Birkbeck College, University of London

Department of Computer Science and Information Systems

MSc Computer Science

Project Report

Solving Logic Problems through the Use of

Specialised Language Parsers

Abigail James

This project is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service.

2013

Contents

[Abstract 4](#_Toc367054927)

[1. Introduction 5](#_Toc367054928)

[2. Background to the Specialist Task 6](#_Toc367054929)

[2.1. Task Suitability 6](#_Toc367054930)

[2.2. Logic Problems 6](#_Toc367054931)

[2.3. Human Solving Methods 7](#_Toc367054932)

[2.3.1. Example Problem 7](#_Toc367054933)

[2.3.2. Using the Grid 8](#_Toc367054934)

[2.4. Linguistic Element 8](#_Toc367054935)

[3. Project Preparation 9](#_Toc367054936)

[3.1. Program Requirements 9](#_Toc367054937)

[3.2. Project Development Plan 9](#_Toc367054938)

[4. Design Considerations 10](#_Toc367054939)

[4.1. Development Environment 10](#_Toc367054940)

[4.2. Puzzle Input Format 10](#_Toc367054941)

[4.3. Program Structure 11](#_Toc367054942)

[4.4. Relation Representation 11](#_Toc367054943)

[5. Implementation 12](#_Toc367054944)

[5.1. Stage One – Logix 12](#_Toc367054945)

[5.1.1. Classes 12](#_Toc367054946)

[5.1.2. The Solving Algorithm 15](#_Toc367054947)

[5.2. Stage Two – Parser 17](#_Toc367054948)

[5.2.1. Structure 17](#_Toc367054949)

[5.2.2. Tagger 18](#_Toc367054950)

[5.2.3. Translator 18](#_Toc367054951)

[5.3. Stage 3a – User Interface 19](#_Toc367054952)

[5.4. Stage 3b – First-order Logic 19](#_Toc367054953)

[6. Testing 20](#_Toc367054954)

[6.1. Logix Testing 20](#_Toc367054955)

[6.2. Parser Testing 21](#_Toc367054956)

[6.3. User Interface Testing 21](#_Toc367054957)

[7. Evaluation 22](#_Toc367054958)

[7.1. Completion of System Requirements 22](#_Toc367054959)

[7.1.1. The program can read in a problem provided in XML format 22](#_Toc367054960)

[7.1.2. The program can produce a category-item dictionary based on the provided categories within the problem 22](#_Toc367054961)

[7.1.3. The program can identify the solution of simple problems provided without human assistance 22](#_Toc367054962)

[7.1.4. The program can time how long it takes to solve a problem 22](#_Toc367054963)

[7.1.5. The program can cope with problems involving more complex hints, including comparative statements 22](#_Toc367054964)

[7.1.6. The program maintains a knowledge base of terms useful in solving comparative hints, such as days of the week 23](#_Toc367054965)

[7.1.7. Extension - The program can cope with a high level of clue complexity, including gendered hints. 23](#_Toc367054966)

[7.1.8. The program can solve problems with more categories without an exponential increase in solving time 24](#_Toc367054967)

[7.1.9. The program can learn about problem-specific semantic information from the user and apply this in finding a solution 25](#_Toc367054968)

[7.1.10. The program can create FOL statements from translated clues for use in an automated theorem prover with the solution to verify success. 25](#_Toc367054969)

[7.2. Overall Evaluation of Capabilities 25](#_Toc367054970)

[8. Summary 26](#_Toc367054971)

[9. Bibliography 27](#_Toc367054972)

# Abstract

This project has been an investigation into the use of specialised parsers to improve the success rate of specialist tasks by handling semantic ambiguities that general parsers would stumble over. Logic problems with natural language clues were chosen as the studied specialist task as current solutions have automated their solution from pre-translated clues, but not with perfect success when handling the prose clues themselves. This project produced the program *Clever Zebra*, which first translates the clues to input rules that it then uses to solve the problems. It is already capable of solving several provided problems and, with further improvements to the sophistication of the parser component, would solve more still. Its success rate depends upon the functioning of the specialist parser as mistranslation or the overlooking of one piece of information renders the problem unsolvable.

Supervisor: Michael Zakharyaschev

# Introduction

*Clever Zebra* is a program that is capable of solving a standard form of logic problems by parsing the natural language clues, translating the information to a chosen representational format for logical rules and then using those rules to find the solution to the problem. It was developed as an investigation into the feasibility of creating a specialist parser to improve the success rate of specialist tasks as a general parser may stumble over translating information that is not in fact relevant to the continuation of that task.

The program is currently able to recognise seventeen word patterns typically found in the natural language style of the logic problems and for each can produce the corresponding relation(s) required to then put the inherent information into context for the solution of the problem. There remain some situations still to be completed that would improve the coverage of problems solved, including in the grammatical translation of clues, basic morphology from a word as it appears in the category listing to the prosaic clues and some user-assisted semantic interpretation to place an item into a subset of the category (e.g. a male or female name).

# Background to the Specialist Task

## Task Suitability

The topic of this project was chosen whilst considering the current ability of software to solve a form of logic problem the author is very familiar with, commonly known as Grid Logic Problems. Existing software can be found that will solve these problems, or ones of a similar format, however some require a human user to translate the natural language clues, and one other could translate the clues, but through use of a general parser found situations in which the full meaning of the clue could not sufficiently be translated as to allow the logical unit to solve the problem. Thus it was decided to investigate the possibility of developing a specialist parser that could more readily handle the particular subject problems.

## Logic Problems

The logic problems considered evolved from an older format, popular in the fifties after publication of the most famous example, known as the Einstein Puzzle, included here. (It is from this puzzle that the software gained inspiration for its name.)

1. There are five houses.

2. The Englishman lives in the red house.

3. The Spaniard owns the dog.

4. Coffee is drunk in the green house.

5. The Ukrainian drinks tea.

6. The green house is immediately to the right of the ivory house.

7. The Old Gold smoker owns snails.

8. Kools are smoked in the yellow house.

9. Milk is drunk in the middle house.

10. The Norwegian lives in the first house.

11. The man who smokes Chesterfields lives in the house next to the man with the fox.

12. Kools are smoked in the house next to the house where the horse is kept.

13. The Lucky Strike smoker drinks orange juice.

14. The Japanese smokes Parliaments.

15. The Norwegian lives next to the blue house.

Now, who drinks water? Who owns the zebra?

At the core of the puzzle is a collection of categories, each containing a certain number of named items. Typically there is a named individual, or in the above case a person of a particular nationality. Each individual is related to just one item in each other category, as partially revealed by the clues. For example, the Ukrainian is known to drink tea. The idea of the puzzle is, by process of elimination and perhaps some reduction to absurdity, to identify who relates to the items in the final question, which won’t necessarily have been explicitly mentioned, but should be inferable by considering all clues.

The modern, popular format that this project uses for the ready supply of different puzzles, is often referred to as a “grid logic problem” due to the solving aid that is provided alongside the clues to assist the user in identifying all the relations within the puzzle. Because of this grid, all of the category items are specified beforehand, so there is no need to be able to semantically group Norwegian, Frenchman, etc. into the category “Nationalities”. The linguistic style of the clues is also somewhat standardised, so creating a specialist parser is not as complex a task as creating a general-purpose language parser, and therefore a good mix of parsing/solving could take place within the confines of this project.

