attempting to include a notation for clue or answer. The syntax is as given in their paper.

```
Anagram → Synonym(.Equ Indicator).AnagramSentence
/AnagramSentence(.Equ Indicator).Synonym
AnagramSentence → AnagramPointer.AnagramMaterial
/ Anagram Material.Anagram Pointer AnagramPointer~ Word(.Word)*
AnagramMaterial → Word(.Word)*
Synonym → Word(.Word)*
Equ Indicator → Word (.Word)*
```

2.5 Context Free?

The grammar we will describe in chapter 3 is not a Regular Grammar (for example: any of the binary operators generate two non-terminals), but it can be formulated as a context-free grammar.

We define a CONTEXT-FREE GRAMMAR (CFG) as one in which the expansion of a non-terminal is not affected by the symbols before and after it. While we can certainly define a working grammar for cryptic crossword clues in terms of a CFG, it may be useful to consider other options as a means of reducing the number of trees generated during the parsing phase to speed up the evaluation phase. We could take, for example, the clue length as a contextual variable: in that case, a 6-letter clue whose parse tree contains an anagram of a 4-letter string cannot yield another anagram of 5 letters.



```
  |
``apples''
```

In this example, the wordplay on the right should not be expanded out to an anagram node featuring a string of 5 characters (to consume, say, the string "mixed pears"). Make this example clearer and update to new graph structure, if graph is the best way of doing it.

2.6 Syntax vs Semantics

Due to the ambiguous and duplicitous nature of the structure of cryptic crosswords, especially the deliberate challenges in the lexing phase, the boundaries between parsing the syntax and evaluating the semantics become less clear.

Strings consisting of one or more words can be at once tokens representing different operators, they can be strings, and can be split in multiple ways into combinations. This is especially true when we have token that, in the original text, represent their semantic meaning in English, and evaluate out to a finite number of equivalent words (roughly, synonyms: see later for discussion about this equivalence relation). Give examples here

Hall and Rapanotti treated these roughly as their own operators: so the string "rough" would parse to the token `Rough`, which later evaluates to a finite number of definitions. This may be tempting, except for the huge number of words which function as indicators. We do, however, need to differentiate these from raw string literals that are subject to Hidden Word or Anagram operators, or occasionally concatenated in their raw form (for example the string "it" is sometimes taken as given where necessary) .

I think a more manageable way and satisfying way to consider these options is to consider them subject to an invisible 'word' operator. This keeps the semantics and syntax more separate, but certainly poses some challenges for a parser/lexer.

3 The Cryptic Crossword Clue

3.1 Syntactic and Metasyntactic Notation

In the definitions here we write not literate Haskell, but in a convention similar to the one used by Hart, in using a modified Backus Naur Form (BNF). We have seen that a context-free grammar may not be sufficient to model a cryptic crossword, and may have further deficiencies as a basis for finding a solution. Nevertheless, we will adopt a similar notation for clarity:

```
→ = is composed of
, = followed by
| = or
(x) = x is optional
```

```
x* = 1 or more occurrences of x
(x)* = 0 or more occurrences of x
```

We also take the BNF conventions

```
Word = non-terminal symbol
‘‘word’’ = string literal
[x, y, z] = list containing x y and z
(x, y) = pair x and y
```

For clarity, we additionally pre-define the type

```
String
```

to represent any string literal.

3.2 Structure of a cryptic clue

This repeats itself - can we move this further towards the front? TF A cryptic crossword differs from a normal crossword in that the clue for each answer consists of two parts. The first is the definition, which performs the same function as a clue in a 'regular' crossword. The answer to the clue is usually a synonym for the definition ('circular' and 'round') or may be an example of the definition ('farm animal' and 'pig'). Other forms that the definition may take will be discussed later on. The second part of the clue is the wordplay. This is an encoded and often ambiguous second method of deriving the answer, using techniques such as anagram, substitution and concatenation. The clue as a whole is presented as a concatenation of the two parts, sometimes with a subsidiary word indicating that one can be derived from the other (for example, 'from' or 'is'). We can present this breakdown as:

```
Clue → Definition, (Indicator), Wordplay
       | Wordplay, (Indicator), Definition
```

The final clue will often resemble a valid English utterance, although this 'surface reading' very rarely has any relation to the answer. Later on we will consider other information and context within the definition of a clue.

3.3 Definition

The definition of the clue consists of one or more English words. The answer to the clue will be a word or phrase that fits an appropriate equivalence function (that we will define later).

The definition carries a variety of linguistic features with it that the overall answer, and so the answer as derived by the wordplay, must match. These include aspect (noun, verb, adjective), plurality (tree, trees), tense (go, going, gone). These features may also be considered as 'context' to the clue itself. We can define the definition as

```
Definition → Words
```

3.4 Wordplay

The wordplay section of a clue is a set of deliberately ambiguous instructions that allows the solver to arrive at the eventual answer. As the instructions are ambiguous, multiple possible parsings of the instructions are possible. Some of these parsing will not lead to a valid English word:

```
Imbecile, bonkers, in a cult (7)
==> Wordplay ‘Imbecile, bonkers = definition ‘in a cult’
==> Anagram ‘imbecile’ [indicator = bonkers] = definition ‘in a cult’
==> ??? (no anagrams of imbecile in english language)
(correct reading was anagram of in a cult = lunatic)
```

Others will lead to a valid English word, but one that is not equivalent to the definition:

```
Minder shredded corset (6)
==> Wordplay ‘minder shredded’ = definition ‘corset’
==> Anagram ‘minder’ [indicator = shredded] = definition ‘corset’
==> ‘remind’ = definition ‘corset?’ X
(correct reading was anagram ‘corset’ = escort = minder)
```

The solver must find the correct parsing of the wordplay that yields the correct definition: even though they may not know which part is wordplay and which is definition.

3.5 Special Operators

I include these two operators first, as they really form the backbone or basis of other clues. They are also unique in being implicitly clued, rather than requiring an indicator word to signify their presence. This is ugly, and we should give them first, and then talk about others

Word Equivalence In the most simple of clues, we have the definition, along with a word or phrase that is somehow semantically equivalent to that definition. [1] A clue that contains

[1] In this case, it becomes a difficult task to be precise about exactly which of these is the definition and which is the wordplay! Sometimes there is a defined answer: From 'Oinking tendency? (8)' we get both 'pen chant' and 'penchant', and we can see from the letters required (no space) that the second half is the solution. In other cases, this may not be defined at all!

just this structure is said to be 'double definition'

```
Metal guide (4) [= LEAD]
```

However, even in this simple example we see that this equivalence relationship is not at all straightforward. While 'guide' and 'lead' are synonyms (as verbs in the present tense), it's not true that 'lead' is a synonym for 'metal'. We must also include 'for example' in this relationship too, which causes us to have to discard reflexivity. Although 'metal' can be a clue for 'lead', it's not the case that 'lead' can be a clue for 'metal' (in that case, we signify 'an example of' by writing 'lead, say' or 'bronze, for instance').

We also include abbreviations, which are perhaps more closely related to synonyms, although not usually found in thesauruses, along with some useful 'setters favourites', where an abbreviation of a synonym or of an example is particularly useful for cluing a difficult letter combination used in a wordplay ('Books' becomes 'NT', for 'New Testament').

```
Words → Synonym | Abbreviation | Example
Synonym → String
Abbreviation → String
Example → String
```

The semantic task of evaluating this will be discussed later.

Concatenation While not strictly necessary for this grammar (as we have included a concatenation in our metasemantics, we could define multiple definitions of each operator in the form Operator → Indicator, Wordplay, (Wordplay)*), it makes sense to add this explicitly as it mirrors the structure of an explanation of a computer solution (i.e. the parse tree). This is a mess

```
Concatenation → Wordplay (ConcatIndicator) Wordplay
```

This represents a key tool for cluers to create more complex wordplay clues in the form of a charade, where two or more parts can be split out (sometimes syllabically as in 'bath', 'tub', or sometimes otherwise 'bat','htub') and clued separately, and then later joined to form the overall solution.

3.6 Other Wordplay Operators

For the other wordplay operators, we define them in terms of our grammar, as well as discussing their semantic meaning.

```
Wordplay → Words | Concatenation | Anagram | Reversion | Contraction
         | Selection | Hidden Word | Containment | Subtraction
         | Homophone
```

These operators all include an indicator word to show they are being applied. Each operator will usually have many different indicators (lists of anagram indicators on the web span multiple hundreds). Only select ones are included in the specification here.

3.6.1 Unitary Operators

Anagram A very commonly used operator in crossword clues is an anagram. These take the form of an indicator word that denotes that the anagram function is being used (called an 'anagrind' within cruciverbalist circles), along with the candidate letters to be anagrammed. The simplest form of this gets the candidate letters verbatim from the clue:

```
Anagram → Anagrind, String | String, Anagrind
```

Sometimes, however sometimes there is some sort of operation applied to the letters before the anagram is applied. For example:

```
Comic bare for short comedy play (7,5)
==> Wordplay 'Comic bare for short comedy' = Definition 'play'
==> Anagram 'bare for short comedy' [anagrind = 'comic']
==> Anagram (''bare for'' + Shorten 'comedy')
==> Anagram (''bare fore'' + ''comed'')
==> Anagram (''bare fore'' + ''comed'')
==> Anagram (''bareforecomed'')
==> ''Bedroom Farce''
```

In which case we find the more general case one proposed structure: What is this sentence

```
Anagram → Anagrind, Wordplay | Wordplay, Anagrind
```

Wherein we know that the repeated evaluation of the Wordplay will eventually result in a string literal that can be anagrammed. In *Art of the Crossword Puzzle,* Ximenes argued against this form of indirect anagram:

> Secondly – and here, for once, I differ from Afrit – I hate what I call an indirect anagram. By that I mean "Tough form of monster" for HARDY (anagram of HYDRA). There may not be many monsters in five letters; but all the same I think the clue-writer is being mean and withholding information which the solver

> can reasonably demand. Why should he have to solve something before he can begin to use part of a clue? He has first to find "hydra" – and why shouldn't it be "giant"? – and then use the anagrammatic information to help him think of "hardy". ... My real point is that the secondary part of the clue – other than the definition – is meant to help the solver. The indirect anagram, unless there are virtually no alternatives, hardly ever does. He only sees it after he has got his answer by other means.

Even so, most setters that claim to be Ximenean will allow small abbreviations and contractions (to be defined later) to be included in their clues. We therefore must define a new class which includes String Literals as well as the abbreviation where appropriate. Get rid of this bit, say that we're going to stick with him and not do that.

```
Anagram → Anagrind, StringWordplay* | StringWordplay*, Anagrind
Anagrind → ‘‘free’’ | ‘‘novel’’ | ‘‘comic’’ [...]
StringWordPlay → String | Abbreviation | Contraction
```

Reversal Clues can also be reversed. While this is functionally a subset of anagrams, there are some crucial differences. Firstly the 'directionality' of the clue (i.e. whether it is a 'down' or an 'across') comes into effect, in determining the sorts of indicators that can form it: "turned back" may only apply to 'across' clues, where "taken up" may only apply to 'down' clues. Further, these clues are usually taken to be 'fairer' game for subsequent operations to be applied to the target of the reversion. Therefore, a clue with nested wordplay such as (`Stressed, made upside-down puddings (7)` = DESSERTS) would be acceptable, where an equivalent clue as an anagram (`Stressed, cooked up puddings`) would often not be seen as Ximenean.

```
Anagram → ReversalIndictator, Wordplay | Wordplay, ReversalIndictator
ReversalIndicator → ‘‘around’’ | ‘‘turned back’’ | ‘‘taken up’’ [...]
```

Contraction Clues of this form range from specific, such of first/last letters ('first in line' = 'l', 'last of the Mohicans' = 'm') to more general operators ('mostly harmless' can yield 'armless', 'harmles', 'harmle'...) whose definitions are more flexible.

```
Contraction → FirstLetterContraction | LastLetterContraction | GeneralContraction
FirstLetterContraction → PreFLCIndicator, Wordplay | Wordplay, PostFLCIndicator
LastLetterContraction → PreLLCIndicator, Wordplay | Wordplay, PostLLCIndicator
GeneralContraction → PreGCIndicator, Wordplay | Wordplay, PostGCIndicator
```

Selection There are three similar operators here: A pair which selects even or odd letters respectively, and one which takes initial letters across multiple words. These are rarely, if ever, applied to anything other than pure strings. The initials indicator needs to be applied to an argument consisting of multiple words.

```
Selection → Evens | Odds | Initials
Evens → EvensIndicator, String | String, EvensIndicator
Odds → OddsIndicator, String | String, OddsIndicator
Initials → InitialsIndicator, String, ‘‘ ‘‘, String* |
           String, ‘‘ ‘‘, String*, InitialsIndicator
```

Hidden word The hidden word clue finds a word which appears as a substring (ignoring spaces) inside its operand. These typically only occur once per puzzle, and are always accompanied by a clear indicator. In this example clue:

```
’Smack which appears in East Anglian ports.(4)’ Change this example.
As it’s shit, and stolen
```

the solution to this example is ’TANG’, (meaning ’smack’ in the sense of ’taste’), and which is concealed (indicated by ’which appears’) in ’easT ANGlian ports’.

```
HiddenWord → HWIndictator String | String HWIndicator
```

Homophone

Also called ’sounds like’, this operator produces homophones of a given word, for example ’right’ and ’rite’. This operator is not applied to words that are both spelled and said the same, but with different meanings (’must’ as an imperative and ’must’ as a noun).

Often, if clues are straightforward, placement of this operator can determine the spelling of the answer.

```
We hear twins shave (4)
```

yields ’pare’ whereas

```
Twins shave, we hear (4)
```

yields ’pair’. A formulation with the indicator in the middle, in this case, would result in a strong ambiguity. The homophone indicator is only applied to equivalence words, not to clued wordplay.

```
Homophone → HomophoneIndicator Words | Words HomophoneIndicator
```

3.6.2 Binary Operators

As with the unitary operator, each of the arguments of binary operators can be one or more words.

Containment Here are two styles of wordplay which are clued very differently, but are actually the same operator, which places one set of letter inside another. This is either presented as a insertion (e.g.. 'end inside ls') or as a containment (e.g. 'ls around end'). This operation always preserves letter order, unless some nested indicator allows otherwise.

```
Containment → Wordplay ContainmentIndicator Wordplay
ContainmentIndicator → ‘‘inside’’ | ‘‘around’’ [...]
```

Subtraction In a subtraction clue, a number of letters are removed from the target. Usually, the target is some wordplay itself, although sometimes just a string literal. The letters to be subtracted are also often the product of some sort of cluing, although this is usually fairly limited in scope (abbreviations, contractions, first letters of string literals). There are two constraints on this: all the letters from the subtraction set must be in the target, and the length of the subtraction set must be less than the length of the target. Give example

```
Subtraction → SubPreIndictator1 Wordplay (SubPreIndictator2) Wordplay
   | Wordplay SubMidIndictator Wordplay
   | Wordplay (SubPostIndictator1) Wordplay SubPostIndictator2
SubPreIndictator1 → ‘‘took’’, ‘‘without’’ [...]
SubPreIndictator2 → ‘‘from’’ [...]
SubMidIndictator → ‘‘without’’ [...]
SubPostIndictator1 → ‘‘with’’ [...]
SubPostIndictator2 → ‘‘removed’’, ‘‘deleted’’ [...]
```

Semantically here, we have the difference in pre- and post- as the difference between “wanted ant removed” and “removing ant wanted”

The letters in the set are thought to be removed in the order in which they're found in order to be a properly clued wordplay. Thus “standing” with “tan” removed, gives “sanding”, whereas “ant” cannot be appropriately removed. Note though that the order in which nested clues are applied can change what the set is applied too. If we also had an anagram indicator, as in “Boy muddled standing missing trap” we can apply the muddled to standing to get “dansting” before removing “sting” to get the answer “Dan”.What the fuck is this example, change it.

