9 Analysis of Single clue solving against test suite

9.1 Test Suite

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

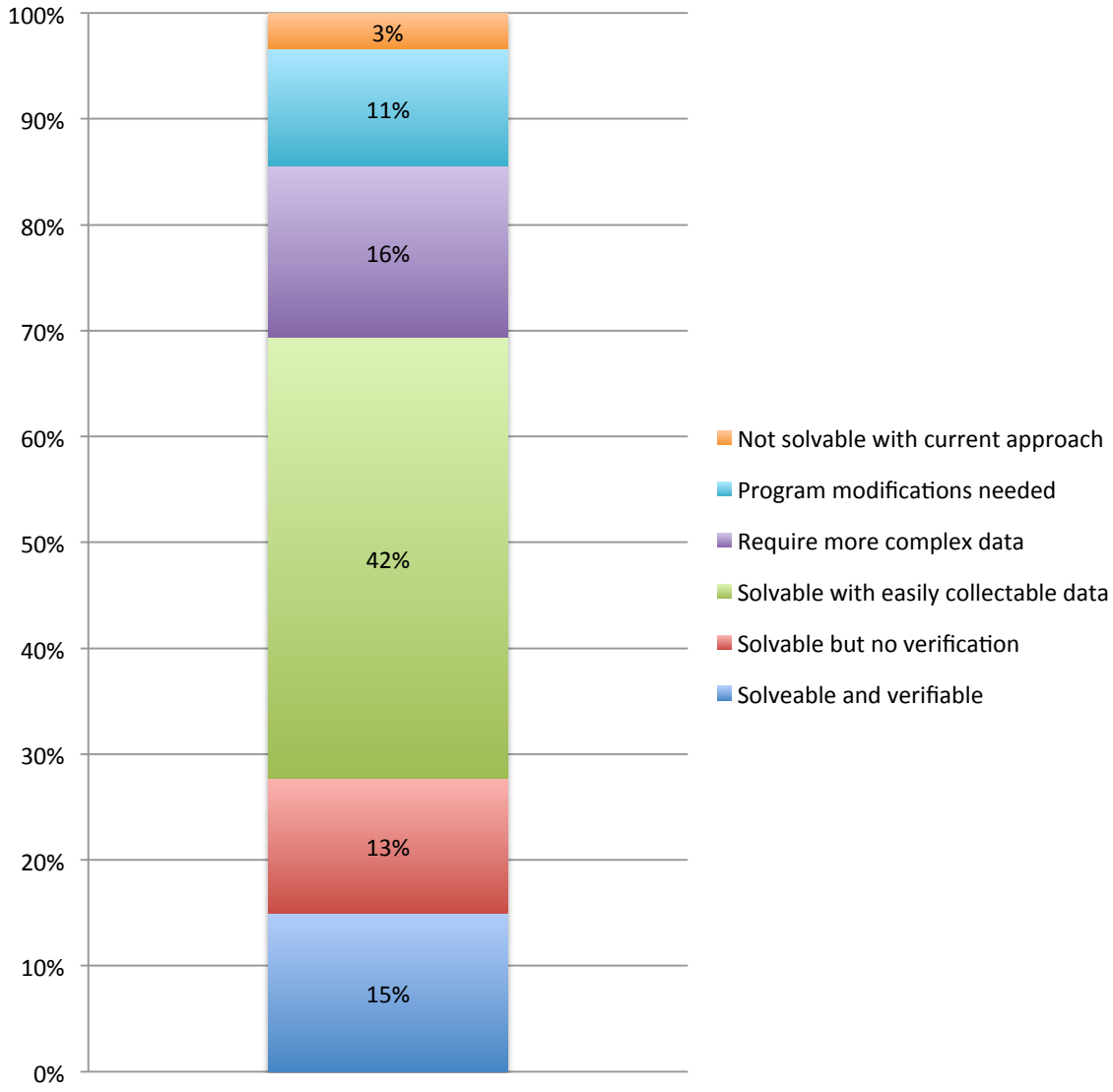


Figure 10: Breakdown of solvability of clues from the testing suite

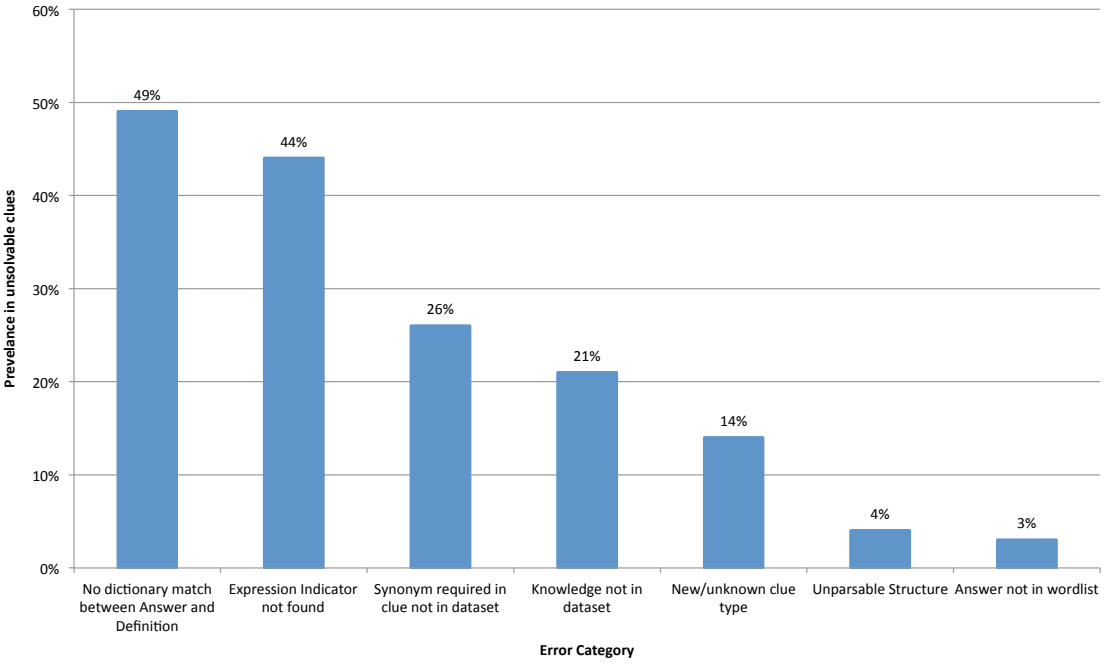