Once the natural language clues have been parsed, the relations can be used to match every item from the first category to an item in every other category, so this type of logic problem could be considered more complete than an Einstein puzzle as it requires a full investigation not limited to just one or two of the provided categories.

## Human Solving Methods

The human user will consider each clue of the puzzle in turn, normally, and mark the grid with any item matches or mismatches that the clue reveals. An example problem is now considered.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Brown | Green | White | 7 | 9 | 10 |
| Anne |  |  |  |  |  |  |
| Brian |  |  |  |  |  |  |
| Mary |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |

### Example Problem[[1]](#footnote-1)

Three children live in the same street. From the two clues given below, can you discover each child’s full name and age?

1. Miss Brown is three years older than Mary.
2. The child whose surname is White is 9   
   years old.

Solution:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Brown | Green | White | 7 | 9 | 10 |
| Anne | y | x | x | x | x | y |
| Brian | x | **🗶** | **✓** | x | **✓** | x |
| Mary | x |  | **🗶** | y | x | x |
| 7 | x |  | **🗶** |  |  |  |
| 9 | x | **🗶** | **✓** |  |  |  |
| 10 | y | x | x |  |  |  |

Miss Brown (clue 1) cannot be Brian, so a cross is entered in the Brian/Brown box. She cannot be Mary, so a cross is placed in the Mary/Brown box. This means she is therefore Anne, so a tick is placed in the Brown/Anne box and further cross marks placed in the other potential surname boxes in Anne’s row.

Anne Brown is three years older than Mary (clue 1). She must be 10 and Mary 7. Ticks are placed to represent this in the forename/age area of the grid and corresponding crosses against mismatching values. In the surname/age of the grid, a tick and corresponding crosses can mark the Brown/10 pair.

The grid then reveals that Brian must be 9, so a tick can be entered.

Clue 2 states that White is 9, so that must be Brian. Ticks are entered in the White/9 and White/Brian boxes and for each, corresponding crosses.

This leaves Green as the only remaining possible surname for Mary. Filling in a tick here completes the grid.

### Using the Grid

For a problem of the size above, the grid might not seem necessary once the formation of the puzzle is understood. With just three items per category and only three categories, it can potentially be solved mentally with a small amount of concentration, however these problems can expand both in terms of size and complexity. The most readily found problem consists of four categories with five items in each, with larger ones provided occasionally for those looking for a problem that will require a longer length of time to solve. The increased size means that more hints are needed, but also that more time must be spent considering other relations found by investigating the grid, rather than the original clue. For example in this partially completed grid we have deduced that A1 cannot be D4 or D5, noted by the crosses in the boxes for A1/D4 and A1/D5. It can also be seen that A1 has been shown to be connected to C4.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | B1 | B2 | B3 | B4 | B5 | C1 | C2 | C3 | C4 | C5 | D1 | D2 | D3 | D4 | D5 |
| A1 |  |  | 🗶 |  |  | 🗶 | 🗶 | 🗶 | ✓ | 🗶 |  |  |  | 🗶 | 🗶 |
| A2 | 🗶 | 🗶 | ✓ | 🗶 | 🗶 |  |  |  | 🗶 | 🗶 |  |  |  | 🗶 |  |
| A3 |  |  | 🗶 |  |  |  |  |  | 🗶 |  | 🗶 | 🗶 | 🗶 | ✓ | 🗶 |
| A4 |  |  | 🗶 |  |  |  |  |  | 🗶 |  |  |  |  | 🗶 |  |
| A5 |  |  | 🗶 |  |  |  |  | 🗶 | 🗶 |  |  |  |  | 🗶 |  |
| D1 | 🗶 |  |  |  |  | ✓ | 🗶 | 🗶 | 🗶 | 🗶 |  |  |  |  |  |
| D2 | 🗶 |  |  |  |  | 🗶 | 🗶 | 🗶 | 🗶 | ✓ |  |  |  |  |  |
| D3 | 🗶 |  |  |  |  | 🗶 |  |  |  | 🗶 |  |  |  |  |  |
| D4 | ✓ | 🗶 | 🗶 | 🗶 | 🗶 | 🗶 |  |  |  | 🗶 |  |  |  |  |  |
| D5 | 🗶 |  |  | 🗶 |  | 🗶 | 🗶 |  |  | 🗶 |  |  |  |  |  |
| C1 |  | 🗶 |  | 🗶 | 🗶 |  |  |  |  |  |  |  |  |  |  |
| C2 |  | 🗶 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| C3 |  | 🗶 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| C4 | 🗶 | ✓ | 🗶 | 🗶 | 🗶 |  |  |  |  |  |  |  |  |  |  |
| C5 |  | 🗶 |  |  |  |  |  |  |  |  |  |  |  |  |  |

By understanding that all items related to A1 are now related to C4, we can complete C4/D4 and C4/D5 boxes, and in fact deduce that A1 and C4 must be connected to D3. This in turn leaves the only possible C connection for D5 as C3. By continuing this process the whole of the above grid can in fact be completed.

## Linguistic Element

The linguistic complexity of the clues provided can vary. For example, clue 2 in the example problem illustrates a simple, direct relation, whereas clue 1 provides a comparative relation between two items. A full set of clues will normally include a mixture of these two, or could perhaps provide a rule with a subset of items within which one individual’s item can be found (“Samantha’s surgeon was a woman”). This last clue type demonstrates the purpose of one of the extension tasks of this project – to provide basic semantic interpretation of a few more commonly-occurring scenarios. Puzzles do occasionally rely on the reader to be able to identify a name as either male or female and with the completion of an extension task of the project, upon discovering the mention of this subset identifier, it would be possible for the software to request the user gender-categorizes available names if leading/following pronouns could not automate this.

# Project Preparation

## Program Requirements

The central aims of the project were to use a test-driven development approach to produce a modularised application capable of solving logic problems from natural language clues. The specialised parser would form one of the modules, and the logical deduction of the solution would take place in a separate module. The itemised deliverables for this project, as stated in the proposal, are as follows:

* The program can read in a problem provided in XML format
* The program can produce a category-item dictionary based on the provided categories within the problem
* The program can identify the solution of simple problems provided without human assistance
* The program can time how long it takes to solve a problem
* The program can cope with problems involving more complex hints, including comparative statements
* The program maintains a knowledge base of terms useful in solving comparative hints, such as days of the week

Additionally, if time allowed, a few extension items were identified:

* The program can cope with a high level of clue complexity, including gendered hints. (Basic semantic interpretation.)
* The program can solve problems with more categories without an exponential increase in solving time
* The program can learn about problem-specific semantic information from the user and apply this in finding a solution
* The program can create first-order logic statements from translated clues for use in an automated theorem prover with the solution to verify success.