3.7 Meta-references

Sometimes, clues contain references that cannot be parsed in isolation, or contain a cluing structure that is incompatible with the main model of cluing. Due to their complexity and requirement for context, I consider clues such as these outside of the scope of this project. These include:

3.7.1 Self reference

A type of clue called an '&lit' clue allows the setter to not include a definition part if the text that makes up the wordplay also can also be read as the definition. Thus in

```
Spoil vote! (4)
```

we have the wordplay Anagram (=spoil) "vote" to give 'VETO', as well as the clue as a whole 'spoil vote' meaning 'veto'.

3.7.2 Reference to other clues

Some publications will have clues that reference the answer to other clues ('8 across. Cake made badly by 7 down.'). Sometimes these may also be cyclical (in this example, 7 down would reference 8 across too).

3.7.3 Contextual References

Sometimes references will refer outside of the crossword itself. For example, The *Sunday Telegraph* on Easter Sunday 2014 had an anagram clue whose answer was EASTER SUNDAY, and its definition part was "today". In a crossword by setter *Araucaria*, "Araucaria is" coded for IAM (= "I am") as part of an answer.

Part III
Naive Approach

4 Solving Through Functional Programming

The approach I will take to parsing and solving cryptic crossword clues will be by using the functional programming language Haskell to generate and evaluate abstract syntax trees.

Haskell lends itself well to parsing languages: there are Haskell parsers for javascript[2], scheme[3], and even natural language[4]. The Glasgow Haskell Compiler itself is written largely in Haskell[5].

There may be many reasons for this. Firstly, Haskell's data structures lend themselves well to modelling abstract syntax trees. Secondly, lazy evaluation means that large quantities of trees may be produced symbolically and only analysed when necessary, meaning that large and complex grammars which produce many parses can be handled elegantly. Finally, Haskell's type strictness makes it possible to write complex programs that act upon complex external data structures without requiring large quantities of unit or integration testing.

While I benefit heavily from many of the features of Haskell, the work here could be implemented without too much adjustment in many other functional languages, and many modern multi-paradigm languages, such as Python.

5 Parsing and Evaluating everything

5.1 Solving a Clue

Our motivation here is to take a cryptic crossword clue, for example:

```
[A]  Ship carrying right flag (8)
[B]  Companion shredded corset (6)
```

and attempt to parse and solve it to provide the correct answer. We will define the datatype of Clue thus:

```
data Clue = Clue String AnswerLength
where
type Length = Int
```

[2]https://github.com/alanz/language-javascript
[3]https://github.com/zenazn/scheme-in-haskell/
[4]http://homepages.inf.ed.ac.uk/wadler/realworld/natlangproc.html
[5]http://www.haskell.org/haskellwiki/Implementations#Glasgow_Haskell_Compiler_.28GHC.29

In order to solve this clue, we want to find a function that takes a clue, which consists of a string containing the text of the clue and an integer representing the length of the required answer, and returns us the answer.

```
solve :: Clue → Answer
```

The intuition behind how our naive solver will work is that it will generated all possible ways of parsing a clue, then generate all possible answers that could be derived from those parses, and then attempt to match those up with the definition and the length constraints. In order to evaluate, measure and optimize each of these steps independently, we split the structure of our program into four parts:

```
solve = choose . evaluate . parse . split
```

where the types are given below:

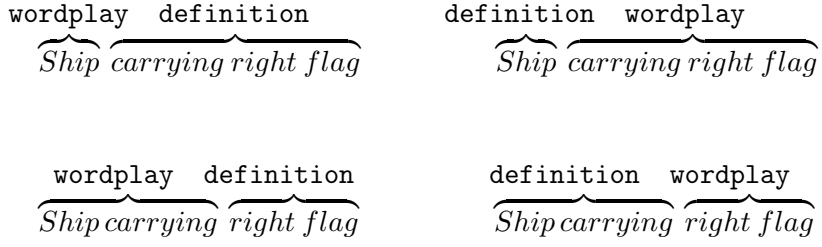
```
split    :: Clue → [Split]
parse    :: [Split] → [Parse]
evaluate :: [Parse] → [Answer]
choose   :: [Answer] → Answer
```

5.2 Splitting

While a clue has a surface reading involving the semantic natural language parsing of it as a sentence fragment (which would yield a phrase, with an subject, a past tense verb and an object), we are only interested in the crossword interpretation of this, which is of the form:

```
  Definition Indicator* Wordplay
| Wordplay Indicator* Definition
```

Let us forget about the optional indicators for now – we will deal with these properly later . We are looking to define a function `split` which splits the clue into a wordplay portion and a definition portion. So for example, clue `A` can be split 6 different ways:



$$\overbrace{\texttt{wordplay definition}}^{\textit{Ship carrying right}}\overbrace{}^{\textit{flag}} \qquad \overbrace{\texttt{definition wordplay}}^{\textit{Ship carrying right}}\overbrace{}^{\textit{flag}}$$

From the types of `Wordplay` and `Definition`:

```
type Definition = String
type Wordplay = String
```

we can create a datatype

```
data Split = Def Definition Wordplay AnswerLength
```

as well as the signature of a function `split`:

```
split :: Clue → [Split]
split (text length) =
    let parts = partitions . words $ text
    in [Def (unwords d) (unwords w) length | [d,w] <- parts]
```

where partitions finds all ways of partitioning a list, and is defined as

```
partitions [] = [[]]
partitions (x:xs) = [[x]:p | p <- partitions xs]
                ++ [(x:ys):yss | (ys:yss) <- partitions xs]
```

5.3 Parsing

Now we have consumed one portion of the string to form the definition in each of a list of splits. Now we need to parse the rest of the clue into a structure which we can evaluate to produce our answer. Let us take for an example the correct split (of the 6 available) of clue A:

$$\overbrace{\texttt{wordplay definition}}^{\textit{Ship carrying right}}\overbrace{}^{\textit{flag}}$$

which would have the Haskell structure of

```
Def ‘‘flag’’ ‘‘ship carrying right’’ 8
```

The `parse` function must, for each split, consume the wordplay and return all possible parses for that wordplay. Since each split will return multiple parses, we will want to collect these afterwards. We define datatype `Parse`:

```
data Parse = Parse Definition ParseTree AnswerLength
```

where `ParseTree` will be an Abstract Syntax Tree based on the structure of our clue.

```
data ParseTree =  ConcatNode ParseTree ParseTree | SynonymNode String
   | AnagramNode Anagrind String
   | InsertionNode InsertionIndicator ParseTree ParseTree
   | SubtractionNode SubtractionIndicator ParseTree ParseTree
   | HiddenWordNode HWIndicator [String]
   | ReversalNode ReversalIndicator ParseTree
   | FirstLetterNode FLIndicator [String]
   | LastLetterNode LLIndicator [String]
   | PartialNode PartialIndicator ParseTree
```

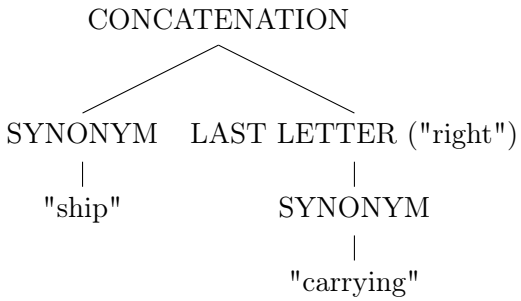
So we will define:

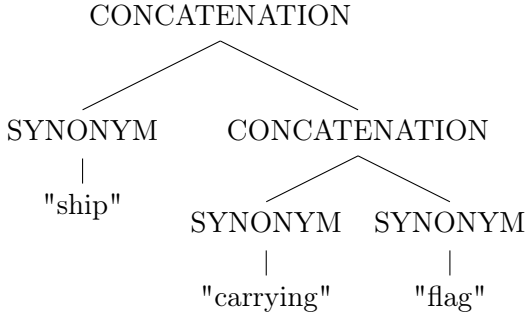
```
parse :: [Split] → [Parse]
parse = concatMap parseClue
```

where

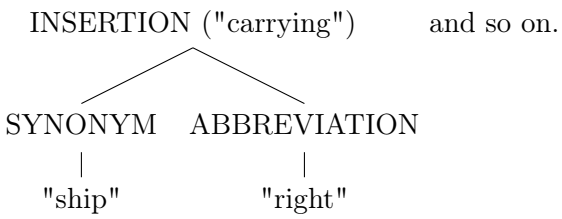
```
parseClue :: Split → [Parse]
```

In our example, we would require `parseClue` to consume the string `Ship carrying right flag` to generate the following parse trees





as well as the correct one:



5.3.1 Traditional Scanner-Based Parsing

The first step in the lexical analysis phase of a parse usually consists of tokenisation. This is the process of grouping characters together into functional groups called TOKENS, to later pass to the parser to perform the semantic analysis on. Tokens consist of a LEXEME the string of characters known to be of a certain type, and the value they represent (for example INTEGER 3 or VARIABLE NAME available_credit). The process is often split into two stages.

The Scanner The first is the SCANNER: this is often a finite state machine, which will consumer characters based on rules to produce potential lexemes. Some more simple scanners can operate under greedy assumptions (called the Maximal Munch principle byReference this R.G.G. Cattel), and some require backtracking (for example, the language C). Due to the complexity and ambiguity of the language of cryptic crossword clues, it is not possible to produce an accurate scanner that produces anything other than a trivial tagging of lexical elements[6]

The Evaluator This stage of the tokeniser... Yeha, finish this. This point is probably less interesting Maybe the above sections should be merged together?

Scannerless Parsing Some parsers, traditionally often ones for simple languages such as...

Todo: need to finish off this section and explain why my parser is more like a scannerless parser. Also – this section is a bit incongruous here and kinda breaks the flow. Maybe it should be pushed elsewhere?

5.3.2 Parsing different clue types

So we need to define a function `parseClue` which will produce a parse tree from an unconsumed split. So we have

```
parseClue :: Split → ParseTree
```

[6]It would, of course, be possible to produce a trivial parser for most languages, in which we lex every character or group of letters to a function with the value of itself, so instead of the desired VARIABLE x EQUALS INTEGER 3 we could instead simply parse to FUNCTION x FUNCTION = FUNCTION 3, and leave it to the rest of the pipeline to determine that FUNCTION 3 is a constant function which always yields the integer literal 3, but this misses the point of having a scanner separately.What the fuck was this again?

We will define parseString in terms of its parsing of various clue types, starting with one of the more simple unary ones. We will then write a parser combination function [Frost, Launchbury] to combine the different sub-parsers into one top-level parser. We will hold back from the details of unpacking a split into the string to be consumed until later.

```
parseAnagram :: String → [ParseTree]
parseAnagram xs =
 [AnagramNode (AIndicator x) y |
              (x,y) <- includeReversals . twoPartitions $ xs
                                    , isAnagramIndicator(x)]
```

where

```
twoPartitions xs = [(x,y) | [x,y] <- partitions xs]
includeReversals xs = xs ++ [(snd(x),fst(x)) | x <- xs]
```

We allow both (x, y) and (y, x) through `includeReversals` in order to allow `muddled word` and `word muddled` both to indicate anagrams of "word". This means a that in example [B] we parse both

.

DEFINITION "companion"
|
ANAGRAM ("shredded")
|
"corset"

and

DEFINITION "corset"
|
ANAGRAM ("shredded")
|
"companion"

.

Anagram clues, along with Hidden Word clues only require a definition and a string, so their operands don't require any further parsing. Other clues, though, may require the operands to be parsed. For example, the parsing of the clue `SWEETHEART NEARLY FINISHED (5)` as `(L)OVER` requires `SWEETHEART` to be parsed into a synonym node after we consumer nearly to be an indicator for a partial word node.

DEFINITION "finished"
|
PARTIAL ("nearly")
|
{unparsed: "sweetheart"}

⇒ parse

DEFINITION "finished"
|
PARTIAL ("nearly")
|
SYNONYM
|
"sweetheart"

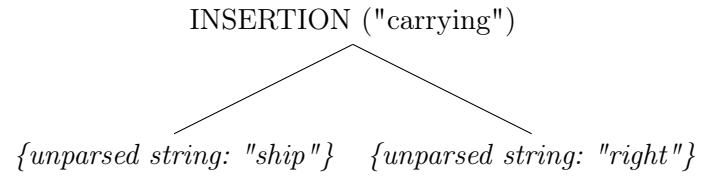
give

a better example - over and lover confusing here

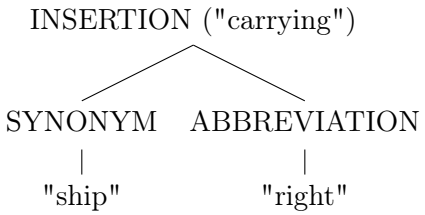
we therefore perform what is called Recursive Descent Parsing [Lewis]citation, letting Haskell's list comprehension take care of matching the correct partition to the correct parse.

```
parsePartialNode :: String → [ParseTree]
parsePartialNode xs = [PartialNode (LLIndicator x) y'
         |(x,y) <- includeReversals . twoPartitions $ xs
          , isPartialIndicator(x)
          , y' <- parseClue y]
```

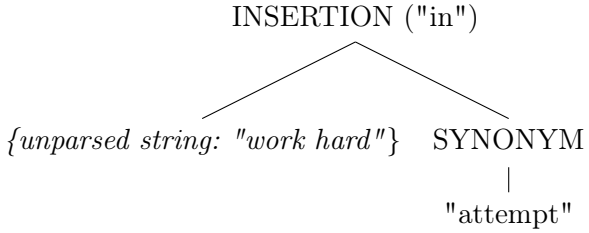
Still more complex clue types require splitting into three parts – two branches and an indicator – and often both of these branches require further parsing. For example, in the case of `SHIP CARRYING RIGHT FLAG`, choosing `FLAG` as the definition, we can generate



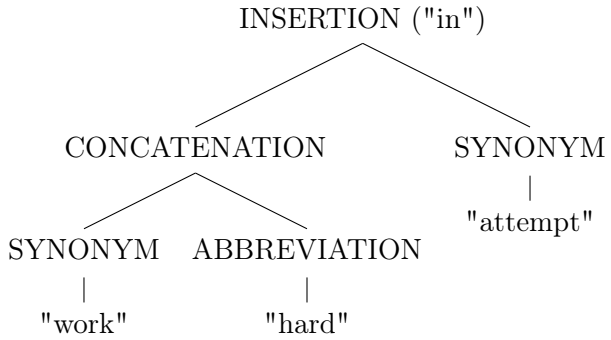
and then consume each of the unparsed strings in turn to produce



