Tableau Methods for Fuzzy Łukasiewicz Logic

Jonathan Peter Tammen

A report submitted in partial fulfillment of the course
COMP3740
The Department of Computer Science
Australian National University

May 2017
Abstract

An implementation of the tableau method in Kulacka et al. was developed in scala, using Gurobi to solve the linear programs that arise at the leaves. The implementation is powerful enough to solve complex formulae in a relatively small time frame, and offers configuration options to the user to allow control over how the tableau is explored. No significant difference was discovered between the different exploration techniques when compared over a sample of 1000 problems, however knowledge of the structure of the problem can help the user make an informed decision on which configuration would work best.
# Contents

Abstract iii

1 Background 1
  1.1 Introduction ........................................ 1
  1.2 Classical & Fuzzy Logic ............................. 1
    1.2.1 Łukasiewicz Definitions .......................... 2
    1.2.2 Tableau Methods .................................. 2
  1.3 Description Logics .................................. 3
  1.4 Approach in Kulacka et al. ......................... 3

2 Implementation Details 5
  2.1 Goals .............................................. 5
  2.2 Tools and Technologies ............................. 5
  2.3 Problem Definition ................................ 6
    2.3.1 Right Existentials .............................. 6

3 Experiments and Results 9
  3.1 Experimental Design ................................ 9
  3.2 Results ............................................ 9
  3.3 Discussion ......................................... 10
    3.3.1 Batch Test ..................................... 10
    3.3.2 Single Problem Test ............................ 10
    3.3.3 Future Directions ............................... 11

4 Conclusion 13

Bibliography 15
Chapter 1

Background

1.1 Introduction

Current methods for checking the satisfiability of fuzzy decision logic problems involve formatting the problems as mixed integer linear programs. This is an inefficient representation of the problem and can be improved by applying operations to the formulae themselves. The purpose of this project was to implement a tableau method for checking these problems and provide a tool that can operate on any given logical expression. Once an implementation was developed, different parameters of the tableau expansion were to be compared and analysed.

1.2 Classical & Fuzzy Logic

Formal logics provide a precise way to reason about the world. Classical logic deals with propositions that are either true or false. There are well established rules for interacting with statements expressed in classical logic, as well as years of research into the best approaches for solving such decision making problems [Ben-Eliyahu and Dechter 1996]. First order logic works with compound predicates made up of atomic predicates, connectives and quantifiers. Negation ($\neg$) is the primary unary operator for classical logic, and provides the ability to assert that something is not true. Connectives are the binary operators which combine predicates, with the best known of these being AND ($\land$) and OR ($\lor$). Quantifiers work by specifying the values which predicates can take on. These include FORALL ($\forall$) and EXISTS ($\exists$). An example of a statement in classical logic could therefore be: $\exists \text{House}. \text{Dog} \land \text{Cat}$. Which is to say that there exists a house such that there is a dog and a cat living there.

While classical logic is useful, it fails to represent the nuances associated with the world around us. It makes less sense to deal with absolute truth values when the distinctions between true and false are blurred [Zadeh 2008]. For example, if one was to say that John is Tall, in classical logic there would have to be a cut off point above which anyone is considered Tall. Conversely, anyone below this cut off point is $\neg$Tall, regardless of how close to the cut off point they are. To reason about indistinct concepts like this, a fuzzy logic system based off the idea of fuzzy set membership has been developed. Where previously John was either Tall or $\neg$Tall, he can now be
**Background**

Tall to degree 0.7. This is more natural, as two people on either side of, but close to, the classical cut off point are now represented by a similar degree of truth. These membership values fall in the range [0, 1], where the integer values 0 and 1 represent absolute falsehood and truth respectively.

### 1.2.1 Łukasiewicz Definitions

Łukasiewicz logic can be thought of as an implementation of fuzzy logic. It defines operators for conjunctions and implications that combine the truth values of their component concepts in intuitive and mathematically sound ways. Table 1.1 defines operators over the \([0, 1]\) valued variables [Hay 1963]. Note that the definition here for conjunction (\(\land\)) is that of strong conjunction. The operators \(\lor\), \(\forall\), and \(\Rightarrow\) are, for the purposes of this paper, undefined. This is because they can be written as combinations of the defined operators, as seen in table 1.2.