## Project Development Plan

The project was broken into three main stages: building the module capable of solving problems from pre-translated clues, building the module capable of translating the clues, and finally creating a user interface and investigating the use of a first-order theorem prover to verify the solution uncovered by the logical module. A phased software development methodology was therefore chosen as the most appropriate approach to take in developing the program. A test-driven approach would also be very important in verifying the success of the logical and parsing modules, and so would control the development undertaken within each stage of work. The planned schedule follows.

|  |  |
| --- | --- |
| Dates | Targets |
| Apr 9 - May 9 | Build up the bank of logic problems and their solutions. Create method to import problems. |
| May 9 – Jun 3 | Build initial dictionaries of conjunctions and comparative terms by studying puzzle clues. |
| Jun 3 – Jun 17 | Create training statements and first methods in solving algorithm (positive/negative associations). |
| Jun 17 – Jun 24 | Create methods for process of elimination and absurdity reduction checkers. Create user-friendly output of solution progress. |
| Jun 24 – Jul 1 | Create “problem solved” detection method and containing class to solve problem from training statements. Include timing of solving methods. |
| Jul 1 – Jul 15 | Create method of transforming real puzzle hints into bare associations through use of dictionaries. |
| Jul 15 – Aug 5 | Improve sophistication of text interpretation module. Add GUI elements to show solution progression. |
| Aug 5 – Aug 19 | Transform clues to FOL statements for FO prover. |
| Aug 19 – Aug 31 | Other Extension Tasks |

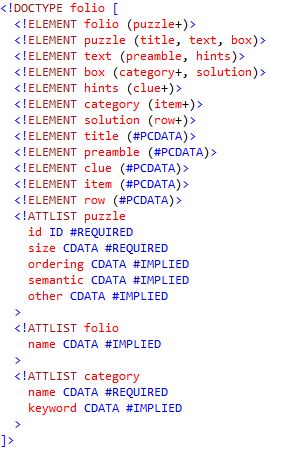
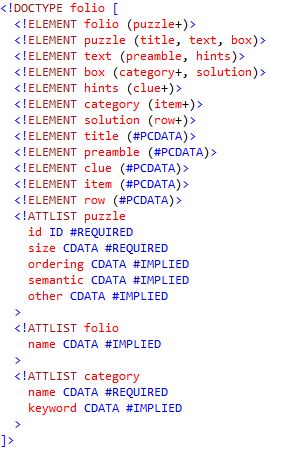
# Design Considerations

## Development Environment

The software was developed in Visual Studio 2012, with code in C#. This was chosen due to familiarity with the IDE and language, whilst also retaining the option to write modules (the Logix module in particular) in C++, if so desired. The chosen third-party first-order prover was also developed in C++ and therefore should be includable as a project within the whole solution from the source code. The user interface would be a simple collection of WinForms, as the presentation was not the main focus of the project.

## Puzzle Input Format

For the input format of the logic problems, XML representation was chosen to allow a whole collection of problems to easily be imported by the software. The puzzle source was defined by the below DTD.



The XML document containing the puzzles, once transcribed, was tested against the DTD using an online XML validator. This helped to identify a few transcription errors that may have caused difficulties.

The names for individual elements of the XML document were chosen based upon the visual representation of the problems in the original sources, which were two editions of the Logic Problems magazine. The *ordering*, *semantic* and *other* attributes were added during transcription as a way of gauging the complexity of the problem. They are not used by the program, just the author when deciding which puzzle to use for each test.

## Program Structure

The logical and parsing capabilities of the program were modularised into individual units so that one could be used without the other in any extension activities. A main, controlling module would call each unit as required to solve a puzzle. The particular forms used to represent the puzzle and relational information was contained in another module, which all three other units refer to. Thus the dependency diagram can be expressed rather simply:

CleverZebra

Logix

Parser

Representation

## Relation Representation

A rather simplistic method of representing relations was used, similar but not identical to that suggested in the project proposal. If the first item in the first category was identified as being related to the first item in the second category, for example, the resulting relation would be “A1=B1”. If they were identified as not being related “A1!=B1” would represent this. A comparative relation, i.e. one category item’s corresponding item in another category is expressed as being more/less than another item’s, would be represented as “A1(B)>C1(B)”. If the exact difference between the two items is known, this would be shown in the form “A1(B)-C1(B)=2”. Later, when considering the possibility of semantic categorisation, the rule format “A1(B)={male}” was added. A representation for conditional statements was also chosen, even though the chosen puzzle source does not feature these types of clues, for the sake of completeness. An example could be “?A1=B1?A1(C)>B3(C):A1(C)>B1(C)”, arising from “Mr Jones bought a tree the day after the other tree was bought” where there are two items in category B that are types of trees.

# Implementation

## Stage One – Logix

After creating the first project within the Visual Studio solution, CleverZebra, the project for the logical module was created and the name Logix chosen. An additional project was created to house the tests, LogixTests. The first test was written in C# and then development began within Logix to create classes to allow the test to pass. The first burst was written in C#, though timings were noted at this early stage. If they seemed disappointingly slow, then development would have switched to C++, however results were passing with promisingly good times, probably thanks to the relatively new .NET 4.5 framework, so work continued in C#.

### Classes

The first development work focussed on the construction of the class that would hold all information relating to one category. It would have an array for positive matches, an array for negative matches and an array for the values of its items. As the whole class revolved around the need to store the information, the class was originally called Line, after a line of the grid solving aid, but as the code became more substantial and the grid less important, this was renamed to the **Category** class.

Category example {

size = 3,

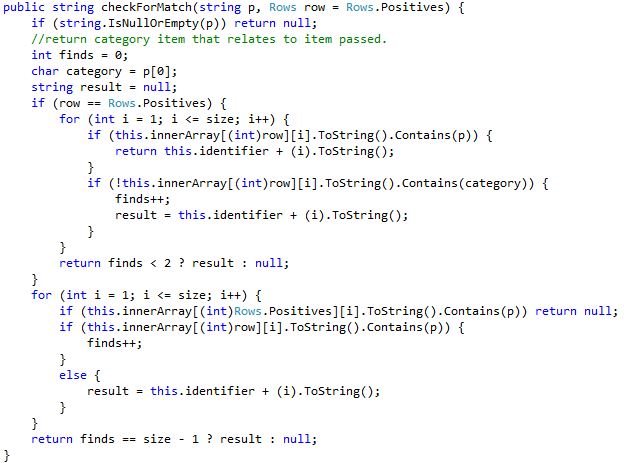
identifier = ‘C’,

keyword = “days”,

innerArray = { {0,1,2,3}, {‘A’, “Monday”, “Wednesday”, “Friday”}, {“+”, “”, “”, “”}, {“-”, “A1”, “D2,B1”, “A3”} }

}

The purpose of the first three tests to be written was to check the creation of an object of this class and two of its basic methods. The Category class would need to receive a whole relation and be able to return whether it used this relation and also whether information held within the class object could provide any further relations, given the one considered. The name of the second test hints at this purpose – Test\_AllButOneFound. This simulates a Category ‘B’ object for a puzzle four items deep and uses the “addRelation” method to mark a list of pairs – A1=B1, A2=B2,A3=B3,A4=B4. It then calls “checkForMatch(“B5”)” and this returns the string “A5”. This method is considering, for a given item, whether enough information for that item’s category has been recorded to match the item to its associated item with the B category. (If the item has already been listed in the positive row of the innerArray, that index is returned with the category identifier to provide its already-known match.) The method also considers whether the item to be checked has been listed as a non-match against every item bar one, thus revealing the only possible item remaining. The result of the checkForMatch function is checked after it is called and a new relation formed if a matched item is found.