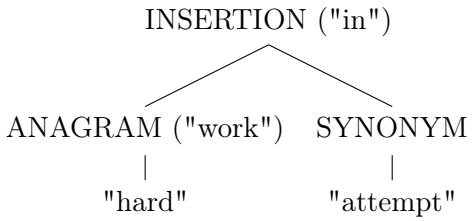
It is worth noting here that as well as the top-level parse generating multiple different options, each of these sub-parses may also generate several different parses, and these themselves may be complex with multiple sub-parses. In the clue `WORK HARD IN ATTEMPT TO GET CUP`, with definition (='`“”to get`') of `CUP`, we can parse the wordplay as



which may subsequently evaluate to the (correct, in this case) parse:



as well as others, such as:



Here, again, we allow Haskell's list comprehension take care of constructing the sub- parse-trees from our recursive calls and constructing them into our final list of trees, for example in

```
parseConcatNodes :: String -> [ParseTree]
parseConcatNodes xs n = let parts = twoParts xs
      in [ConcatNode x' y' |( x,y,z) <- parts
                           , x' <- (parseClue x)
                           , y' <- (parseClue y)]
```

and in

```
parseInsertionNodes :: String -> [ParseTree]
parseInsertionNodes xs n = let parts = threeParts xs
      in [InsertionNode (IIndicator y) x' z'
          | (x,y,z) <- parts, isInsertionWord(y)
            , x' <- (parseClue x)
            , z' <- (parseClue z)]
```

We can then compose each expression type together to form our final definition of `parseClue`, checking the number of words in the phrase to check that we will be able to split the string correctly into 2 or 3 parts.

```
parseClue :: Split -> [ParseTree]
parseClue (Def def ys n) = let len = length . words $ ys in
[SynonymNode ys]
++ (if len > 1 then parseConcatNodes ys else [] )
++ (if len > 1 then parseAnagramNodes ys else [] )
++ (if len > 1 then parseHiddenWordNodes ys else [])
++ (if len > 2 then parseInsertionNodes ys else [])
++ (if len > 2 then parseSubtractionNodes ys else [])
++ (if len > 1 then parseReversalNodes ys else [])
++ (if len > 1 then parseFirstLetterNodes ys else [])
```

```
    ++ (if len > 1 then parseLastLetterNodes ys else [])
    ++ (if len > 1 then parsePartialNodes ys else [])
```

We can then define `parse`as

```
parse = concatMap parseClue
```

5.3.3 Number of parses

These nested parses can cause the number of produced parses to grow exponentially as the depth of the nesting increases. Longer strings with more indicators – especially indicators for binary expressions such as insertion indicators and subtraction indicators – are more likely to produce deeply nested parses, and therefore return a large number of parse trees.

Either give some numbers here, talk about the numbers from later on, or just push this whole thing later altogether

5.4 Evaluation

In the evaluation stage we look to define a function `evaluate`with the type signature:

```
evaluate :: [Parse] → [Answer]
```

As the evaluation of each `Parse` will yield a list of multiple `Answer` (e.g. an anagram node of a five-letter word will evaluate to 120 different answers, although very few of them will be valid words), we can define `evaluate`as

```
evaluate = concatMap eval
```

where `evaluate` will consume `Parse` data in the form `Def Definition ParseTree AnswerLength`and produce `[Answer]`, where:

```
data Answer = Answer String Parse
```

The parse is included along with the answer, as it contains the definition for that parse, which will later allow us to check that our generated answer has some relation to what we thought we were looking for in that parse, and also allows us to reconstruct the reasoning behind the clue by inspecting the parse tree.

We then define `eval` as:

```
eval (Def d pt l) = [Answer x (Def d pt l) |
                                        x <- evalTree pt]Do this
whole section bottom-up pls
```

We can then define `evalTree`Refeerence in terms of the different types of node in our `ParseTree` type. Those without subtrees will be defined simply with reference to a Haskell function that performs their action:

```
eval_tree (AnagramNode ind xs) c = anagrams xs
eval_tree (SynonymNode xs) = synonyms xs

anagrams :: String -> [String]
anagrams [] = [[]]
anagrams xs = [x:ys | x<- nub xs, ys <- anagrams $ delete x xs]

synonyms :: String -> [String]
synonyms xs = Map.lookup xs thesaurus
```

and so on. Clues with sub-trees are treated with a similar recursive call, with either a map, or a list comprehension applying the expressions function to each generated sub-answer

```
eval_tree (ReversalNode ind ys) = map reverse (eval_tree ys)
eval_tree (ConcatNode ind xs ys) = [x ++ y | x <- eval_tree xs
                                           , y <- eval_tree ys]
```

Todo: there's probably space here to give each definition, since this really is the bread and butter of the whole affair

5.5 Selection

Finally, given that we've produced our list of answers, most of which will be meaningless combinations of jumbled letters and synonyms pressed together, we need to filter down to the answer containing a string which in some way meets the criteria set for us in the clue, that is 1. finding an answer that is a synonym of the part of the clue we chose as the definition 2. being the right number of letters.

So we can define

```
choose :: [Answer] → Answer
choose = head . filter valid

valid (Answer ans (Def def pt len)) = (length ans == len)
                                    && (is_synonym ans def)
```

Of course, we may not have generated a valid solution, so we can redefine to include this uncertainty:

```
choose :: [Answer] → Maybe Answer
choose = headM . filter valid
```

6 State space and performance analysis

6.1 Overview - does it work?

This approach has the required structure to correctly parse and solve most cryptic crossword clues — with some caveats.

Firstly, although in most cases the correct parse was generated, often the number of other parses to be evaluated before reaching the correct one was so great that the computation would effectively not end. In this case, the heap size wasn't continually growing, as each evaluation branched and then diminished in turn, but the running time was sufficiently large (>48hrs) such that the computation would be useless in a practical situation. The data for this is considered in **6.2**

In other cases the correct parse was created, however the semantic data wasn't available to evaluate the clue correctly. In other, very rare cases, there is a clue which does not fit the structure of the grammar defined in **Part I**. These do not generate the correct parse trees, and so are not soluble. These are discussed in **6.3**.

6.2 Correctly parsed and evaluated clues

Most clues, if they yield any results at all, yield them within 30 seconds of being run. Many others yield them a very long time afterwards – multiple hours of runtime is required to reach them. Others seem to run indefinitely.

Of those that do not terminate within an acceptable timeframe, the generated parse trees can be inspected and it can be shown that the correct one has been generated, and that since no individual evaluation takes infinite time, and each evaluation uses a non-problematic amount of stack space (that is to say – the stack does not increase with each subsequent evaluation), then we can say that the clue is solvable, even if not in a reasonable amount of time.

The clue `FRIEND FOUND IN OKLAHOMA TERMINAL (4)` yields the correct parse:

```
Def "friend" (HiddenWordNode (HWIndicator ["found","in"])
                                    ["oklahoma","terminal"])
```

however it also generates 59 others, including:

```
Def "friend" (InsertionNode (IIndicator ["in"]) (SynonymNode "found")
               (ConsNode (SynonymNode "oklahoma") (SynonymNode "terminal")))
Def "terminal" (InsertionNode (IIndicator ["in"])
            (SynonymNode "friend found") (SynonymNode "oklahoma"))
Def "oklahoma terminal" (ConsNode (SynonymNode "friend")
                                  (SynonymNode "found"))
```

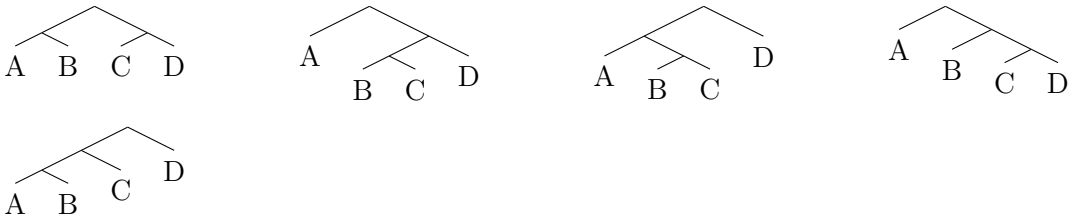
```
Def "terminal" (ConsNode (SynonymNode "friend") (ConsNode (SynonymNode "found")
               (ConsNode (SynonymNode "in") (SynonymNode "oklahoma"))))
Def "terminal" (ConsNode (ConsNode (SynonymNode "friend")
                         (SynonymNode "found")) (SynonymNode "in oklahoma"))
Def "friend found" (SynonymNode "in oklahoma terminal")
Def "in oklahoma terminal" (ConsNode (SynonymNode "friend")
                                     (SynonymNode "found"))
Def "in oklahoma terminal" (SynonymNode "friend found")
```

While evaluation of the correct parse takes 0.05 seconds, the evaluation of the first of the other examples takes over 10 seconds - it is the cumulative effect of the evaluation of the others, as well as the order in which they appear in the list which determines how long the total solving time takes. Some of these effects can been seen in **Figure 1.**

Need a big summary of this table, and to explain that it's just a sample, and to give some top-level figures, or reasons why not

6.2.1 The effect of clue length on the number of parses

The length of the clue has an exponential effect on the number of parses produced. This is due partly to the increasing number of ways in which binary trees can be constructed from N elements, as in:



It also increases the availability for function words to interact with each other - when any A, B, C, or D in the examples above also have multiple parses, this is when we see the strongly trended exponential growth seen in **Figure 2** (displayed on a logarithmic scale).

Clue	Solution	Clue Length	# Parses	# Solutions	Solve Time
COMPANION SHREDDED CORSET (6)	ESCORT	3	8	148,500	0.2s
HOPE FOR HIGH PRAISE (6)	ASPIRE	4	25	105,718,343	1.39s
MARIA NOT A FICKLE LOVER (9)	INAMORATA	5	60	84,855,252[2]	—[1]
FRIEND FOUND IN OKLAHOMA TERMINAL (4)	MATE	5	59	92,995,844[2]	—[1]
PAUSE AT THESE I FANCY (8)	HESITATE	5	54	5,358,615	4.59s
ANKLE WAS TWISTED IN BALLET (8)	SWAN LAKE	5	84	203,991,525	12.13s
NOTICE SUPERVISOR IS GOING NUTS AT FIRST (4)	SIGN	7	853	—[3]	—[1]
ATTEMPT TO SECURE ONE POUND FOR A HAT (6)	TRILBY	8	2930	—[3]	—[1]

Figure 1: Solving statistics for selected clues on a 2014 MacBook Pro

[1] Although the correct parse was generated, and selective evaluation of that parse yielded the correct results (i.e. a solution would be available eventually), the normal solving procedure did not compute the correct answer within 48hrs of running time

[2] Due to Haskell's lazy evaluation, this can sometimes be calculated without actually computing the solution

[3] Could not yield answer within 48hrs

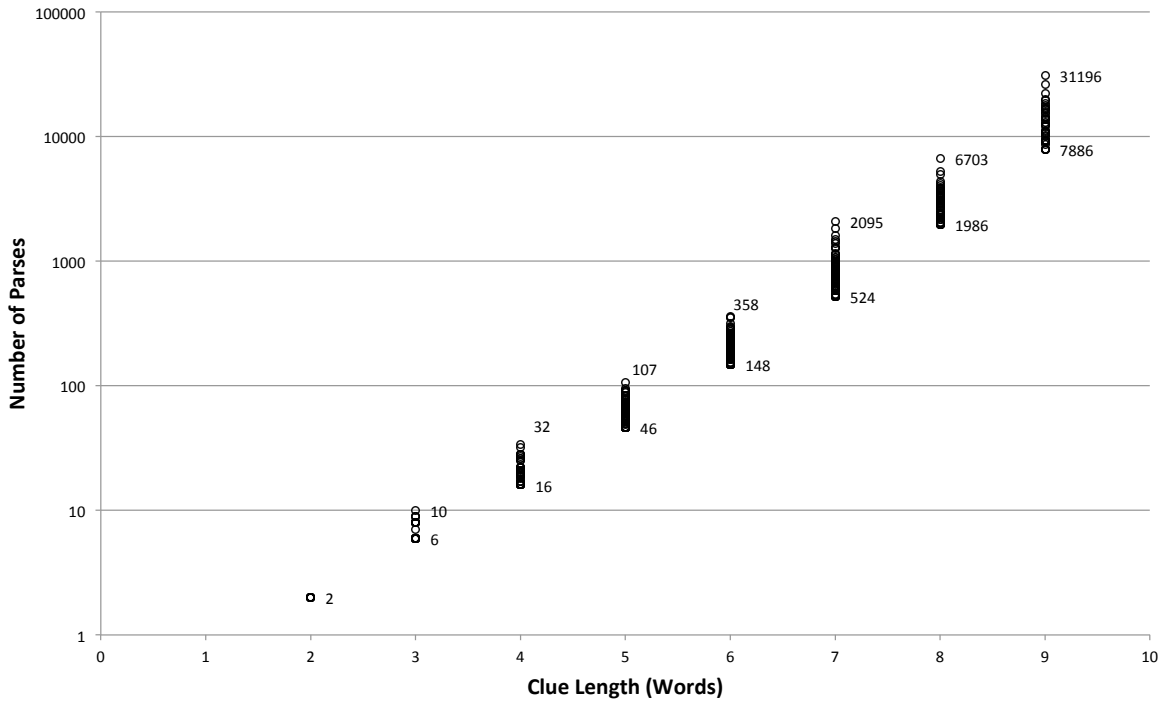
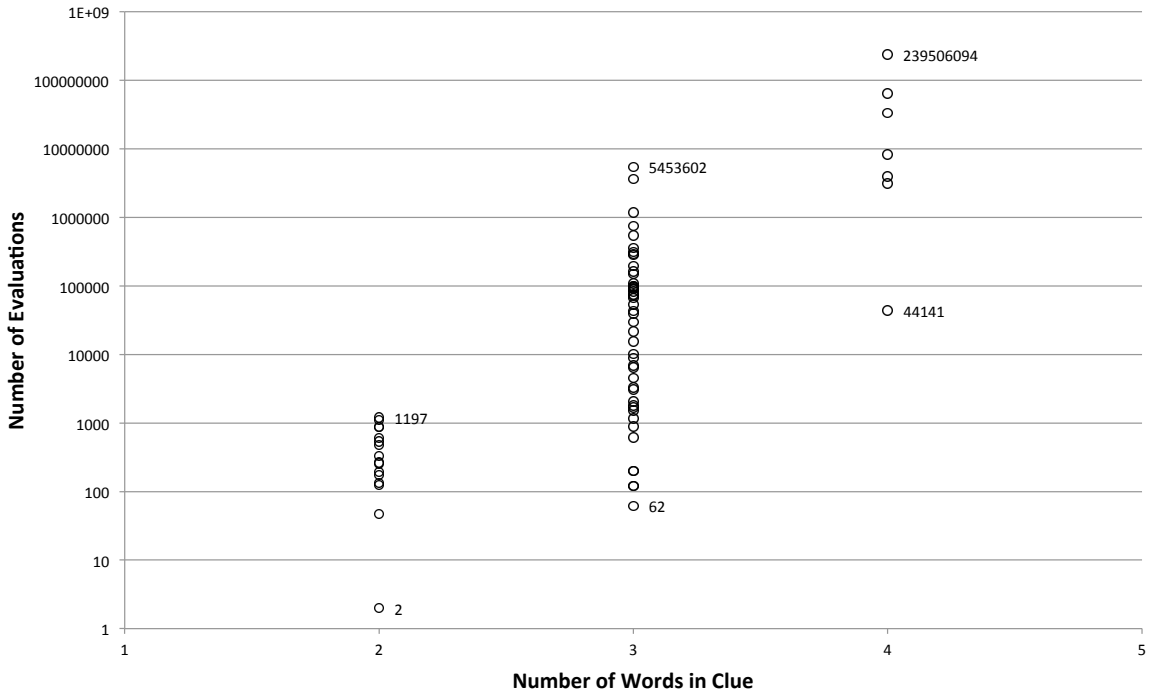


Figure 2: Number of parses generated for varying clue lengths over 600 sample clues

6.2.2 The effect of clue length on the number of solutions

We see a similar but even greater effect on the number of solutions produced, with the effect of the exponential growth per parse compounded by the fact that each parse can evaluate out to to thousands of options. This is due to two effects, Firstly, clues types like anagrams can have many thousand solutions per parse (there are 120 anagrams of a 5-letter word, rising to 40,320 anagrams of an eight letter word – an $n!$ relationship). Secondly, compound clues like insertions, which can take the result of one wordplay and insert into the second, can magnify the effect of branching in its sub-clues.