To use an example from Kulacka et al., when casting for a movie, a director may consider someone who is only moderately popular \((P = 0.5)\) and moderately talented \((T = 0.5)\) as altogether unsuitable for the role \((P \land T = 0)\). This kind of reasoning is where Łukasiewicz logic is good at representing our human intuition.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Łukasiewicz Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\neg A)</td>
<td>1 - A</td>
</tr>
<tr>
<td>(A \land B)</td>
<td>max((A + B - 1, 0))</td>
</tr>
<tr>
<td>(\exists A.B)</td>
<td>supremum(_B)(A \land B)</td>
</tr>
</tbody>
</table>

### 1.2.2 Tableau Methods

A tableau is a method of simplifying logical expressions. It works by creating a tree structure, where children in the tree are logical consequences of their parents [D’Agostino 1999]. A logical tableau branches where there is a disjunction, with different possibilities each having their own branch. If a contradiction occurs between a parent and child node in the tableau then the branch of the child is said to be closed or dead. If all branches are closed then the original formula is not satisfiable, though if even once branch still open at an atomic leaf node then the problem can be declared satisfiable. [D’Agostino 1999].
1.3 Description Logics

A Description Logic (DL) is one that allows inferences to be made from a set of known values. There are many variants of description logics, however this paper focuses on the description logic $\mathcal{ALC}$ [Kulacka et al. 2013]. Description logics allow inference to be made over combinations of concepts and roles. Concepts can be thought of as sets containing variables, where membership in the set is represented in the format $x : C$ [Baader et al. 2003]. This is to say that $x$ is a variable that is found in the concept $C$. This is conceptually similar to the propositional logic format $C(x)$, and can take on values in $[0, 1]$ to denote fuzzy membership. Roles are used to describe relationships between concepts and are represented as $(a, b) : R$. As an example of how roles are used, taken from Kulacka et al., consider the role $\text{Likes}$ and the variables $\text{Alice}$ and $\text{Bob}$. We can express the fact that $\text{Bob}$ (sadly) likes $\text{Alice}$ more than $\text{Alice}$ likes $\text{Bob}$ as the inequality:

$$(\text{Bob}, \text{Alice}) : \text{Likes} > (\text{Alice}, \text{Bob}) : \text{Likes}$$

The description logic $\mathcal{ALC}$ logic is made from two main building blocks [Kulacka et al. 2013]. The first of these is a Terminology Box (TBox), which describe relationships between different concepts. For instance, the concept $\text{LikesDogs}$ is a subset of the concept $\text{LikesAnimals}$. Our TBox here would be $\text{LikesDogs} \subset \text{likesAnimals}$. Such relationships allow complex hierarchies between concepts to be expressed [Baader et al. 2003]. The other building block is an Assertion Box (ABox), which is an assertion to be made. These contain concept and roles assertions that, to build on the previous example, express things like Sarah likes dogs ($\text{Sarah} : \text{LikesDogs}$). ABoxes such as this are used to hold the truths that our inference is based upon.

1.4 Approach in Kulacka et al.

Kulacka et al. introduces a tableau method for simplifying and deciding Łukasiewicz $\mathcal{ALC}$. The specifics of the DL are slightly changed, as now an individual interpretation for a concept is no longer an assertion ($\text{Sarah} : \text{likesDogs} \in [\text{true}, \text{false}]$) but a membership value expressed as a Łukasiewicz expression. ABox assertions are therefore slightly modified to be linear inequalities of role and concept sums ($\text{Sarah} : \text{LikesDogs} > 0.4$). For this particular implementation, TBoxes are not used as they result in problems potentially becoming undecidable when the TBox is unbounded [Kulacka et al. 2013]. Rules are proposed for how to generate consequences, with some rules introducing branching in the tableau. The rules can be found in Figure 1.1. Each of these rules is an implementation of the Łukasiewicz definitions of the operators, and take the branching that was previously done in the mixed integer linear program and apply it directly to the formulae themselves.

The technique in the paper realizes the best known complexity bound, $\text{NEXP} – \text{TIME}$, through use of ABoxes to contain linear inequalities containing sums of roles
and concepts [Kulacka et al. 2013]. Furthermore, if there is a leaf in the tableau with no clashes then the problem can be declared satisfiable as a whole. This means that in practice the algorithm does not need to explore the entire search space, and hence can do much better than the upper bound. While linear programming is still necessary to detect clashes at the leaves of the tableau, this is standard linear programming rather than mixed integer linear programming [Kulacka et al. 2013].

It is interesting to note here that applying the left existential rule \((\exists L)\) generates a new label, or variable, that has not appeared in the problem so far. There are only a finite number of these artificial variables that can be added, however, so termination is assured [Kulacka et al. 2013]. Another interesting observation is that the right existential rule \((\exists R)\) requires a related role somewhere in the same ABox.
Implementation Details

2.1 Goals

The primary goal for this project was to create a solver that efficiently determined the satisfiability of a given formula. The tableau method described in [Kulacka et al. 2013] was to be used and evaluated in the process. Given this method, the satisfying assignment of variables was not needed, just the existence of such an assignment. Different branching and search methods were explored once the implementation became stable, looking into how the exploration of tableau affected solution times.