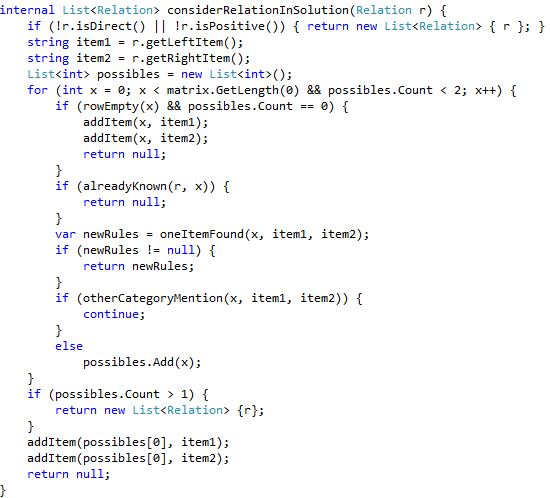
The relations themselves were originally held in one class, with the intention of analysing the held string when needing to interact with other classes depending on whether the relation was direct, comparative, etc. However this was soon separated into a collection of **Relation** classes inheriting from the base Relation class. By project end, the relation types created were DirectRelation, RelativeRelation (for comparative relations), ConditionalRelation and SemanticRelation. Each Relation type held a particular number of items and overridden functions such as getBaseItem() to return the correct item required by context, given its sort. A **RelationFactory** class, following the Singleton pattern, was created to instantiate the correct type of relation based upon the given string. The RelationFactory can also build a DirectRelation from the individual parts, as this is often required by methods generating newly-discovered relations. For the purpose of maintaining the level of abstraction necessary to allow the modules to function independently, the RelationFactory references the external Representations module when deciding what sort of Relation to create.

Another method created in the early tests is the considerRelationToCategory method, which would be one of the most crucial functions within the Logix module. Similar to the checkForMatch, method, but operating on the level of a whole relation, rather than an individual item, considerRelationToCategory checks whether the information held within that relation should be stored in the subject category and whether it reveals any new relations in conjunction with information stored within the category. The method contains a pathway for each type of relation and returns either a collection of relations generated by considering the input relation, just the input relation itself, if nothing can be discovered within this category, or an empty collection if the category has used the input relation in the only place it should be used (i.e. for a direct relation featuring the subject category). The function is included in the full listing of the Category class as Appendix B4.

To handle comparative relations, a family of classes derived from a base **Calculator** class were added. These classes can put the relative difference between two items into context of the actual values available and instantly create negative relations to any items which this rule alone shows are impossible partners for the two items mentioned. Once an item actually has an assigned value within that category due to other rules, the calculator then determines which item the second must be related to if the relation is quantified (A1(B)-C2(B)=5) or to any values which are higher/lower than the second lower/higher item can be.

After these first few tests of the Category class, the class that actually holds the main solving algorithm was created: the **Deducer** class. A Deducer object is instantiated based on a given puzzle and contains an assortment of functions to control the process of finding the solution to that puzzle and also to fire events that can update the UI with progress.

A **Solution** class was created to record the positive matches found that would be mirrored in the UI and also to cover the situation where a pair of relations can only be related to other categories by considering existing relation pairs. When the Solution class receives a positive DirectRelation (the only sort of Relation it considers), it tries to fit this pair of related items into the table of matches if it can.



The matrix is the Solution’s internal representation of the answer grid with one row per item in each category. The method considers each row of the matrix in turn, looking to fit the two matched items into the grid, if possible. So if the relation “A1=C3” was to be considered, then depending on the state of the Solution, different resulting relations might be returned.

If the row is empty and no possible rows have yet been encountered, e.g. if this is the first row, the two related items in the relation begin considered are added to this row.

|  |  |  |  |
| --- | --- | --- | --- |
| A1 |  | C3 |  |
|  |  |  |  |
|  |  |  |  |

Otherwise if both items are already contained within that row, we exit the function, returning no new relations. Otherwise if one of the items is found in that row, then the second is added and positive relations between it and any other items in that row are created.

|  |  |  |  |
| --- | --- | --- | --- |
| A1 | B4 | C3 |  |
|  |  |  |  |
|  |  |  |  |

If neither item is found in that row, but a different item of the same category as either is populated then the rule cannot be placed in that row. If no other item of either category is mentioned, then that row is considered a possible fit for the input rule. Once all rows have been considered, if the process has not already returned with new relations and it has found only one possible row for the input rule, it now places the items into that row and creates any new relations revealed (e.g. “A1=B4”).

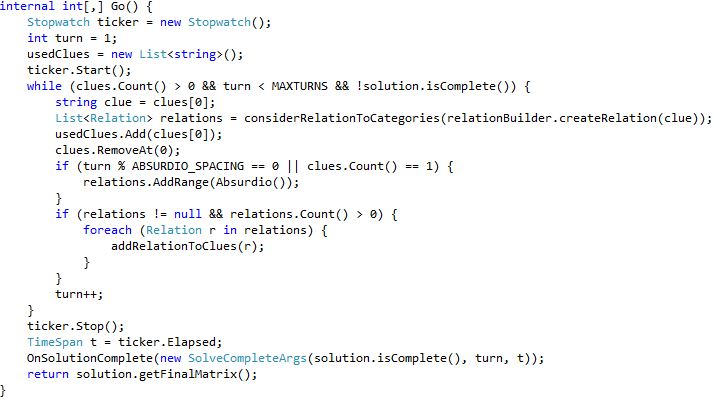
|  |  |  |  |
| --- | --- | --- | --- |
| A2 | B3 |  |  |
| A1 | B4 | C3 | D1 |
| A3 |  | C2 |  |

With each item added to the matrix, the Solution also fires an event so that the front-end can be updated with the new item. The solution becoming completely filled is one way in which the whole solving process will come to a halt.

Within the Logix project there are also various utility classes that report event and exception information at various times: LogixException, MatchEventArgs, SolutionUpdateArgs, and SolveCompleteArgs. The final class diagram for Logix is included as Appendix B2.

### The Solving Algorithm

From the start, one of the aims of this stage of the project was to consider a couple of ideas for solving algorithms and determine which one had more success. The two separate ideas came from thinking about methods used by humans to solve the puzzles. You can start from the beginning of the clues, add in the basic relational information these contain and then when the clues are exhausted look for information highlighted by the grid to uncover new relations and effectively then mark those until the whole grid is filled in. The other tactic is to look for these additional relationships each time a new relation is added to the grid. This helps to keep more relevant information in mind to spot and include new relations whilst working through the clues. For the computerised algorithm, therefore, the idea was to do an initial pass of all clues and then consider others found, or to start looking around for these additional relations between each input clue. The relevant functions for the algorithm lie within the Deducer class, with the simply-named Go() function controlling the main process.



The basic idea behind the Go() function is that each rule provided is considered against each category and relevant information is noted. Any new relations uncovered are added to the bank of clues to work through and the function loops until the solution is found. As mentioned previously, it is necessary to perform “grid checks” to look for information not given directly as clues, and this is what the Absurdio()[[2]](#footnote-2) function does; it is so named because as well as checking for deductible information from existing matches, it also performs a reduction to absurdity test for incomplete categories. Without the call to Absurdio, the solution to the problem is normally not found and without a limit on the number of iterations, the algorithm would not halt. With Absurdio present and correctly functioning, the original puzzle clues provide enough information to successfully find the solution and the MAXTURNS limit is not required for the algorithm to halt. However, if a relation has been missed through to imperfect translation of natural language to puzzle clue, then once again the algorithm would not halt. This was discovered whilst completing the first series of tests for Logix in which ready-translated clues were fed to the Deducer and the solution verified. One overlooked rule resulted in the Go function not terminating and the MAXTURNS limit was introduced.