There are 4 ways that a word 'A' can be inserted into a 5-letter word 'B'. There are 480 ways that it can be inserted into each of the 120 anagrams of word 'B', and if there are also 120 different anagrams of word 'A', then there are 57600 different solutions for that parsed arrangement. reference figure 3. Mention that it's sparse because it becomes v hard to solve the things



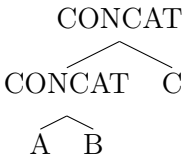


Figure 3: Number of solutions evaluated for varying clue lengths over 75 sample clues

The size of the thesaurus has a large impact on the number of solutions produced, as all clue types (other than Anagram, Hidden Word and Initials, which use String) use Synonym as the lowest level node in their sub-trees.

Figure 4 (also displayed on a logarithmic scale) shows how limiting the number of synonyms returned by the thesaurus affects the number of solutions. The graph plateaus as the restriction exceeds the actual number of synonyms per entry for each word in the thesaurus.

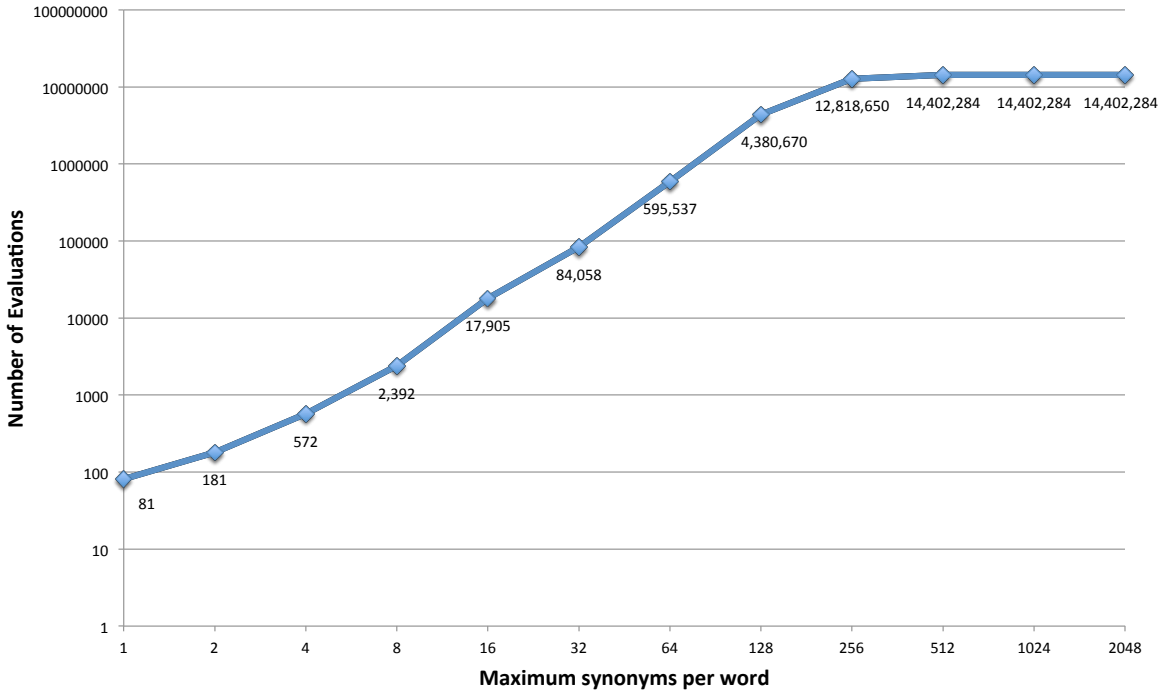


Figure 4: Number of solutions evaluated by restricting the maximum thesaurus length for the clue "Good opportunity in school" (5)

6.3 Analysis of selected clues which are not correctly solved

It is difficult to perform a large-scale analysis of the numbers of clues for which the data does not exist, or where the correct parse is not generated, as often these will present themselves in the same way as the correct clues with too large a search space, that is by not terminating within an acceptable time.

These clues are therefore presented as an illustrative sample of the sorts of errors that prevent correct parse (**6.3.4**) or correct evaluation (**6.3.1 – 6.3.3)** being generated.

6.3.1 SHINY SILVER PAPER IN THE STREET (8) (= "AGLITTER") [Guardian]

Although the correct parse is generated ([SILVER] + [PAPER IN THE STREET]), some natural language analysis would be required to derive the fact that "PAPER IN THE STREET" = "litter"

6.3.2 PLAYWRIGHT AT HOME HAVING CAUGHT DISEASE (5) (="IBSEN") [Everyman]

This clue requires two pieces of category knowledge, firstly that Ibsen is a member of the set of playwrights (and not a synonym for playwright), and that BSE is a member of the set of

diseases

6.3.3 HE SCORED HARLEM WINDS (6) (= "MAHLER") [Guardian]

Not only is knowledge of composer Gustav Mahler required, but also a cryptic understanding that 'HE SCORED' can refer to a member of the set of male composers. Note that this is structurally different from the examples above: while (1) was a more oblique version of a synonym (litter **is** paper on the street), and (2) is membership of the set of of playwrights, we must now consider the set of people who fit the description "he scored", which may include composers, sportsmen, and maybe even engravers.

6.3.4 WHERE AND HOW A SUPERHERO MIGHT LABEL HIS FAUCET (4) (= "BATH") [Guardian]

This clue requires not just specialist knowledge, but also natural language parsing of the sentence of a whole. The answer can be derived from the concept that the superhero Batman would append bat- onto the names of objects (batmobile, etc.), and that a hot tap (or faucet) might be labeled H, so his faucet might be labelled BAT-H.

Along with that, the definition bears reference to the clue as a whole, and may be properly expanded as:

$$\overbrace{\textit{where a superhero might label his faucet}}^{\texttt{definition}} \textit{ and } \overbrace{\textit{how a superhero might label his faucet}}^{\texttt{wordplay}}$$

This clue represents the upper level of challenge for a computer based solver, being unique structure, self referential, using very specialist knowledge and oblique humour.

Part IV
Optimizations

7 Algebraic + computational simplifications

7.1 Pruning out equivalent trees (Canonization)

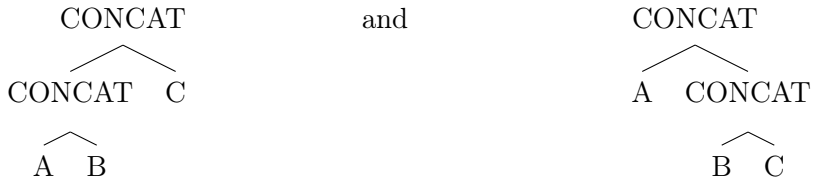
7.1.1 Motivation

One large factor in the rapid proliferation of what? is in the our binary tree representation of concatenation. While keeping them in a similar representation to the rest of the expression

nodes in our naive solution kept their representation in a similar form to the rest of the nodes, the fact that no indicator is required to generate a concatenation node means that any expression of two ore more words can generate them.

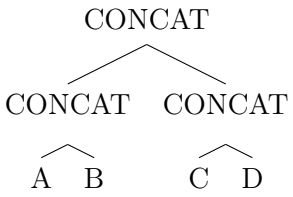
The number of trees with n leaves is given by the $(n-1)^{\text{th}}$ Catalan number[7], so ignoring any other type of expression (anagram, etc.), for a clue of length n we have C_{n-1} trees created with each of the clue words taken as a synonym node. This number grows rapidly as the clue length increases, and yields an increasingly large number of parses.

Due to the associativity of concatenation, each of these parses evaluates to an identical output:

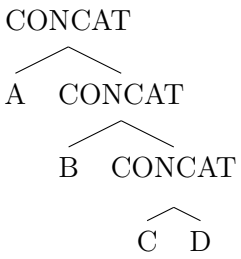


both yield the output `ABC`. This means that much of our outputted parses are identical and therefore redundant.

One strategy to deal with this would be to perform canonization on the trees, and prune all concatenation trees which don't conform to our decided 'ideal tree'. For example, we could choose to create a right-handed binary tree, wherein trees such as:



would become



The problem with this solution is that we need to look ahead while parsing: the above parse only is acceptable if the parse of `A` also doesn't produce a concatenation – this means we can't parse recursively as before.

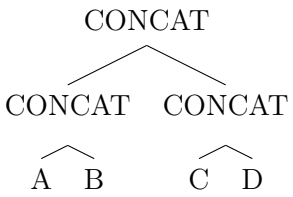
[7]Catalan numbers are given by the formula $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n} \frac{n+k}{k}$ for $n \geq 0$.

7.1.2 Implementation

We can instead define a new version of our Concatenation Nodes which, instead of describing a binary tree by storing the data as two parsetrees:

```
How did I arrive a this?  Maybe suggest it and then show this is the
answerdata ParseTree = ConcatNode ParseTree ParseTree | [...]
```

structured as:




```
(as well as 4 other equivalent trees)
```

instead stores it as a forest, i.e. a list of trees –

```
data ParseTree = ConcatNode ParseForest | [...]
type ParseForest = [ParseTree]
```

structured as:

We define a new version of parseConcatNodes to reflect the new structure. This time, instead of considering all the ways to partition the wordplay of the clue into two parse, and subsequently combining each of the different parses of both of them, this time we need to consider all the ways to partition the string (which

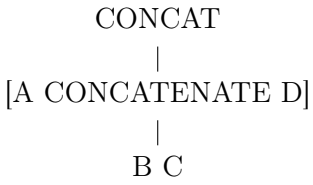
```
parseConcatNodes' :: String -> [ParseTree]
parseConcatNodes' xs n = let parts = partitions xs
       in [ConcatNode ys | part <- parts
                 , (length part) > 1
                 , ys <- [sequence . map parseClue $ part]  ]
```

the Prelude function `sequence`, which has the type `sequence ::  Monad m => [m a] -> m [a]`, which when applied to a list of lists will provide all lists comprising of an element from each sublist:

```
sequence [ [1,2,3], [40,50], [666,777,888] ] =
  [ [1,40,666],[1,40,777],[1,40,888],[1,50,666],[1,50,777],
    [1,50,888],[2,40,666],[2,40,777],[2,40,888],[2,50,666],
    [2,50,777],[2,50,888],[3,40,666],[3,40,777],[3,40,888],
    [3,50,666],[3,50,777],[3,50,888] ]
```

7.1.3 Avoiding Nesting

Unfortunately, this solution alone will not prevent us from creating a forest of parse trees that itself contains a concatenation node:



leading to a even more parse trees than before!

In order to prevent our new concatenation nodes nesting again give examplewe need to define a version of `parseClue` which doesn't generate concatenation nodes:

```
parseClueNoConcat :: String -> [ParseTree]
parseClueNoConcat ys = let len = length . words $ ys in
  [SynonymNode ys]
  ++ (if len > 1 then parseConcatNodes ys else [] )
  ++ (if len > 1 then parseAnagramNodes ys else [] )
  ++ (if len > 1 then parseHiddenWordNodes ys else [])
  ++ (if len > 2 then parseInsertionNodes ys else [])
  ++ (if len > 2 then parseSubtractionNodes ys else [])
  ++ (if len > 1 then parseReversalNodes ys else [])
  ++ (if len > 1 then parseFirstLetterNodes ys else [])
  ++ (if len > 1 then parseLastLetterNodes ys else [])
  ++ (if len > 1 then parsePartialNodes ys else [])
```

and re-define our original `parseClue` as

```
parseClue :: String -> [ParseTree]
parseClue (Def def ys n) = let len = length . words $ ys in
  parseClueNoConcat ys
  ++ (if len > 1 then parseConcatNodes ys else [] )
```

7.1.4 Improvement Analysis

By cleaning up the redundancy in our different parses, we can improve our parsing function from exponential growth against clue length, to a low quadratic growth, as can be seen in **Figure 5** and **Figure 6**. As each parse may have thousands of solutions, this should represent a significant improvement in the number of outputs, and so the solve time, of each clue.get rid of exponential one

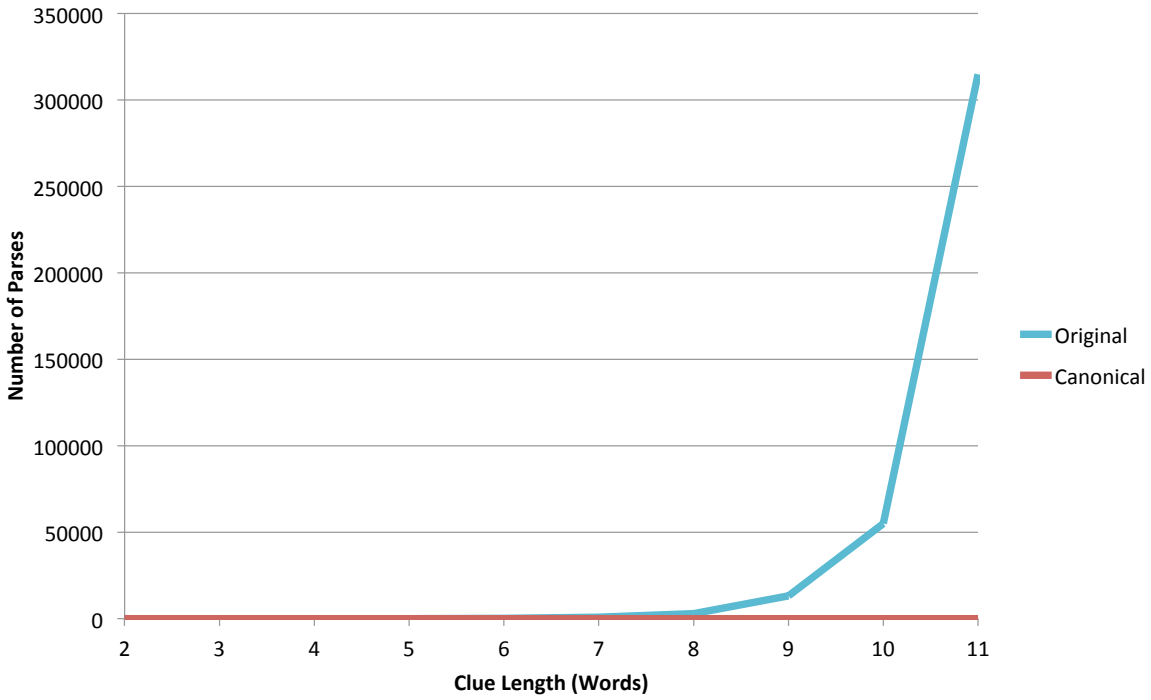


Figure 5: Average number of parses before and after canonization by clue length, averaged over 710 clues