2.2 Tools and Technologies

Scala was chosen as the language of implementation. This allowed the solver to take advantage of functional programming paradigms such as pattern matching while also being able to leverage off the numerous libraries available for programs running on the JVM.

Gurobi was chosen as the linear programming solver because it allows problems to be checked for feasibility quickly and simply. Checking only for feasibility is quicker than optimising a linear program, and given that we are only interested in satisfiability Gurobi was told not to optimise the truth values in the formula. It has a Java API that was callable through scala, although a wrapper called scaLP was found on GitHub. This was used as it allowed problems to be constructed in a more natural way.

The program reads a problem from a file using a Scala ParserCombinator to match nested concepts. The initial, hierarchical representation of the problem is recursively built and then passed to the solver. The solver allows the user to select both a branching strategy and a search method. Using these values it passes the ABox to the solver function matching the rule application order specified by the user. Once a leaf node in the tableau has been found, representing an ABox where no rules are applicable, each linear inequality in the ABox is used as a constraint in Gurobi. If the problem is feasible then the solver returns a message to say that the original problem is satisfiable. If not then the solver closes the branch and continues to explore the tableau. If all branches in the tableau are closed then the solver concludes that the original problem
is unsatisfiable. The solver is called as follows:

```scala
cala fuzzyLTC.jar <problem> [branchStrategy] [searchMethod]
```

Where `branchStrategy` is one of `[early, late]` and `search method` is one of `[bfs, dfs]`. The default behaviour of the solver is the branch early and use BFS. For this implementation, early branching applies the rules in the order specified by 2.1 and late branching applies the rules as seen in 2.2.

\[
\land, \neg L, \neg R, \land, \exists L, \exists R \quad (2.1)
\]

\[
\neg L, \neg R, \land, \neg L, \exists R, \exists L \quad (2.2)
\]

These different strategies decide how the tableau will be expanded.

### 2.3 Problem Definition

Problem files consist of a single inequality with sums of roles and concepts on the left and a target value on the right. Table 2.1 shows how the Łukasiewicz operators are represented in the input files. Operators are expressed in prefix notation for these input files, however each concept is also bracketed to make the file more human readable. An example problem is shown in equation 2.3.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Problem Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬C</td>
<td>(NOT(C))</td>
</tr>
<tr>
<td>C₁ ∧ C₂</td>
<td>(AND(C₁)(C₂))</td>
</tr>
<tr>
<td>∃R.C</td>
<td>(EXISTSR(C))</td>
</tr>
</tbody>
</table>

\[
x_1 : (AND(NOT(C₁))(C₂)) + x_2 : (EXISTSR(C₁)) > 0.4 \quad (2.3)
\]

#### 2.3.1 Right Existentials

An issue was uncovered in the course of implementing the tableau in the paper. If an existential concept was to appear on the right of an inequality with no supporting roles on the left then the right existential rule can not be applied. If this were to happen the existential remains on the right and the ABox cannot be reduced to an atomic form. This is a problem because linear programming at the leaves of the tableau was only meant to detect clashes between atomic concepts. There is actually the possibility that the leaves of the tableau can contain existentials and therefore the linear programming formulation must be changed.

Given that every inequality take the form \( \Gamma > \Delta \), setting any remaining right existentials to zero maximises the likelihood that the problem will be satisfied. It
seems at first that we are therefore considering a relaxation of the original problem, however zero is a valid truth value for any remaining existential concepts and as such this does not impact the completeness of the solver.
3.1 Experimental Design

Two axes of difference, resulting in four solvers, were tested. The first area of interest was to look into whether it was better to branch earlier or later in the tableau. The motivation for this was that early branching can help to reduce the number of distinct concepts we have to work with. This comes at the cost of potentially executing the same rule instance over multiple different branches; a duplication of effort. The other test was to investigate expanding the tableau in a breadth or depth first fashion. The idea here is that a depth first approach would potentially reach a leaf node faster, though if the linear program at this leaf is infeasible then the solver essentially has to start back at the root.

Two different tests were carried out to evaluate the performance of the different configurations. The first of these compared run times over a common batch of problems, the second compared the run times over single problems.

To perform the tests, a script was written to generate a random set of test formulae. For the batch test, 15 problems were randomly generated and each configuration solved the batch 100 times. For the individual test, 1000 problems were generated and the mean time it took to check a single problem was recorded.