To test the alternate algorithm ideas, the Absurdio\_Spacing was not originally present and a check for additional relations was made every turn. Once the algorithm had been used to successfully pass the first test, however, optimisation began, which included deciding at what point the Absurdio function should be used. Absurdio being called at every pass resulted in the first test problem, containing four categories each containing three items, being solved in 52 turns. As the whole problem at this size only contains 54 possible pairings ((3\*3\*3)+(2\*3\*3)+(1\*3\*3)), this seemed quite poor. Following the original plan for the second algorithm, based on human solving methods, the Absurdio\_Spacing value was set to 5 – the average number of clues in these puzzles, meaning it would first be considered once all main clues had been placed. This immediately dropped the number of turns required to 37. Experimenting with other values resulted in a nonlinear relation: increasing the spacing improved results up to a point, but reached a maximum value at around 8, which was therefore chosen as the value to use as development continued. A second round of optimisation of this figure occurred later in the project and is discussed in the evaluation section.

Timing of the tests had shown that the problems could be solved in time intervals so small, that the real comparison was in the number of turns, and not timing. Originally every relation was compared to the individual categories and then separately to the solution that was being developed. When looking for ways to optimise the Deducer’s progress, however, an investigation into the use of events to simultaneously update the solution as a positive item is added to or created by a category turned out to be very fruitful. The third test puzzle was the first of size 4x4 to be considered and it was being solved in a disappointing 106 turns and ~80ms. Adding in an event trigger for the solution instantly reduced this to 55 turns and a solve time of 8ms. Further performance figures are included in the evaluation section.

## Stage Two – Parser

Whereas for Logix the inspiration for code design came from knowledge of manual processing methods, the influence for Parser’s methods came from reading about the subject of natural language processing[[3]](#footnote-3). The most important concepts learnt were the use of “tagging” to categorise each word in a sentence as a particular part of speech, dictionaries to allow the tagging to take place and a buffer to match to grammatical rules based on the sequence of individual tags. This project did not require the sentences to be matched to rules of natural language grammar, but rather to expected patterns in the more restricted grammar of puzzle clues. Not every word would be tagged, just those relevant to the problem type generally and then additionally those that are relevant to the particular problem. This meant there would be one largely static dictionary and another created per puzzle.

### Structure

Mirroring the structure of the whole program, the **Parser** class was developed to call the **Tagger** and **Translator** classes in turn. The Parser is instantiated with the puzzle, of which it uses parts to then in turn instantiate the Tagger and Translator. The Tagger creates a **TermsDictionary** which contains words required by every puzzle and which considers keywords within the puzzle to include additional words that might also be necessary. For example, if a category carries the keyword “currency”, the TermsDictionary will add words such as “money”, “cash”, “cheap” and “expensive”. Again, it is the more simple and repeated use of language within these puzzles that allows a parser with less than a full Oxford dictionary to successfully reach a solution. The specialisation of the parser for this task is shown in the limits of the word lists contained in this dictionary. Full copies of the dictionaries appear in Appendix B.

### Tagger

Alongside the TermsDictionary, the Tagger instantiates a **CategoryDictionary** from the puzzle items which splits all multi-word category items into individual words. It ignores any short words, such as “as”, “in” or “the” whilst doing this so as not to unnecessarily bloat the tag pattern. Tagger then considers each clue in turn and each word within that clue in turn. Initially, it creates a string with every tag matched for the length of that sentence. An example translation is shown in this table:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Zachary | Zeffer | watched | his | hat | fly | into | the | centre | of | the |
| A3 | A3 |  |  | B,B2 |  |  |  |  |  |  |
| dual | carriageway | and | under | the | wheels | of | the | speeding | lorries | . |
| D1 | D1 | Ta |  |  |  |  |  |  |  | . |

It then condenses this tag pattern down using the condenseToString function to prepare it for Translator. This involves considering each tag and whether it should remain in the final pattern or not. The above tag line would thus become simply “A3 D1”. This is because the tag “B,B2” without an adjacent “B2” is considered irrelevant and both the “Ta” (and) tag and punctuation tags require another tag some time after themselves before they would be kept. To achieve this, condenseToString makes use of the **PatternBuffer** class. This buffer allows adjacent tags to be considered and kept in if they make a sensible combination, removing anything irrelevant. A more complicated example would come from what is actually the second half of the above clue:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| This | happened | two | days | before | Gareth | Gale | watched |
| Tt |  | Tx(two) | Tq(days) | Tp(-) | A2 | A2 |  |
| his | flat | cap | disappear | on | the | wind | . |
|  | B3 | B1,B3 |  |  |  |  | . |

The first tag is kept as it will indicate that a relationship must be formed with the previous sentence and the item we will see in this one. The TermDictionary has noted the keyword “days” and now picks up on numbers and comparative terms (“before”) along with the specialist term “days” itself. These form a Tx,Tq,Tp[[4]](#footnote-4) pattern and are therefore kept. The remaining items are deduplicated and kept, except for the punctuation, resulting in the string “Tt Tx(two) Tq(days) Tp(-) A2 B3”.

### Translator

Tagger then calls upon Translator to turn the finished tagline into relational statements. To do this it compares the part or parts of the line to tag patterns within the **PatternBank** class. [A3 D1] is the most simple pattern – a pair of category items, however the [Tx,Tq,Tp] is just part of a longer one. Because of the “Tt” tag, the pair of sentences featured would be considered together and the PatternBuffer would perform its second purpose in translating the second half.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A3 | D1 | . | Tt | Tx(two) | Tq(days) | Tp(-) | A2 | B3 |

First, the Translator forms a relation from the left-hand side of the punctuation mark (“A3=D1”) and makes a note of the first item. It then replaces the “Tt” with this item and passes the second half to the buffer.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A3 | Tx(two) | Tq(days) | Tp(-) | A2 | B3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A3** | Tx(two) | Tq(days) | Tp(-) | A2 | B3 |
| **A3** | **Tx(two)** | Tq(days) | Tp(-) | A2 | B3 |
| **A3** | **Tx(two)** | **Tq(days)** | Tp(-) | A2 | B3 |
| **A3** | **Tx(two)** | **Tq(days)** | **Tp(-)** | A2 | B3 |
| **A3** | **Tx(two)** | **Tq(days)** | **Tp(-)** | **A2** | B3 |
| A3 | Tx(two) | Tq(days) | Tp(-) | **A2** | B3 |
| A3 | Tx(two) | Tq(days) | Tp(-) | **A2** | **B3** |

The buffer takes the first item and then keeps adding more from the string until a pattern in the PatternBank is matched. The pattern it will match in this case is “C,Tx,Tq,Tp,C”. A relation is formed based on this pattern and then the final item is kept and the process repeats until the string is consumed.

From each recognised pattern, a specific way of translating the tags to relations is known. In total, this version of Parser recognises and can create relations for seventeen different patterns.