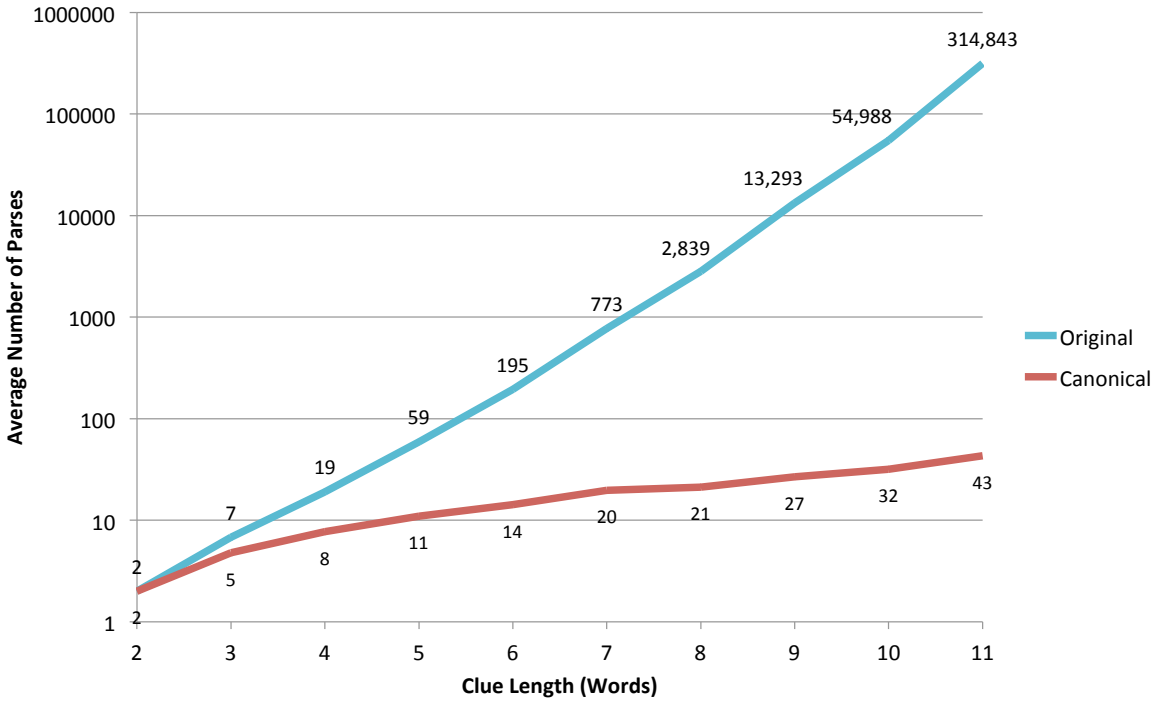


Figure 6: Average number of parses before and after canonization by clue length, averaged over 710 clues, on a logarithmic scale

8 Heuristics from Human Solvers

We can take cues for further improvements to our solving process by considering the heuristics that a human solver uses to navigate the huge state space and find the correct solution without having to enumerate all possible solutions.

8.1 Filter parses by output length

8.1.1 Motivation

Here we seek to mimic the following thought processes of a human solver:

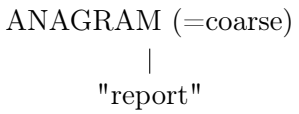
> *"This can't be an anagram of that word, as that'd only make 6 letters, and the clue is 9"*

> *"We can't have an insertion here, as we've already got 5 letters, and so if we add another 5 then it's too long"*

These are constraints on the parses that we can generate based on an understanding of the maximum and minimum number of letters than a given reading of a clue could produce. In the first example, a clue such as

```
Report coarse players (9)
```

could identify 'coarse' as a possible anagram indicator, and yield a parse such as

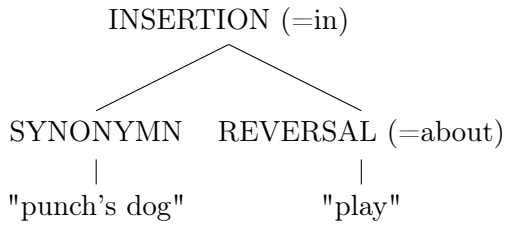


This, however, can never yield a solution that is 9 letters long, so a human solver, and so our improved computer solver, will not consider it for further evaluation.

In the second example, we see that we may also need to consider the parse recursively to calculate the total length parameters:

```
Punch's dog in play about bishop (4)
```

can be parsed to



Since the reversal of play ('yalp') is already 4 letters long, we can see that which ever word we choose to signify `punch's dog` will increase the length of the evaluated solution over the prescribed solution length of 4.

8.1.2 Implementation

We can recursively evaluate a parse to determine its maximum and minimum lengths, to check that the maximum is at least as big as the desired output length, and the minimum is at least as small.

We define the functions `minLength` and `maxLength` :

```
minLength :: ParseTree -> Int
minLength (ConcatNode trees) = (sum . map minLength) trees
minLength (SynonymNode string) = let x =
             minimum ( map length (string : syn string)) in x
minLength (AnagramNode ind strings) = (length . concat) strings
```

```
minLength (HiddenWordNode ind strings) = 2
minLength (InsertionNode ind tree1 tree2) = (minLength tree1)
                                          + (minLength tree2)
minLength (SubtractionNode ind tree1 tree2) = min (
                           (minLength tree2) -          (maxLength tree1)) 1
-- and definitions for other clue types
```

Some clue types can be defined directly from their inputs – both the maxium and minium length of an anagram node is the length of the input string – while an insertion node need to be defined based on the maxium and minimum of the two subtrees.

Notable is that here we see some 'contextual bleed' from evaluation across into the parsing, as we consider the semantics of what the thesaurus could yield for a synonym node in determining its minimum length.

It's also worth noting that sometimes we need to make a judgement: what is the minimum that a Hidden Word could yield?

From these definitions, and similar ones for maxLength, we can check a parse for validity.

```
valid_parse_length :: Parse -> Bool
valid_parse_length (Def d clue n) = (minLength clue <= n)
                                 && (maxLength clue >= n)
```

and so redefine parse as

```
parse = filter valid_parse_Length . concatMap parseClue
```

8.1.3 Analysis

Figure 7 shows the effect on number of parses generated following the addition of the parse length constraints.

This filtering constraint now means that many clues now yield 0 parses. Some of these are clues that could never be correctly parsed, while some are clues which we can generate correct parses, but do not have the thesaurus and synonym data to solve the clue.

This transformation is, though, safe – any parse that previously would have generated the correct answer will not be filtered out.

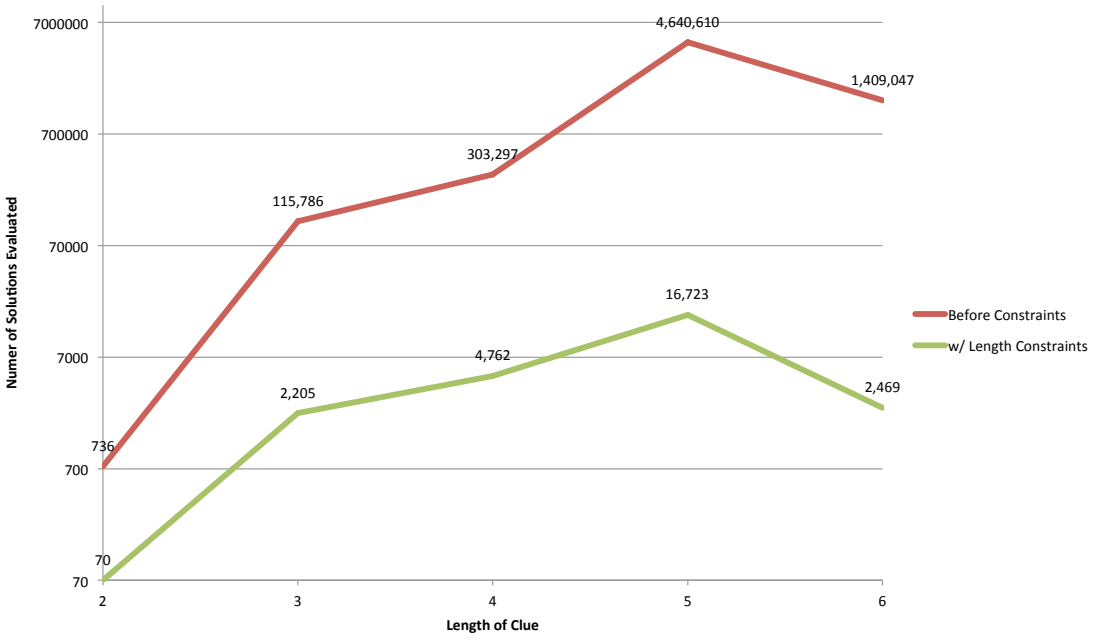


Figure 7: Average number of parses before and after canonization by clue length, averaged over 710 clues, on a logarithmic scale

8.2 Taking Advantage of Lazy Evaluation

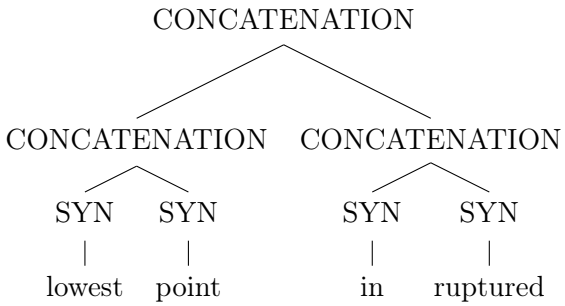
8.2.1 Motivation

Although we are now generating far fewer parses, we still have some solveable clues generating hundreds of parses. This means that for these clues we will have to perform on average $n/2$ evaluations to find the right clue, assuming it will be randomly distributed down the list – lazy evaluation means that our use of the `head` with `filter` will yield the first result computed from the head of the list toward the tail.

We could take further advantage of the intuition that some parses are more likely, given the input words. For instance in the clue `Lowest point in ruptured drain (5)`, we see the angram indicator 'ruptured' next to a 5 letter word:

$$\begin{array}{c}\text{ANAGRAM (=RUPTURED)}\\ \Big|\\ \text{drain}\end{array}$$

with the definition `lowest point`, intuitively feels more likely than



cluing the definition `drain`.

More formally, we're looking for a heuristic which weights toward consuming words into indicators for more 'interesting' clue types: in that clues using expressions more varied than synonym and concatenation are considered better clues, and so are more likely than not if they are an available parse.

Furthermore, these expressions consume more of the string in indicators than other types (reversal nodes consume one word from the clue as its indicator, while synonyms and concatenation both don't consume any indicators)Diagram to show this and are less likely to produce nested parse trees (both anagrams and hidden word nodes treat their input as a pure string to be transformed, and so do not generate any nested parse trees). This means that clues featuring these types tend to be less complex.

Both of these factors make them good candidates to evaluate sooner than other options.

8.2.2 Implementation

We define a method `cost` which gives a weighting to a given ParseTree

```
cost :: ParseTree -> Int
cost (ConcatNode trees) = 2 * (length trees) + sum (map cost trees)
cost (AnagramNode ind strings) = 1
cost (HiddenWordNode ind strings) = 4
cost (InsertionNode ind tree1 tree2) = 4 + cost tree1 + cost tree2
cost (SubtractionNode ind tree1 tree2) = 3 + cost tree1 + cost tree2
cost (ReversalNode ind tree) = 2 + cost tree
cost (SynonymNode string) = 8 * length (words string)
cost (FirstLetterNode ind strings) = 2
cost (LastLetterNode ind strings) = 2
cost (PartialNode ind tree) = 6 + cost tree
```

we can then define

```
May want to rationalize a styleguide for code re:  underscores or
not
```

```
cost_parse :: Parse -> Int
cost_parse (DefNode s tree n) = cost tree * (length_penalty s)

length_penalty :: String -> Int
length_penalty ws = 60 + (length (words ws))
```

which can then be integrated into our definition of `parse`:

```
parse = sortBy cost_parse . filter valid_parse_length . concatMap parseClue
```

It should be noted that the weights here are intuitive only. Woah say much more about why we do this Pull out params into constant, and then give what I chose as mine

Synonym nodes have a high weighting against consuming long lists of words – this is to prevent them from being low scoring (as they consume large portions the clue) while being unlikely to yield the correct answer.

8.2.3 Analysis

The weighting above mean that the correct parse had the highest score in 70% of the clues that the system can solve, as opposed to approximately 10% when not sorted by weight. In cases where the clue can not be solved, the order of the parses is irrelevant.

Generate much more data for this and display in a nice way.

8.2.4 Determining a correct weighting

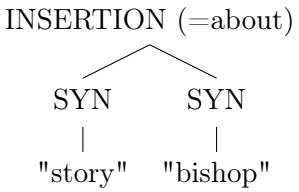
While the current weighting given has been developed though trial and error to be reasonably successful, a more structured approach to detemining the correct weighting could generate even better results. Using a large dataset of clues and the correct parses, hill climbing or statistical analysis of clue types could produce optimal numbers.

8.3 Constrain length while evaluating

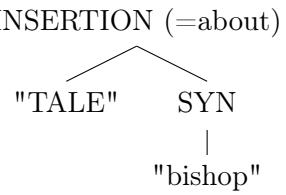
8.3.1 Motivation

While evaluating a parse tree of a given clue, we expect the overall length of the generated solution to be equal to the length specified in the clue. Furthermore, while evaluating different sub-trees of a given parse tree – either different branches of a concatenation list, or the two constituent parts of an insertion or subtraction experession – the solutions generated from one influence and limit what can be generated from the others.

For example, in the clue `Story about bishop and food (5)`, if we are evaluating the parse



then the partial evaluation of the left hand branch to one of its possible solutions



means that as the subsequent evalaution of `SYN ‘‘bishop’’` should only yield solutions that are one-letter one, in order to stay within the constraint of a five-letter solution. If we can successfully apply this constraint, then we can limit the subsequent evaluations of this partial parse tree to one or two, rather than the order of 100.