Figure 11: Reasons for unsolvable clues

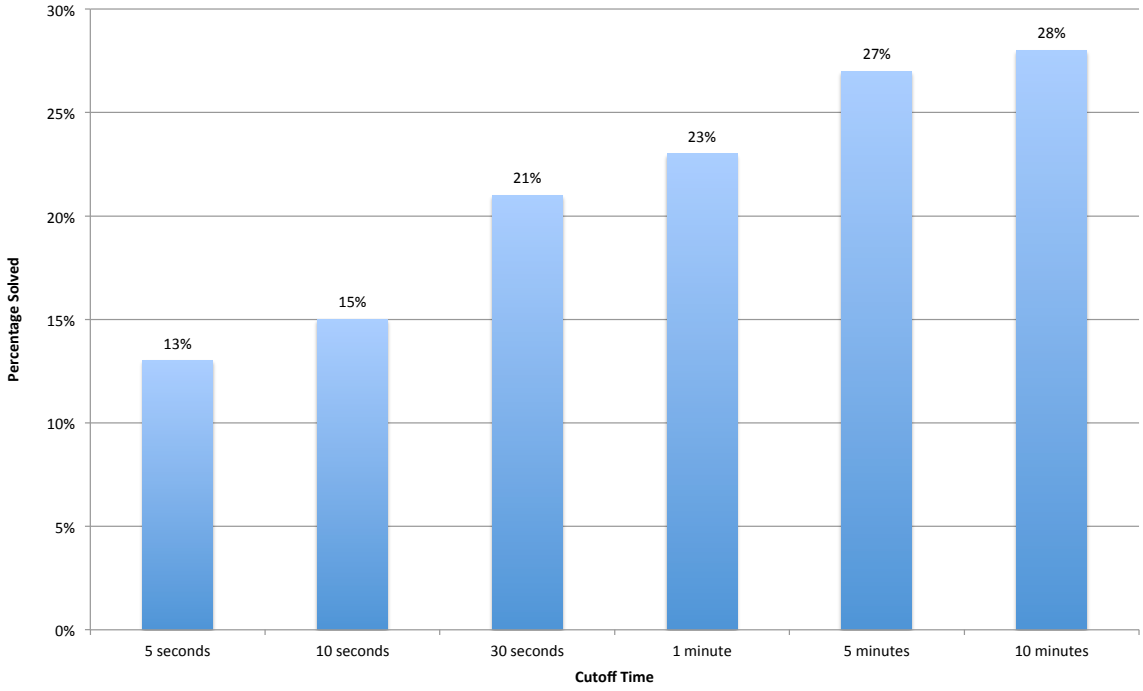


Figure 12: Percentage of clues solvable before cutoff

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.

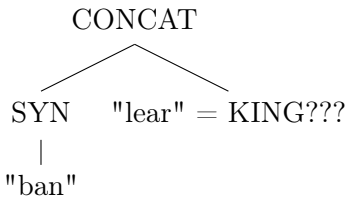
```
composer(malher).