## Stage 3a – User Interface

The user interface is, as intended, a simple collection of WinForm classes – MainMenu, Options, Puzzles and Solver. Each of these is a form with navigational buttons and relevant information. The Solver form is where the user actually sees the puzzle clues, the relations found by the software and the solution the software found, together with a feedback area showing the time taken to find the solution and the number of turns taken, if successful, or some feedback of why the puzzle was not solved, if it could not yet be solved. There was nothing overly complicated involved in setting up the interface, just the perennial fussiness of form controls to contend with. A screenshot of each form is included as Appendix C.

## Stage 3b – First-order Logic

The first-order theorem prover that was to be used was *Vampire*, a software program authored by Andrei Voronkov et al at the University of Manchester that had won CISC competitions since the late nineties. This was chosen because of its success and because it was authored in C++, allowing it to be integrated with the *CleverZebra* solution. Unfortunately when it came to be time to acquire the source code to add this third module, the project was unavailable due to the imminent release of a new version. After a brief search, no suitable alternative that could be incorporated into the project solution was found and instead development focus returned to increasing the parser’s sophistication.

This sort of logic problem centres around one particular rule:

[∀a ∃b (R(a,b) ∧ ∃b' (R(a,b')) → b=b')]

That is, each category item is uniquely related to just one item in each other category.

Additionally:

[(∃a,b,c ((R(a,b)) ∧ R(b,c)) → R(a,c)]

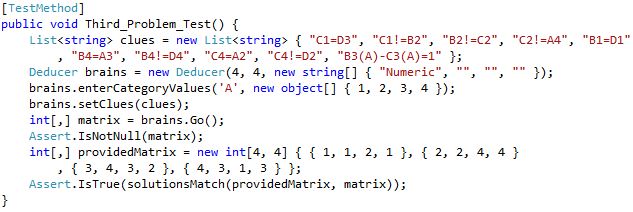
That is, for each category item that is related to another, all items that the second item relate to also relate to the first. The automated proving of the software-found solution would therefore entail providing these two basic rules along with a retranslation of the uncovered relations as the governing rules and then the translated, calculated solution as a theorem to be proved. Experience of creating the Logix module to determine the solution to the problem would suggest to the author that if a problem solution can be verified from the input rules, it will be done so fairly quickly. However if a rule has not been detected at the parse-and-translate stage, then the attempt to prove the solution would not terminate at all, as it will have insufficient information to obtain a satisfactory proof. With extra time, it would have been good to see if this observation is truly paralleled in the automated prover.

# Testing

Testing was an integral part of the coding process due to both the desire to follow a test-driven development methodology and the desire to provide a sound method of logical deduction of the problem solution. Tests were used to check the functionality of classes (such as for the Category class, mentioned above) and the performance of the individual modules. The standard Microsoft Unit Testing framework was used for all tests. A screenshot of the test results at project end is included as Appendix D.

## Logix Testing

The main test series from which the brains of the Logix module, Deducer, was built is contained in the file DeducerTest.cs. After initial tests checking the creation of a Deducer object, there are five problem tests. Each of these is given a hand-translated set of relations for a problem, such as Deducer would receive from Parser’s output, and then the solution formed is checked against the actual solution. If the problem involved keywords, these would be provided manually, as shown below.



The solutionsMatch() function simply confirms by string comparison that each line in the reached solution is contained in the provided solution and vice versa.

Following the original plan for development, these tests began with the small (3x3) puzzles and worked up in size and complexity. In this way, new requirements for the Logix module were added one by one and its complexity steadily increased. If the new test failed, debugging would identify the area(s) that needed further enhancements. This could involve the creation of a new Calculator type, or altering the flow of logic within Category’s crucial considerRelationToCategory function. After each improvement made, the full series to date could be run to confirm new changes did not affect previously successful tests. The existence of these test methods provided confidence in the robustness of the solving routine and made it possible to progress on to the Parser module in good time. (Due to external pressures, work did not begin on Logix until June 25th, however the focus of development switched to Parser on July 14th, after one week less than was originally scheduled for Logix.) A full listing of the LogixTests project is included as Appendix B5.

## Parser Testing

The Parser, from the beginning, would be making use of the puzzle source directly, so the first test before development could begin properly was actually loadPuzzles(), which checked Puzzle objects could be created from the XML document. Parser’s dictionaries were next tested using the first puzzle which had a keyword (and therefore required specialist items in TermsDictionary):

[TestMethod]

public void Check\_Dictionary\_Creation() {

Puzzle p = puzzles[0];

Parser parser = new Parser(p);

List<string> puzzle1Items = new List<string> { "brendan", "briese", "gareth"

, "gale", "zachary", "zeffer", "baseball", "cap", "bowler", "hat", "flat"

, "cap", "monday", "wednesday", "friday", "dual", "carriageway", "river"

, "tree" };

for (int i = 0; i < puzzle1Items.Count; i++) {

Assert.AreEqual(puzzle1Items[i], parser.tagger.catWords.getItems()[i]);

}

List<string> puz1Quants = new List<string> { "day", "days", "night", "nights" };

for (int i = 0; i < puz1Quants.Count; i++) {

Assert.AreEqual(puz1Quants[i], parser.tagger.terms.getQuantifiers()[i]);

}

}

The initialisation of the Tagger object used by the Parser happens as the Parser object itself is created and so this test checks that the word lists the Tagger has created match the expected items.

The tagging process was then checked in the next test before working on tagging and translating to relations. This was, again, done in stages, so for each test puzzle there would be a “Tagging” test and a “Translating” test. Similarly, after one stage was completed, all previous tests would also be run for the sake of regression testing. The tag patterns created in the “Tagging” test would be compared to the manually-created patterns and the relations from the “Translating” test would be compared to the expected relations resulting from the manual tag patterns. The tests therefore all look similar, but with varying length of patterns and relations, however each typically added one or several new patterns to be considered and accounted for in the tag/translation process.

The testing of Parser does not feel complete, as the limitations of the module have not yet been met. There is still one test that does not pass, for example, as this is for the next extension task for development. The test suite, instead of completely testing the Parser module functionality, simply tests its current capabilities. As this could still be stretched and improved upon, there are still more tests that would be written as development continues.

## User Interface Testing

There are no automated unit tests that check the user interface forms. Instead these have been put through a sort of user acceptance testing during development as they were actually used to run more puzzles than the tests alone feature. It was verified that the options settings actually have an effect when watching a problem being solved and that the navigational buttons all have the desired effect.

# Evaluation

Overall the project has achieved what it set out to do: it has resulted in a piece of software capable of using its specialist parser to solve logic problems directly from the source. There are still parts of the process to be completely automated (i.e. the identification of keywords from the clues themselves) and the sophistication is not yet at the level that would allow the software to solve all of the source problems, but the stepwise development has shown that the refinements should be possible to allow this to be so.

## Completion of System Requirements

### The program can read in a problem provided in XML format

The project makes use of the standard Microsoft System.Xml library to open and read from a transcribed document containing more than forty logic problems. This format was chosen for ease of use within the program, and although the transcription task took quite a while, the quality assurance compared to scanning magazine pages, performing optical character recognition and then reordering the resulting text as necessary was definitely worth the effort. At one point during development, it was decided that the DTD should be amended slightly and as the puzzle source was already in XML format, this was a simple enough task to do, making use of the document tree to move one child element in each <puzzle> element to a different parent element. The document tree is also useful when creating a Puzzle object as the different child elements can be referenced as required to instantiate the different object properties. The performance is unquestionably fast with the Solver form loading a puzzle without any perceptible delay.