8.3.2 Implementation

In the example above, we see the length constraints preventing overflow - that is, a maximum length which the generated solution should not exceed. We also need to constrain against 'underflow', wherein the evaluation fails to yield enough letters to fit the solution. Constraining both maximum and minimum length will have the effect of forcing the generated solution length to be equal to the prescribed length.

We can therefore define a datatype to carry both of these constraints.

```
newtype Constraints = Constraints MaxLength MinLength
```

In some cases, we will not be able to prescribe a definite maximum length for a clue: in the case of a subtraction expression of parse trees A and B, where the evaluation of tree A will have the evaluation of B removed from it to yield the final solution, the length of clue A will exceed the overall solution length by an amount only limited by the length of B.

We also, then define `MaxLength` and `MinLength` as new datatypes.

```
data MaxLength = Max Int | NoMax
data MinLength = Min Int | NoMin
```

Although we could use the `Maybe` monad here, by defining our own datatype we can subsequently take advantage of Haskell's type class system later on to allow us to treat these, and other constraints, in a similar way.

We define `is_lte_max` and `is_gte_min` to account for both the case when we have a defined constraint (e.g. `Max 3`), as well as when we have no constraint (e.g. `NoMax`):

```
is_lte_max :: MaxLength -> Int -> Bool
is_lte_max (Max mx) n = n <= mx
is_lte_max NoMax n = True
is_gte_min :: MinLength -> Int -> Bool
is_gte_min (Min mn) n = n >= mn
is_gte_min NoMin n = True
```

We can define a typeclass `Constraint` which gives us the ability to define the method for checking if a given string fits a constraint of either type.

```
class Constraint c where
    fits :: c -> String -> Bool
instance Constraint MaxLength where
    fits mx s = is_lte_max mx (length s)
instance Constraint MinLength where
    fits mn s = is_gte_min mn (length s)
```

which subsequently allows us to write a function to check if a given output string fits the each of the constraints:

```
fits_max (Constraints mx mn) x = fits mx x
fits_min (Constraints mx mn) x = fits mn x
```

and so can define an overall function for checking a string against all our constraints:

```
fits_constraints c x = (fits_max c x) && (fits_min c x)
```

This allows us to start to redefine our definitions of `eval_tree`, with the new type signature

```
eval_tree :: ParseTree -> Constraints -> [String]
```

For simple synonym nodes, we can apply the check in a straightforward manner, as there are no subtrees to evaluate.

```
eval_tree (SynonymNode xs) c = filter (fits_constraints c) (synonyms xs)
```

For anagram nodes, redefining in the same way, as

```
eval_tree (AnagramNode xs) c = filter (fits_constraints c) (anagrams xs)
```

would still require the costly computation of all our anagrams. Instead, we can use our min and max criteria on the input string to check if it's worth evaluating at all:

```
eval_tree (AnagramNode xs) c = if ((fits_max c xs) && (fits_min c xs))
                               then filter (fits_constraints c) (anagrams xs)
                               else []
```

While here the first line could be replaced with if `fits_constraints c xs`, we avoid this, as it's only due to the fact that angrams preserve length that the max and min constraints are applicable to the initial string as a filter for its output. Once we add other constraints later which aren't preserved over the anagram operation (anything involving letter order!) then we would violate this condition.

Often, we will want to change the constraints on the evaluation of the subtrees in a parse tree. For example in the clue

PARTIAL
|
SYN
|
"word"

we can't easily define a maximim length for our evaluation of synonym, as an unspecified amount of letters will be removed when we apply the Partial expression. We need to define function which modifies our constraints to remove the maximum length constraint, as well as a similar one for the minimum:

```
noMax :: EvalConstraints -> EvalConstraints
noMax (Constraints p mx mn) = (Constraints p NoMax mn)
```

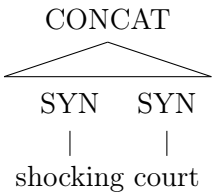
fo

```
noMin :: EvalConstraints -> EvalConstraints
noMin (Constraints p mx mn) = (Constraints p mx NoMin)
```

we can then define the eval_tree function for partial nodes as

```
eval_tree (PartialNode ind y) c =
      filter (fits_constraints c) . concatMap partials $ eval_tree y (noMax c)
```

For our concatentation nodes, the requirement is slightly more complex, as the constraints imposed on the parse of any one tree depend on the each possible evaluation of all the other possible trees in the forest. In a simple example: From a clue such as`Outspoken shocking court (6)`, we take a parse such as



and look at the evaluation criteria at each stage. The `ConcatNode` will be given the constraints `Max 6 Min 6`. We will the start to parse the subtrees in the forest of the concatenation. Parsing left-to-right, we parse the left `SynonymNode` subtree with the constraints `Max 6 NoMin`. Two of the partial evalautions we might come to are:

1. and 2.

We can see here that in partial evalaution 1, the constraints that we need to apply to the right `SynonymNode` will be `Max 2 NoMin`, whereas in 2, it will be `Max 1 NoMin`. This means we will need to evaluate each subtree in the forest of a concatenation expression with different constraints depending on the outcomes of the evaluation of previous evaluations.

We define a function to allow us to decrease the maximum length constraint as we go further down the list (and also similar for increasing, and for changing the min constraint).

```
decreaseMax :: Int -> EvalConstraints -> EvalConstraints
decreaseMax n (Constraints (Max mx) mn) = Constraints  (Max (mx - n)) mn
decreaseMax n (Constraints NoMax mn) = Constraints NoMax mn
```

and a function which updates the constraints given a string we've just generated:

```
add_partial :: String -> EvalConstraints -> EvalConstraints
add_partial x c = decreaseMax (length x) c
```

So in this example, applying the function `add_partial ‘‘dire’’ (Constraints Max 6 NoMin)` will give the correct constraints for the second tree's evaluation: `Max 2 NoMin`.

We can then define a function `eval_trees` that will handle passing the right constraints down the list of

```
eval_trees :: [ParseTree] -> EvalConstraints -> [String]
eval_trees (x:[]) c = eval_tree x  c
```

```
eval_trees (x:xs) c =
       let starts = [start | start <- eval_tree x (noMin c)]
         -- Generate options for the first in our list
       in concatMap f $ starts
         -- For each option, evaluate the rest with updated constraints
       where f start = map (\x -> start ++ x) (eval_trees xs (apply_partial start c))
         -- Append from each possible evaluation, after updating constraints
```

overall

Now we can re-define `eval_tree` for the concatenation expression

```
eval_tree (ConcatNode xs) c = eval_trees xs c
```

Finall, we can re-define our definition of `eval` to set the initial top-level maximum and the minimum constraints to the length of the required answer

```
eval (Def d pt len) = [Answer x (Def d pt len) |
                                          x <- evalTree pt (Constraints Max len Min len)]
```

8.3.3 Analysis

Do analysis of graph - reference exponential increase

8.4 Constrain against known letters

8.4.1 Motivation

Here again we turn to behaviour of a human solver:

> *"It can't be that, as there are no words that start with "*
> Insert a good example here, that we can use later on too

Currently, we are performing a large amount of wasteful evaluations on subtrees that will never be ouputted as a valid answer, as we already know that given the the preceding letters that we've already evaluated, there are no possible words in our wordlist that we can make.

Add some illustrated examples

8.4.2 Implementation

We want to add another contraint while solving, which is a check that what we are evaluating can be a prefix of a valid word. We add a function to calculate all proper prefixes of a given word

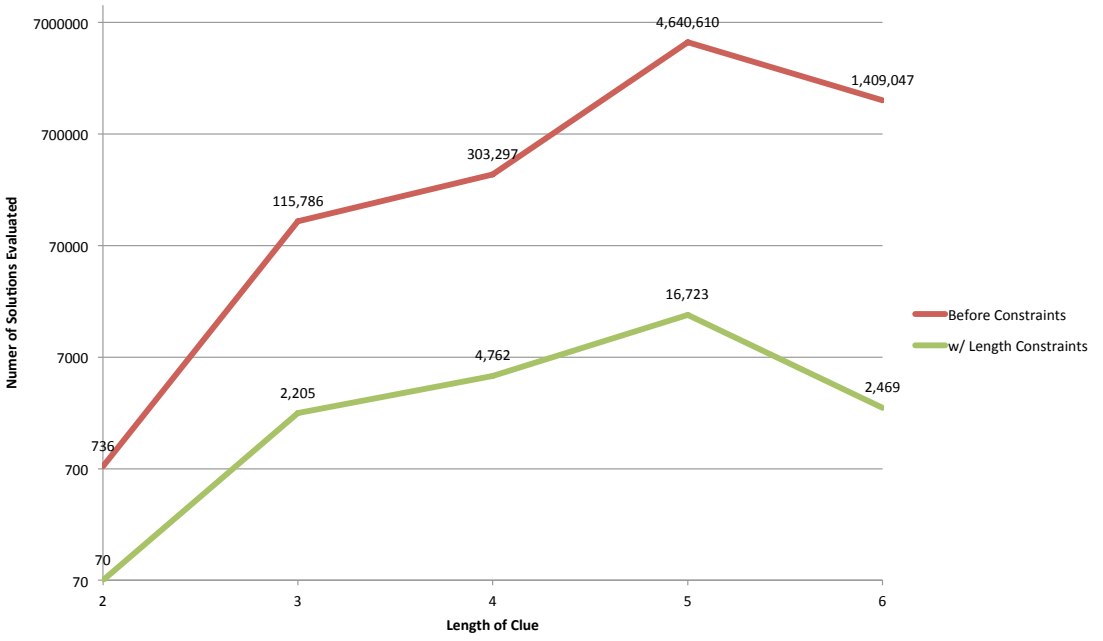


Figure 8: Number of solutions evaluated before and after implementing length-based constraints

```
prefixes :: String -> [String]
prefixes = rprefixes . reverse
rprefixes (x:xs) = [reverse xs++[x]] ++ rprefixes xs
rprefixes [] = []
```

and some functions to precomputer a set of prefixes[8] for our dataset and check if a given word is a prefix

```
is_prefix x = member x wl_prefixes
wl_prefixes = fold add_prefixes empty wordlist
add_prefixes word set = union (fromList (prefixes word)) set
```

We want to add prefix constraints alongside the current maximum and minimum length constraints, to take advantage of the current mechanisms we have set up to propegate the prefixes down the nested parse trees. We update the type definitions and create similar functions for our new constraint:

```
data EvalConstraints = Constraints PrefixConstraint MaxLength MinLength
```

[8]For a very large wordlist it may be preferable to create a prefix tree. For my dataset, however, I found the extra memory footprint to be an acceptable tradeoff for constant-time lookups

```
data PrefixConstraint = Prefix String | NoPref
is_prefix_with (Prefix p) x = is_prefix (p ++ x)
is_prefix_with NoPrefix x = True
class Constraint c where
[...]
instance Constraint PrefixConstraint where
fits p s = is_prefix_with p s
extend_prefix_by x (Constraints (Prefix p) mx mn) = (Constraints (Prefix (p++x)) mx mn)
extend_prefix_by x c = c
add_partial x  = decreaseMax (length x) . extend_prefix_by x
noPrefix (Constraints p mx mn) = (Conare straints NoPrefix mx mn)
```

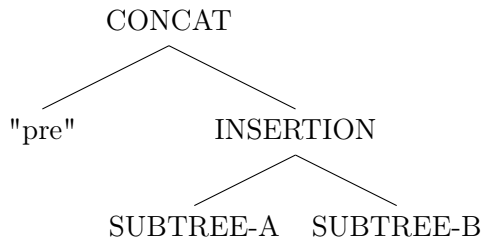
It is worth noting that `NoPref` and `Prefix ""` are not equivalent: the former means that there are no prefix-based constraints on the evaluation, while the second means that the evaluation is taking place a the start of a solution, and so all sets of letters generated by that parse will need to be valid prefixes of a word in the wordlist.

Some clue types will not require any changes for this to work:

```
eval_tree (SynonymNode x) c = filter (fits_constraints c) (syn x ++ [x])
```

while others which have subparts which generate letters which are subtracted from or used out of sequence will need to ignore the prefix constraint for the evaluation of their subtrees.

For example, in the following illustrated partial evaluation of a clue:



We can see that the prefix constraint on the insertion node is that anything it generates must be able to be added to the prefix "pre". It's not, however, possible to pass that constraint down to its sub trees. We do not yet know where subtree B, which is going to be inserted into subtree A, is going to go, so we know nothing about the letters immediately preceding it. Furthermore, as we don't know where into A it is going to be inserted, we can't apply the prefix constraints to A either, as the only letter of A we know for sure will be sequentially following "pre" will be its first one.

Not finished writing here!!!

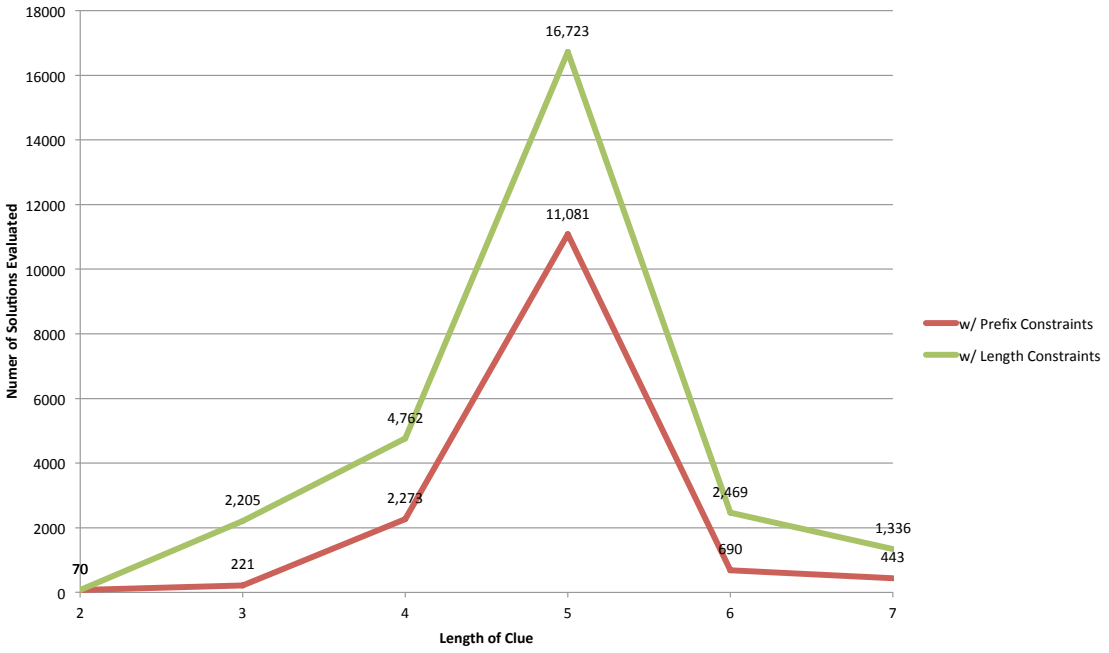


Figure 9: Number of solutions evaluated after implementing prefix-based constraints

```
eval_tree (InsertionNode ind x y) c = concat[insertInto x' y' |
        y' <- eval_tree y (noMin . noPrefix $ c),
        x' <- eval_tree x c',
        c' = (decreaseMax (length y') . decreaseMin (length y') . noPrefix $ c)]
```

8.4.3 Analysis

Do some analysis of this vs figure - moderate improvement

9 Analysis of Single clue solving against test suite

9.1 Test Suite

Make sense from this These clues are evaluated against a testing suite comprising of 10,000 clues extracted from the Observer's Everyman series. These were chosen for being published in a major British newspaper, and for being both scrapable from publically open websites (as The Times' and Telegraph's are not) and for being Ximenean[9] (as the Guardian's are not).