composer(brahms).
[...]
scored(X) :- conductor(X).
scored(pele).
[...]
```

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.

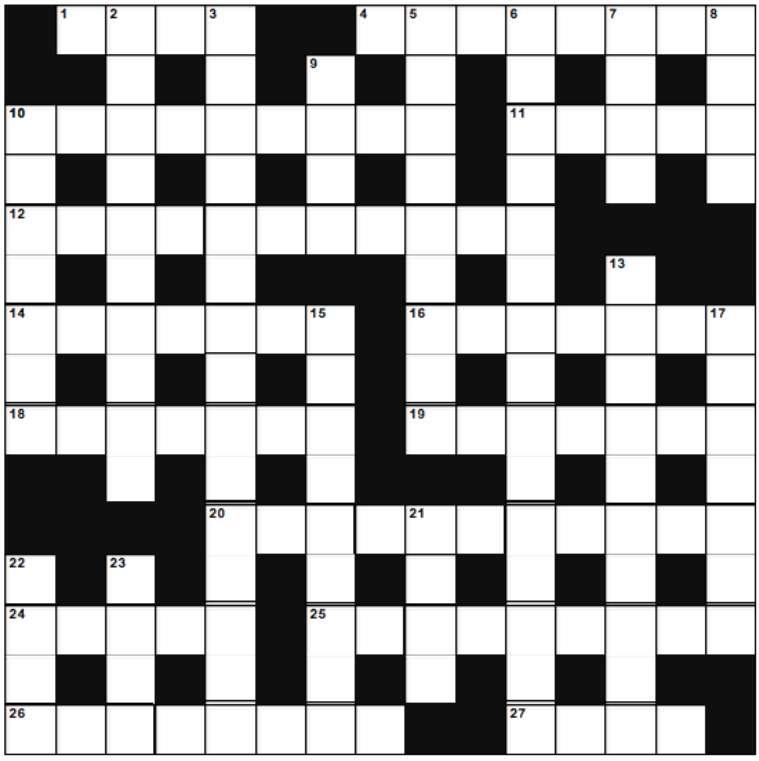


Figure 14: A crossword grid

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