### The program can produce a category-item dictionary based on the provided categories within the problem

Whereas one of the issues with existing logic problem solving software projects discovered whilst researching this project topic was that they required the user to supply the category items manually, *Clever Zebra* is able to collect these from the problem source and build a specific CategoryDictionary based upon them. This object ignores small words such as a, the, in, of, etc. to be left with just the relevant words that when mentioned in the clues will be assignable as that category item and not an incidental item of speech.

### The program can identify the solution of simple problems provided without human assistance

The software is perfectly capable of solving several of the provided problems and any further ones that would be provided that are of similar complexity. This includes any mixture of direct and relative relational clues, potentially mentioning differences in days of the week, dates, months of the year, currency amounts, numerical facts and more. The identification of keywords is not yet automated and the software is relying upon the keywords added at the time of transcription for now, however a short list of terms which already features in the TermsDictionary could be used as a pre-parse checklist to identify any category that should have a keyword and the correct word added. (The standard linguistic style of the puzzles is what makes this possible. Checking for key phrases such as “the day before” would be sufficient to determine whether comparative relations must be looked for throughout the clues pertaining to a specific problem.)

### The program can time how long it takes to solve a problem

The software does track the length of time taken to reach the solution to the problem and it also records the number of turns, i.e. the number of cycles through the main solving algorithm, that it took. This information is relayed back to the user in an information box as part of the interface.

### The program can cope with problems involving more complex hints, including comparative statements

The first test problem actually involved a comparative statement, and so *Clever Zebra* began interpreting these relations from the very start. Each comparative statement requires a keyword for the category so that it is known what the effect of “more” or “less” should be in context. These keywords dictate the subclass of Calculator that is instantiated with the Category object. The Calculator then uses information provided by the Parser to check for inappropriate values when considering a RelativeRelation formed from a comparative statement and so this information is used to identify the corresponding items. The use of “calculators” seemed like a natural choice, made early in the development of the Logix module, but the choice has proven very wise as the creation of just one provided a template for any others that became necessary and the implementation is very straightforward, allowing quite rapid development of additional functionality. Towards the end of the project span, as the beginnings of the semantic interpretation extension task were tasked out, it seemed something similar to a calculator for these terms as well would be best. This means that with not much more time it should have been possible to fully implement this extension task.

### The program maintains a knowledge base of terms useful in solving comparative hints, such as days of the week

The knowledge base of terms is currently held as hard-coded strings within the TermsDictionary and CategoryDictionary. These did not seem too cumbersome and so were not moved to external files, however if the extension task regarding learning from user input were begun, then a move to an external file would have been necessary to allow new items to have been retained.

### Extension - The program can cope with a high level of clue complexity, including gendered hints.

The basic framework for this task was created – the addition of a new representational form, a Relation type (SemanticRelation) and some positions within the logical flow of the deduction process were identified, but the full implementation was not completed. In particular, no addition had yet been made to Parser to allow it to recognise the necessary terms for the creation of the new relation type. However, for the first part of this requirement (“high level of clue complexity”), many different grammatical patterns have been considered and combining them all has been sufficiently complex enough as to qualify for this. For example, consider the translation of these two clues:

|  |
| --- |
| Barrister Damien Dowte, wearing the head of a mop on his head, strode… into court; the judge presiding was not Judge De Cree who wasn’t presiding in the court with barrister Garfield Grille, who wasn’t at work in court 4. |
| [Damien Dowte] = [mop]; [Damien Dowte] != [Judge De Cree]; [Judge De Cree] != [Garfield Grille]; [Garfield Grille] != [Court 4] |
| The man from Mobile, Alabama, who made his fortune from German bullion…, was not Anson Burgess and did not end up with $8 million. |
| [Mobile] = [Stole German Bullion]; [Mobile] != [Anson Burgess]; [Mobile] != [$8 million] |

The difference in the phrasing of these clues is slight, yet crucial in producing the correct relations, and this subtlety is detectable by Parser. Whereas for the first clue it must recognise a chain of relations, formed from the rightmost item in one pair becoming the leftmost item in the next, for the second clue it must recognise that the final part relates not to the immediately preceding item, but to the item before that. This is achieved by careful consideration when condensing the initial tag patterns and words that are sometimes, but not always, important; in this case the pivotal word is actually “and” in the second sentence, but this common word cannot be considered universally important. The word “and” is now tagged by the TermsDictionary, though it was not considered necessary at first, but the condenser function will only leave it in if a tag it pairs with is adjacent. In this example becase the dissociative tag “Td” is present from the nearby word “not”, the “Ta” and-tag remains. This then causes the line to be recognised by Translator as a different pattern to the dissociatives chain of the clue above, and the correct relations come about as a result.

|  |  |  |  |
| --- | --- | --- | --- |
| Barrister Damien Dowte, wearing the head of a mop on his head, strode… into court; the judge presiding was not Judge De Cree who wasn’t presiding in the court with barrister Garfield Grille, who wasn’t at work in court 4. | | | |
| C C1 C1 , To D3 , ; Td B B2 B2 Td C C2 C2 Td A A4 | | → | C1 D3 ; Td B2 Td C2 Td A4 |
| The man from Mobile, Alabama, who made his fortune from German bullion he and…, was not Anson Burgess and did not end up with $8 million. | | | |
| B3 , , C5 C5 Ta To , Td A1 A1 Ta Td D2 D2 | → | | B3 C5 Td A1 Ta Td D2 |

The tag patterns above also reveal another example of an item not always, but sometimes needed – commas. Most clues will include commas as the pun-filled text adds colour to the basic fact contained within, however sometimes they separate a sub-clause that must be recognised as such for the overarching grammatical structure of the sentence to be considered when grouping tags for relations. For example, the clue

*The slave who blamed a cloud obscuring the sundial, who was to meet his master later than the one who should have been at the baths, wasn’t Gormulus.*

contains one subject and two indirect objects; the first indirect object does not become the subject over the second indirect object, as was seen in the barrister-judge-barrister-court example. This sentence becomes the condensed tag pattern [D2 , Tp(+) B3 , Td A4] as can be seen in the ParserTest listing, as it is within the test method Check\_Sixth\_Tagging().

### Extension - The program can solve problems with more categories without an exponential increase in solving time

Solving time proved a rather poor indication of Logix’s performance, as stated earlier, and instead the number of turns taken to reach the solution was used to not the effect of an increase in puzzle size. Some stats that were recorded are detailed in the table below.

|  |  |  |
| --- | --- | --- |
| **Size (categories x items)** | **Number of Turns** | **Best time noted (ms)** |
| 4x3 | 16 | 3.16 |
| 4x3 | 25 | 3.87 |
| 4x3 | 16 | 2.75 |
| 4x5 | 61 | 6.61 |
| 4x5 | 61 | 8.21 |
| 4x5 | 57 | 9.94 |
| 4x5 | 66 | 9.96 |
| 4x5 | 72 | 7.87 |

A couple of larger puzzles (4x6 and 5x4) were available, but they were not solvable by project-end as one required the identification of subjects according to their gender and the other employed a universal constraint, a rare feature that has not yet been incorporated into CleverZebra. Whilst the Logix module will be able to find the solution from ready-translated clues without any difficulty, the current limitations of Parser cause it to continue iterating without making any progress if a maximum number of turns is not utilised. This was observed whilst debugging the process of Deducer’s Go() method to look for any deductions missed. Each time it would reach a point of considering the same one or two comparative clues without finding any new information in much less than the chosen number of maximum turns (200). The debugging process would then focus on re-reading the natural language clues and discovering what piece of information had not made it to the final list of relations that Deducer was considering.