The clues selected have been limited to those with single-word answers, as few of the multi-word answers appear in wordlists. Clues with numbers are not included, as they are often the self-referential type (see the section on **Meta-reference**), and thus cannot be solved in isolation.

9.2 Thesaurus and Knowledgebase data

Write about data sources here

9.3 Solvable Clues

An analysis on the accuracy of the program's solving capabilities can be found in **Figure 10**check reference is up to date. or fix the damn auto reference thing. The details of each individual status are discussed here.

9.3.1 Solveable and verifiable

If a clue is solvable and verifiable it means that the program succesfully generated the correct split of definition and wordplay, generated a correct parse tree, correctly evaluated the parse tree to the correct answer, and verified the answer using the definition. One example of a clue solved this way is

```
Rule amended to include married primate (5)
```

for which the program generates the parse tree

```
(Def "primate" (InsertionNode (IIndicator ["to","include"])
                        (SynonymNode "married")
                        (AnagramNode (AIndicator ["amended"]) ["rule"])))
```

along with the correct answer

[9] Macnutt, as Ximenes, was the first setter for the Everyman in the 1940s.

```
Answer "lemur"
```

which it can successfully match to the definition of primate based on our thesaurus/knowledgebase.

15% of previously unseen clues from the test suite could be solved in this way – this figure was derived by running the program over the entire testing suite and filtering for where the generated answer could be verified against the definition, and where it matched the correct answer from the test suite.

Along with clues which couldn't be accurately verified, there were also some 'false positive' answers: clues where there was a solution which could be verified but did not match the correct answer as expected by the test suite. One example `Teases Spurs (4)`, for which the correct answer is `RIBS`. My knowledgebase didn't contain the equivalence between 'ribs' and 'spurs', but the program generated the answer `SETS`, drawing off the senses 'sets' = 'besets' = 'teases' and 'spurs' = 'starts' = 'sets'.

While this, and others like it, are not the correct answers from the original context of the clue, and would likely not fit in the completed grid, in isolation they are valid answers for the clues themselves - although sometimes 'low quality' answers based on more spurious semantic links, as in the example given. Around 2% of the clues in the Solveable and verifiable category were false positives.

9.3.2 Solveable but not verifiable

Clues in this category succesfully generated the correct split of definition and wordplay, generated a correct parse tree, correctly evaluated the parse tree to the correct answer, however didn't manage to match that answer to the definition. Sometimes, multiple answers could be produced, most of which would not be valid answers for this clue.

For example the clue

```
A new member returned with a backer (5)
```

will correctly return the solution `ANGEL`, but cannot match it to `A BACKER`. It also generates other answers, such as

```
Answer "inarm"
(Def "returned with a backer"
      (ConcatNode [SynonymNode "a",SynonymNode "new",SynonymNode "member"])
```

which, to the system, are equally valid readings as the correct one, as there is no semantic link available for either.

Clues in this category will often take orders of magnitude more time to solve, as all solutions need to be generated. Because of the extensive time taken to solve, the figure of 13% was generated by sampling over 700 clues from the testing suite.

9.4 Unsolvable

Continuting to refer to **Figure 10**, categories from here onwards were not solvable by the program. In order to analyse these clues, a random sample of 100 clues that were not correctly solved and verified were drawn from the testing suite, and examined by hand to determine the correct parse, and the factors missing from the data in order to solve them. These were then assigned one or more of the following labels:

- Answer not in wordlist
- Expression Indicator Not Found
- Unparsable Structure
- New/unknown clue type
- Knowledge not in dataset
- Synonym required in clue not in dataset
- No dictionary match between Answer and Definition

The frequency of these labels in this group can be seen in **Figure 11update figure reference.**

9.4.1 Solvable with easily collectable data

Clues in this category are those which recieved only the labels 'No dictionary match between Answer and Definition', 'Answer not in wordlist', 'Synonym required in clue not in dataset', 'Expression Indicator Not Found'. I have deemed these to fall into the category of 'easily collectable data' – that is, data that is finite or has a clear scope and could be consumed by the system in the same way as other data.

Answer not in wordlist In order only to output useful words and to limit useless evaluations, large wordlist is used in addition to the knowledgebase. If a word is not in the wordlist, then it cannot be given as a solution. Thus, in the clue

```
Girl feeding pygmy rattlesnake (4)
```

the answer `Myra` is not given, even though the program can generate the right parse tree.

These issues could be resolved by collecting a larger wordlist including, for example, proper names, places: one possible source for this information would be Wikipedia article subjects, along with commerically available listings.

Synonym required in clue not in dataset These are clues that generate a valid parse but cannot be solved as the equivalence information is not there to perform the correct evaluation. For example

```
Exaggerate concerning party (6)
```

which should yield `OVERDO`, but the current thesaurus data lacks the link 'over' = 'concerning'. Clues in this category lack only the sort of synonym-based information one might find in a very thorough thesaurus. Any more complex data such as membership (Handel is a composer, etc.) is covered under the category of **Knowledge not in database**.

This could be rememdied by providing a more thorough and permissive thesaurus than the one integrated into the knowledgebase currently.

No dictionary match between Answer and Definition This has the same properties as the examples above.

Expression Indicator Not Found In this case, the clue contains an expression type that we can generate parse trees for with an indicator word that we haven't defined. For example, in

```
Last in science failing to pass (6)    (= ELAPSE)
```

we fail to parse "last in science" as a final letter expression with the indicator "last in".

Most of these are common indicators which occur frequently by convention in crosswords, and could be easily collected manually, and extracted from crossword solving guides to give a much greater coverage than the system currently offers.

9.4.2 Require more complex data

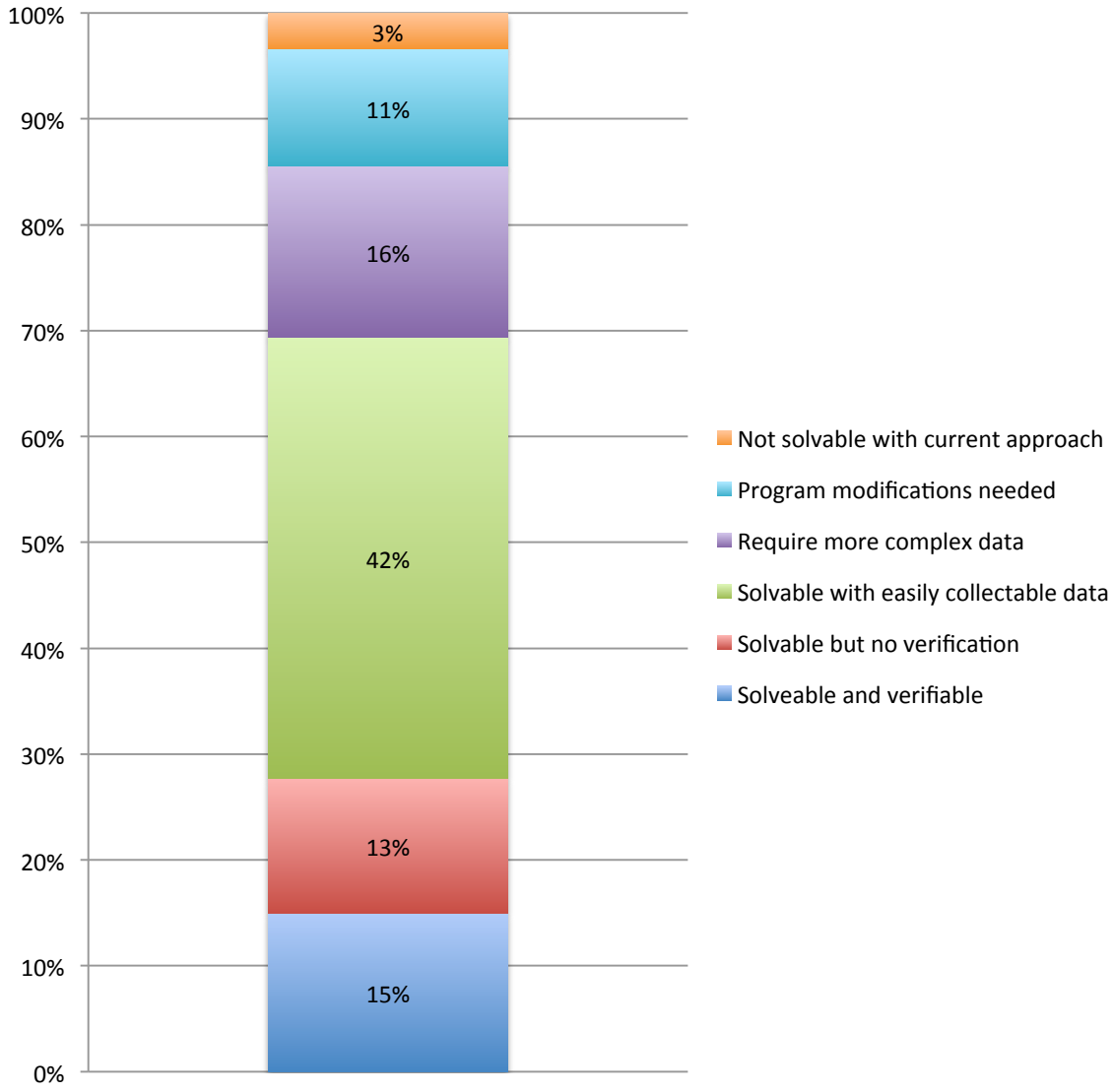


Figure 10: Breakdown of solvability of clues from the testing suite

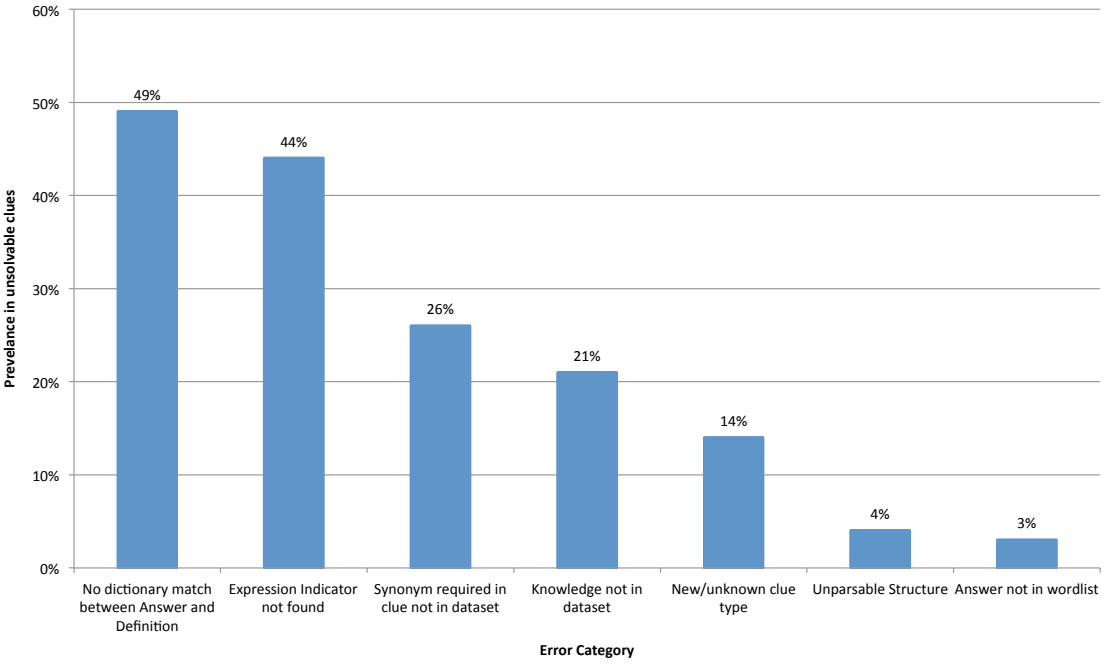


Figure 11: Reasons for unsolvable clues

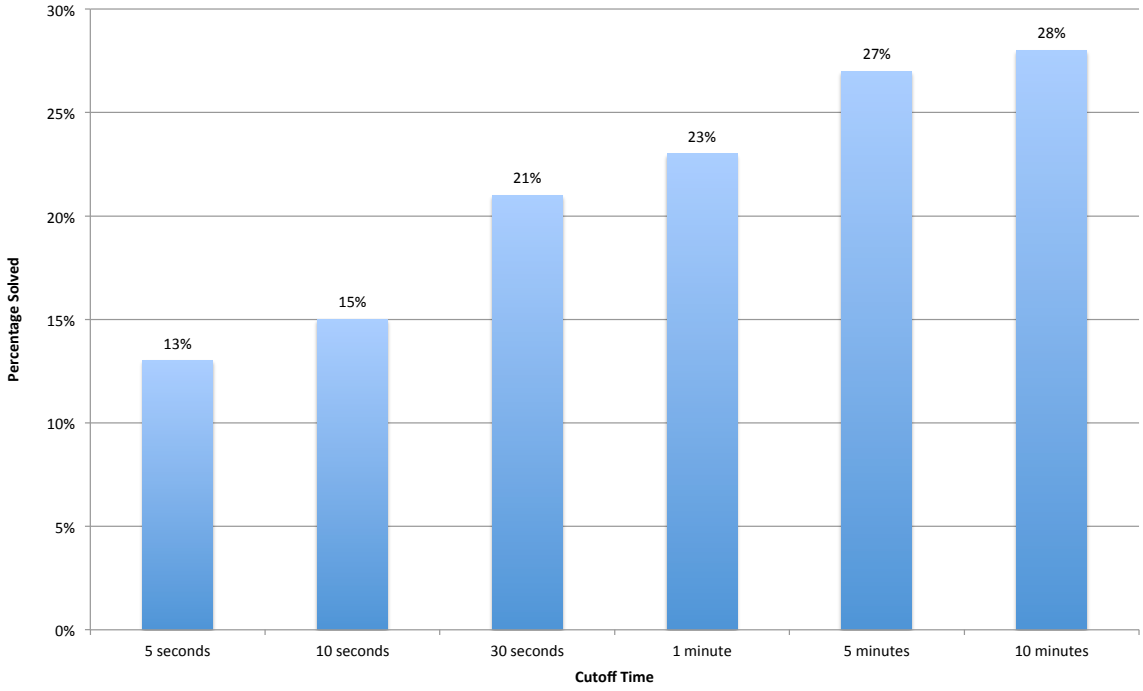


Figure 12: Percentage of clues solvable before cutoff

9.5 Performance Discussion

Write about solving time, RAM required etc. Maybe mention Watson, for comparison.