The second test was used as a validator for the first, the reason being that different subtleties in the problems could produce different results. It was expected that the best performer would be early branching with a depth first search. This is because the branching rules often eliminate concepts from one of their conclusions, meaning it could be quicker to get to a leaf node. Furthermore, the depth first search would also aim to get to any leaf as soon as possible. Most of the problems in the test cases were broadly satisfiable, and as such almost any leaf node would represent a feasible linear programming problem.

3.2 Results

The results of each of these tests are displayed in tables 3.1 and 3.2. The results reported are in the format (mean, std dev) and are reported in seconds.
For the batch results, DFS was significantly better than BFS for both branching strategies, though no significant difference was found between late BFS and early DFS. The differences between early and late branching were statistically significant for both search strategies as well.

No statistically significant results were obtained with the single solution results, though the solution times for DFS were lower than the respective BFS times.

Table 3.1: Results of the Batch Solution Test: ($\mu, \sigma$)

<table>
<thead>
<tr>
<th></th>
<th>N=100, B=15</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Early Branching</td>
<td>(351.36, 0.10)</td>
<td>(350.65, 0.53)</td>
<td></td>
</tr>
<tr>
<td>Late Branching</td>
<td>(350.66, 0.01)</td>
<td>(349.92, 0.93)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Results of the Single Solution Test: ($\mu, \sigma$)

<table>
<thead>
<tr>
<th></th>
<th>N=1000</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Early Branching</td>
<td>(0.59, 192.66)</td>
<td>(0.25, 6.81)</td>
<td></td>
</tr>
<tr>
<td>Late Branching</td>
<td>(0.94, 312.30)</td>
<td>(0.67, 408.75)</td>
<td></td>
</tr>
</tbody>
</table>

### 3.3 Discussion

#### 3.3.1 Batch Test
The results obtained through the batch experiment confirmed that DFS was quicker than BFS, though surprisingly also found that late branching was better than early branching. These results confirmed the hypothesis that DFS would be faster because it can reach a leaf node faster than BFS can. Late branching looks to be faster based on these experiments, which indicates that the duplication of effort associated with having multiple similar branches creates too much overhead.

The weakness in this experiment lies in the fact that a single batch was created for the tests. This means that the results obtained, while being statistically significant, apply to that specific problem set. It is expected that different problem sets will produce different results according to the specific properties of the batches.

#### 3.3.2 Single Problem Test
The single problem test revealed no significant differences in the solver configurations tested. Depth first search with an early branching strategy had a substantially lower variance though, indicating that it potentially handled the 'hard' problems better than the other configurations. This would be in line with our intuition that branching early and looking to the leaves of the tableau would be faster, especially for tableaus where the branching factor is high. While the results were not significant, each configuration solves problems of this format in less than a second on average, so the primary goal of the project has been met.

The problems were generated randomly, with no control over their component concepts. The test was meant to benchmark average case performance, though the
problems themselves could have been separated out to give a more fine grained set of results.

3.3.3 Future Directions

This project was primarily focused on the implementation of the technique developed in Kulacka et al. As such, the software is to be used to aid researchers check for satisfiability of the formulae they work with. There is scope for more specific testing of the solver to determine the types of problems it struggles to solve. Furthermore the project could be easily extended to handle more generic forms of input formulae, carrying out the transformations itself. Finally, a heuristic search solver could be implemented if a valid heuristic can be developed to estimate the number of rule applications required to reach a leaf node. This could speed up the search significantly if implemented correctly.

Different parameters could be used to target different use cases. If the goal was to prove a formula satisfiable then applying rules early and using DFS could be preferable, as we do not expect to expand the entire tableau. To show that a problem is unsatisfiable however it could be better to branch late, as this should keep the size of the expanded tableau to a minimum.

Gurobi does require a license to use, however academic licenses are available. As such, the calls to Gurobi are wrapped in a function that converts an ABox into a linear program, and as such a different linear program solver can easily be substituted if need be. This would be as simple as writing a function to turn an ABox into the required linear program format of the new solver.
Conclusion

A fast, efficient solver was developed to check the satisfiability of Łukasiewicz fuzzy ABox assertions. The scala implementation offers the user four distinct configurations that allow control over how the problem is to be explored. In its current form the solver is ready to use, as any expression can be represented in the form read by the solver. In future it could be beneficial to accept a more flexible input format and apply logical transformations internally.

The different solver configurations were tested over single problems as well as batches, and while statistically significant results were found in the batch test, the differences were not big enough to make a practical difference. The single problem test revealed little difference between solvers, though knowledge of the problem structure can be used to select which parameters would be best to solve the given problem.
Bibliography