Working to improve the number of turns taken involved considering the order in which newly-found relations were noted in either the Category or Solution objects. One of the first improvements made was to immediately note positive, direct relations as they were identified, which helped to improve solve times, as noted in the implementation section. The frequency with which the solving algorithm employed the Absurdio() function had been set at the best-performing number noted in the initial development phase, however further reflection on this throughout the length of the project resulted in the recognition that the optimum number would be related not to the number of clues, but to the number of relations created from those clues. As this was a number that would be known to the Deducer object, an experimental switch to this figure was made and the number of turns improved across the board.

It is suspected that further improvements could be made to the algorithm, given more time. Frequently, whilst debugging, a long list of negative direct relations are seen waiting to be processed, and though it is imperative that all are noted, the performance often approaches the worst-case time because none of these are considered preferentially to others, even though it could be from just one that the rest of the positive matches can be discovered. A human solver would focus the grid cross-referencing on a line that is visibly close to being completed, however Deducer’s method simply works methodically through the categories. If it could establish which category is closest to completion and target Absurdio on that first, the more crucial relations could be produced and therefore considered first, improving the algorithm's performance.

### Extension - The program can learn about problem-specific semantic information from the user and apply this in finding a solution

As mentioned earlier, the beginnings of this task are complete, but the equivalent of a Calculator class for semantic considerations and the corresponding parts required in Parser or the UI remain to be implemented. As such, this extension task has not been shown possible, but the groundings and the success of the similar Calculator classes, suggest this is eminently achievable.

### Extension - The program can create FOL statements from translated clues for use in an automated theorem prover with the solution to verify success.

There was not time enough to add this task in, regrettably. Having an automated verification of the solution reached would be beneficial to highlight the success of Logix, whereas for now a comparison must be made to the source-provided solution.

## Overall Evaluation of Capabilities

CleverZebra as a whole performs well at translating and solving simpler puzzlers. As the problems get more difficult there are more complex patterns that it can cope with already, but there are also patterns and details that it cannot yet cope with. As development ended for the scope of this project, the final test written that remained unsuccessful was for a puzzle that required gender categorisation of items in one of the featured categories. This is the most common hurdle remaining from the sample input of forty problems, and this is why it was being undertaken in preference to other tasks that also remain to be completed. The list of failure reasons for those problems that cannot yet be solved automatically also includes unimplemented calculators in the Logix module (length and time), a couple of patterns for inclusion in the TermsDictionary, a couple of patterns for consideration in Tagger’s condensing function, and an additional requirement for basic morphology of a word (e.g. to recognise “explodes” as indicative of the category item “explosive”). For additional semantic requirements beyond gender, one puzzle required the user to translate “mother’s sister” into the category item “aunt”. It might be quite an extension of the intended capabilities of *Clever Zebra* to allow it to be able to recognise this, but even with that particular problem failing, but with the other reasonable improvements complete, its success rate would be quite impressive, especially given the speed with which solutions can be reached by using a specialist parser, rather than a general-purpose one. A general-purpose parser would be slowed down by considering every word and then every grammatical part of every sentence and sub-clause, and given the need to add frivolous colour to the clues in problems of this format, that would be a lot of time spent to understand language that is not at all relevant to the task at hand.

The challenges already faced and completed in the development of this project have been similar to the items identified as work outstanding and each has so far been overcome, which lends confidence to the viability of this method. The test-driven approach has resulted in robust code that is reusable and maintainable, which has allowed the rapid development necessary to build this modular program. The framework has been kept as decoupled as possible so that the particular form of representation can be changed without affecting the success rate, and even the representation of the puzzle could be changed and with the extraction and mapping to a core puzzle interface, it could be used in place of the standard format used throughout the course of this project. This means that in fact similar logic problems could also be potentially solvable using this program too with some cross-format translation. Although beyond the scope of this project, which was an investigation into the performance of a specialist parser for one particular, common form of logic problem, this transferability of development work also highlights the potential of this approach.

# Summary

*Clever Zebra* is a modular program capable of reading in a logic problem provided in XML format, parsing natural language clues, identifying relationships featured between items belonging to different categories, translating these relationships to a standard form of representation and then finally using those relationships to find the solution to the logic problem. The different modules work together well to produce the solution to simple as well as some more complicated problems and the staged development approach allows for improvements to be made readily to increase the sophistication of the software as a whole.

# Bibliography

[1] Lev, I., B. MacCartney, C. Manning, R. Levy. 2004. Solving Logic Puzzles: From Robust Processing to Precise Semantics. *Proceedings of the Second Workshop on Text Meaning and Interpretation at ACL 2004.* pp.9-16.

[2] Milicevic, A., J. Near, R. Singh. 2009. *Puzzler: An Automated Logic Puzzle Solver* [online]. [Accessed 10 February 2013] . Available from: http://people.csail.mit.edu/jnear/puzzler.

[3] [Author unknown]. 1962. Who owns the zebra? *Life International* magazine.Issue December 17, 1962.

[4] MARCUS, M. 1980. *A theory of syntactic recognition for natural language processing.* London: MIT Press.

[5] GUTTENPLAN, S. 1997. *The Languages of Logic: An introduction to formal logic.* Oxford: Blackwell.

[6] Bull, S. ed. 2013. Logic Problems. Issue No 341. Puzzler Media.

[7] WINSTON, P.H. 1984. *Artificial Intelligence.* Reading, Mass: Addison-Wesley.

[10] Jurafsky, D., J. MartiN. 2000. Speech and Language Processing. Prentice Hall.

[11] Charniak, E., Y. Wilks ed. 1976. *Computational Semantics.* Oxford: North-Holland.

[12] WIKIPEDIA. 2013. *Automated Theorem Proving* [online]. [Accessed 5 April 2013]. Available from: http://en.wikipedia.org/wiki/Automated\_theorem\_proving.

[13] WIKIPEDIA. 2013. *Zebra Puzzle* [online]. [Accessed 10 February 2013]. Available from: http://en.wikipedia.org/wiki/Zebra\_Puzzle.

1. This is the example problem from the Logic Magazine, paraphrased where helpful. The full original is included as Appendix A1 [↑](#footnote-ref-1)
2. The full listing of the Abusrdio function is contained in Appendix B6. [↑](#footnote-ref-2)
3. In particular, Marcus’s *Theory of Syntactic Recognition* (Bibliography item 5) was most helpful and discussed the use of a buffer. [↑](#footnote-ref-3)
4. All tags from the TermDictionary begin with “T”. Here “Tq” is a quantifier tag, “Tp” a preposition tag and “Tx” a number tag. Other term tags include “Te” (either), “To” (of) and “Tw” (with). The Tx,Tq and Tp tags keep the original meaning so that the correct relations can be formed, though only the direction of the proposition is required. [↑](#footnote-ref-4)