9.5.1 Feasibility

Part V
Future Work

10 Adding new clue types

Analysis of un-answerable clues has highlighted types of clues expressions not found in the literature. These include:

Language Clues Phrases like "man in Paris" or "Spanish article" indicate a translation (in these examples, to "homme" and "el", "un" etc.)

add some more examples of the other clue types -> 'after', etc..

In order to simplify the process of adding additional clue types, we can observe that each clue type is characterized by a specific pattern in just a few key functions: `eval_tree`, `parseClue`, `cost`, `maxLength`, `minLength`. We could modularise our system and make it more easily extensible for new clue types by encoding this information in a `NodeType` record:

```
data NodeType = NodeType {
    eval_tree  :: ParseTree -> [String],
    parseClue  :: String -> [ParseTree],
    maxLength  :: ParseTree -> Int,
    minLength  :: ParseTree -> Int,
    cost       :: ParseTree -> Int,
 }
```

11 Improving current solving capabilities

11.1 Improving the knowledgebase

Currently, a large quantities of clues are unsolvable due to missing information. Most of the current information comes from thesaurus definitions, and is stored in a directed graph structure, similarly to a thesaurus: each word is connected to all the words it is in some way equivalent to. This is a clumsy representation of the real world: we lose the information that 'dog' related to 'poodle' in a different way from the way it relates to 'mamal' (hyponymically, and hypernymically, respectively). Furthermore, if we want to augment the knowledgebase with further information about dogs (dogs = man's best friend), then we'd also have to add that fact to all hyponyms ('poodle', 'labrador', and so on). If we wanted to add a propagatable fact to something much higher level, such as 'mamal' or 'solid object', then the number of new 'facts' or graph connections we'd have to add would grow quickly indeed! Very quickly, our database would become very difficult to manipulate, or hold in RAM for quick access.

11.1.1 Using Propositional Logic

We would like to be able to infer a fact, such as the fact that Mahler was, as a composer, someone who scored[10].

Instead of doing this through direct entry into the database, we could use a prepositional logic language like PROLOG to represent this as a minimal sets of facts.

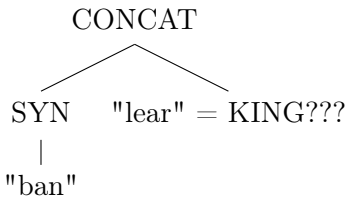
```
composer(malher).
composer(brahms).
[...]
scored(X) :- conductor(X).
scored(pele).
[...]
```

Thus, we could take 'HE SCORED', and generate a query to our logical database asking "for which X did X score" (or as an SWI prompt: `?- scored(X).`), which would return the set of everyone that can be infered by the knowledgebase to match the criteria.

With the use of knowledgebases come many additional avenues for complexity: How can we transform a natural-language phrase into an answerable question in an appropriate logic language? How can we represent sufficient amount of inference rules for the database to be useful while still dealing with exceptions (e.g. penguins are birds, birds can fly, but penguins can't fly). The field is, itself, a large and complex one, but may warrant further investigation.

11.2 Generating solutions with missing data

In cases in which we don't have suitable data to generate any solutions at all, it would be useful to generate tentative solutions with assumptions stated around the missing data. So in the clue `Ban Lear: Mad! (7)`, with correct solution "barking", the program could return



The issue is that the number of words we could generate with the ability to generate any combination of letters from any subpart is infaesibly large, especially for longer and more complex clues.

[10] In order to solve the sorts of clues we ruled out in Chapter II: HE SCORED HARLEM WINDS (6) (= "MAHLER")

11.2.1 Solving Forwards and Backwards

One solution to this is in re-working the entire method by which we solve the clues. Our solver is currently works 'forwards': based on the available clue text, we attempt to parse into a tree which, when evaluated, generates all possible outputs. Another option would be to work in reverse: from the selected definition, evaluate all words that are synonyms, and parse to match the letters in the solution with parts of the clue text. In some ways, this could be though of generating possible clues for a given solution, and matching them to the given clue.

This method could reasonably work for a simple subset of expressions, such as limiting to, for example: synonyms and concatenation. **Update ref**Figure 13 shows a possible output, illustrating how such a search could take place.

```
Select definition: Mad
Select synonym of definition: BARKING
Split definition into parts: BAR  KING
Match clue text to first part: BAR = ban - confirmed in thesaurus
Match clue text to second part: KING = lear - not confirmed
Possible solution found. Searching for more...
Split definition into parts: BARK ING
[...]
```

Figure 13: Example output from a possible 'reverse' solver

11.3 Parallelization

Write stuff about parallelization

12 Whole Grid Solving

12.1 Intersections and known letters

An obvious extension of this system is to allow it to solve whole grids instead of just individual clues. While more computation is needed to solve a whole grid of around 30 clues, this is evened out by the fact that we have more information about the clues in the form of their intersections.

This extra information would form another filterint criteria: this would need to be applied to the 'weak' solve: the list of all possible solutions that could be produced by the clue, including those that our thesaurus is unable to match as a synonym of the definition. For the 'strong' solve, this extra data is redundant – if we haven't been able to generate any answers, then further filtering is useless.

A solution to filter answers that fit a known pattern of intersected letters (in the form `CRO??W???D`) is implemented below

```
known_letter_fits :: String -> String -> Bool
known_letter_fits [] [] = True
known_letter_fits [] (y:ys) = False
known_letter_fits (x:xs) [] = False
known_letter_fits (x:xs) (y:ys) = if x=='?' then (known_letter_fits xs ys) else
                        if x==y then (known_letter_fits xs ys) else
                          False
answerFits ::  String -> Answer -> Bool
answerFits fitstring (Answer x y)  = known_letter_fits fitstring x
stripFits :: String -> [Answer] -> [Answer]
stripFits s = filter (answerFits s)
```

12.2 Solving strategy

The prolem we have now is one of recursion. As crossword grids are usually heavily intersected, we will have to deal with cycles in our intersection graph: for example in **Figure 8**, we can see many such cycles. One example: 1-across intersects 2-down, which intersects 10-across, which intersects 3-down, which intersects 1-across again! We therefore have to find a strategy to deal with this.

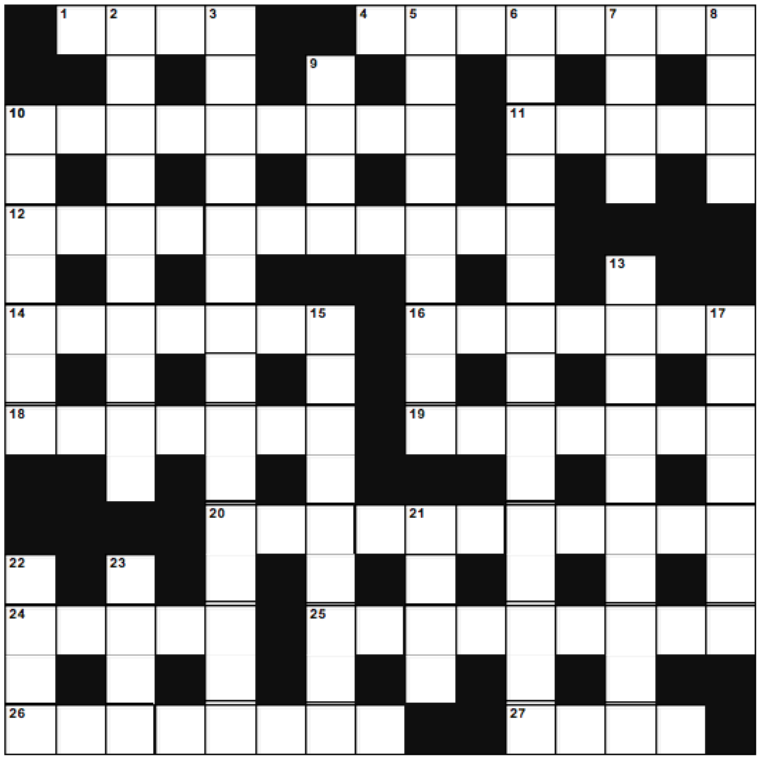


Figure 14: A crossword grid

12.2.1 All permutations

We can take the combination of all the possible words generated by each clue, and once I know more about the number of possible words generated, finish this

If we had a list of possible answers generated for each clue, and a function to check an arrangement of answers against a grid of intersections, then we could form a solution like so:

```
check_valid_intersections grid . sequence $ answerList
```

This, however, means we cannot take advantage of lazy evaluation, and that we need to compute all possible solutions of every clue. This means our remaining clues for which solving time is still very high (>10 minutes) could potentially mean that no answers at all are yielded, as the system waits for all possible answers to be generated before matching.

12.2.2 Lazy evaluation and backtracking

In a similar way to the way we evaluated the items in the parse forest left-to-right checking combinations and culling where no available solution fitted the constraints, so we could decide upon an arbitrary order to evaluate the clues, and then backtrack where necessary – for example, Show a partial grid with possible solutionslock-in the answer to clue 1, and try to solve clue 4 given those constraints. If one or more fit, then try each of them recursively in turn; if none fit, then backtrack and try a new solution to clue 1.

This method could work well, as it only requires us to compute the solutions necessary when we need them. Although we may end up with re-computation of answers, we could avoid this through memoization of the solution to each clue given a set of constraints. This could be futher improved by observing that the solutions to a clue constrained by a set of known letters (?????A) are a superset of those given tighter constraints (????MA), so further computation may be required.

Issues with this approach are that the stack may grow very large, as with around 30 clues, even a small branching factor can lead to a long parse path. This solution would also need extra work to deal with unsolvable clues: one for which no possible answers can be generated, or where only an incorrect answer is generated for a given clue.

12.2.3 Functional iteration

Another solution which may solve some of the issues of the others is by generating a finite set of solutions for each of the clues, generating a likelihood for each of the generated solutions, and then using the intersection letters of those solutions to help weight future iterations of the solve. **Table 1** illustrates how this may occur.

This solution could work correctly in the case that no solutions can be found for one clue: those missing would simply bear no weight on their intersections.

Iteration 1:

			P	U	P			
			P	I	G			
			C	**A**	**T**			
P	M	**R**	1		2	**T**	G	A
U	E	**U**		■		**I**	U	X
R	W	**B**	3			**E**	N	E
			H	**O**	**P**			
			R	U	N			
			J	O	G			

1a.	1d.	2d.	3a.
CAT	RUB	**T**IE	HOP
PUP	MEW	GUN	RUN
PIG	PUR	AXE	JOG

This is the initial solution, taking no information from other clues - solutions pictured closer to the grid represent more probable solutions.

Iteration 2:

			P	U	P			
			P	I	G			
			C	**A**	**T**			
M	R	**P**	1		2	**T**	G	A
E	U	**U**		■		**I**	U	X
W	B	**R**	3			E	N	E
			R	**U**	**N**			
			H	O	P			
			J	O	G			

1a.	1d.	2d.	3a.
CA**T**	**P**UR	**T**IE	**R**UN
PUP	RUB	GUN	JOG
PIG	MEW	AXE	HOP

Based on the available solution in 1a., PUR has become more likely than RUB, as 1a. has no solutions begining with 'R.' HOP has changed to RUN for similar reasons.

Iteration 3:

			C	A	T			
			P	U	P			
			P	**I**	**G**			
M	R	**P**	1		2	**T**	G	A
E	U	**U**		■		**I**	U	X
W	B	**R**	3			E	N	E
			R	**U**	**N**			
			H	O	P			
			J	O	G			

1a.	1d.	2d.	3a.
PIG	**P**UR	**T**IE	**R**UN
PUP	RUB	GUN	JOG
CAT	MEW	AXE	HOP

Based on the change to 1d., 1a.'s probabilities also change – the influence on its intial letter as P is now greater than the influence from its final letter.

Iteration 4:

			C	A	T			
			P	U	P			
			P	**I**	**G**			
M	R	**P**	1		2	**G**	T	A
E	U	**U**		■		**U**	I	X
W	B	**R**	3			**N**	E	E
			R	**U**	**N**			
			H	O	P			
			J	O	G			

1a.	1d.	2d.	3a.
PIG	**PUR**	**GUN**	**RUN**
PUP	RUB	TIE	JOG
CAT	MEW	AXE	HOP

Now, 2d. updates based on the changes to the other cells to reach a stable solution.

Table 1: Example of how iterative function application might converge to solution

Part VI
Appendix

13 Data considerations

13.1 Corpus / wordlist

13.1.1 Loading in an unsafe IO manner can

13.1.2 Conjugated forms -> we should match tense, plurality etc. Expanding out keywords (e.g. Anagram indicators)

13.2 Knowledge -> capital of Paris

13.3 Derived knowledge -> Qulog to create knowlegebase?

13.4 Unsupervised learned?

14 A benchmarking suite to check performance + accuracy

14.1 Clues from real newspapers

Part VII
References

Cryptic crossword clues: generating text with a hidden meaning David Hardcastle - 2007

The Generation of Cryptic Crossword Clues G. W. Smith, and J. B. H. du Boulay - 1986

Crossword Compiler-Compilation H. Berghel and C. Yi. - 1989

PROVERB: The Probabilistic Cruciverbalist Greg A. Keim, Noam M. Shazeer, Michael L. Littman - 1999

Computer Assisted Analysis of Cryptic Crosswords P.W.Williams and D. Woodhead - 1977

LACROSS language, formal definitions - good building material Cryptic crossword clue interpreter M Hart, RH Davis - 1992

Microcomputer compilation and solution of crosswords RH Davis and E J Juvshol - 1985

Give Us A Clue Jon G. Hall and Lucia Rapanotti - 2010

A Statistical Study of Failures In Solving Crossword Puzzles Naranana, 2010

Expertise in cryptic crossword performance Kathryn Friedlander, Philip Fine, 2009

Cattell, R. G. G. "Formalization and Automatic Derivation of Code Generators". PhD thesis, 1978. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA

Part VIII

References

Lewis, Forbes D.

Recursive Descent Parsing

http://www.cs.engr.uky.edu/~lewis/essays/compilers/rec-des.html

Frost, Richard; Launchbury, John (1989). "Constructing natural language interpreters in a lazy functional language". The Computer Journal. Special edition on Lazy Functional Programming 32 (2): 108–121. doi:10.1093/comjnl/32.2.108.

Cryptic crossword clues: generating text with a hidden meaning David Hardcastle - 2007

The Generation of Cryptic Crossword Clues G. W. Smith, and J. B. H. du Boulay - 1986

Crossword Compiler-Compilation H. Berghel and C. Yi. - 1989

PROVERB: The Probabilistic Cruciverbalist Greg A. Keim, Noam M. Shazeer, Michael L. Littman - 1999

Computer Assisted Analysis of Cryptic Crosswords P.W.Williams and D. Woodhead - 1977

LACROSS language, formal definitions - good building material Cryptic crossword clue interpreter M Hart, RH Davis - 1992

Microcomputer compilation and solution of crosswords RH Davis and E J Juvshol - 1985

Give Us A Clue Jon G. Hall and Lucia Rapanotti - 2010

A Statistical Study of Failures In Solving Crossword Puzzles Naranana, 2010

Expertise in cryptic crossword performance Kathryn Friedlander, Philip Fine, 2009

Cattell, R. G. G. "Formalization and Automatic Derivation of Code Generators". PhD thesis, 1978. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA