Sapienza
Università di Roma

FOND Planning for LTL_f Goals: Theory and Implementation

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea Magistrale in Engineering in Computer Science

Candidate
Francesco Fuggitti
ID number 1735212

Thesis Advisor
Prof. Giuseppe De Giacomo

Academic Year 2017/2018

Thesis not yet defended

FOND Planning for LTL_f Goals: Theory and Implementation
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: fuggitti.1735212@studenti.uniroma1.it

Abstract

MDPs extended with LTL_f non-Markovian rewards have recently attracted interest as a way to specify rewards declaratively. In this thesis, we discuss how a reinforcement learning agent can learn policies fulfilling $\text{LTL}_f/\text{LDL}_f$ goals. In particular we focus on the case where we have two separate representations of the world: one for the agent, using the (predefined, possibly low-level) features available to it, and one for the goal, expressed in terms of high-level (human-understandable) fluents. We formally define the problem and show how it can be solved. Moreover, we provide experimental evidence that keeping the RL agent feature space separated from the goal's can work in practice, showing interesting cases where the agent can indeed learn a policy that fulfills the $\text{LTL}_f/\text{LDL}_f$ goal using only its features (augmented with additional memory).

Contents

Chapter 1

LTL$_f$2DFA

In this chapter we will present LTL$_f$2DFA, a software package written in Python.

1.1 Introduction

LTL$_f$2DFA is a Python tool that processes a given LTL$_f$ formula (with past and future operators) and generates the corresponding minimized DFA using MONA (Elgaard et al., 1998). The main features provided by the library are:

- parsing an LTL$_f$ formula with past or future operators;
- translation of an LTL$_f$ formula to MONA program;
- conversion of an LTL$_f$ formula to DFA automaton.

LTL$_f$2DFA can be used with Python>=3.6 and has the following dependencies:

- PLY, a pure-Python implementation of the popular compiler construction tools Lex and Yacc. It has been employed for parsing the input LTL$_f$ formula;
- MONA, a C++ tool that translates formulas to DFA. It has been used for the generation of the DFA;
- Dotpy, a Python library able to parse and modify `.dot` files. It has been utilized for post-processing the MONA output.

The package is available to download on PyPI and you can install it typing in the terminal:

```
pip install ltlf2dfa
```

All the code is available online on GitHub[1], it is open source and it is released under the MIT License. Moreover, LTL$_f$2DFA can also be tried online at ltlf2dfa.diag.uniroma1.it.

[1] https://github.com/Francesco17/LTLf2DFA

1.2 Package Structure

The structure of the LTL$_f$2DFA package is quite simple. It consists of a main folder called `ltlf2dfa/` which hosts the most important library's modules:

- `Lexer.py`, where the Lexer class is defined;
- `Parser.py`, where the Parser class is defined;
- `Translator.py`, where the main APIs for the translation are defined;
- `DotHandler.py`, where we the MONA output is post-processed.

In the following paragraphs we will explore each module in detail.

1.2.1 Lexer.py

In the `Lexer.py` module we can find the declaration of the `MyLexer` class which is in charge of handling the input string and tokenize it. Indeed, it implements a tokenizer that splits the input string into declared individual tokens. To our extent, we have defined the class as in Listing 1.1

Listing 1.1. `Lexer.py` module

```
import ply.lex as lex

class MyLexer(object):

    reserved = {
        'true':    'TRUE',
        'false':   'FALSE',
        'X':       'NEXT',
        'U':       'UNTIL',
        'E':       'EVENTUALLY',
        'G':       'GLOBALLY',
        'Y':       'PASTNEXT', #PREVIOUS
        'S':       'PASTUNTIL', #SINCE
        'O':       'PASTEVENTUALLY', #ONCE
        'H':       'PASTGLOBALLY'
    }
    # List of token names. This is always required
    tokens = (
        'TERM',
        'NOT',
        'AND',
        'OR',
        'IMPLIES',
```

```
        'DIMPLIES',
        'LPAR',
        'RPAR'
    ) + tuple(reserved.values())

    # Regular expression rules for simple tokens
    t_TRUE = r'T'
    t_FALSE = r'F'
    t_AND = r'\&'
    t_OR = r'\|'
    t_IMPLIES = r'\->'
    t_DIMPLIES = r'\<->'
    t_NOT = r'\~'
    t_LPAR = r'\('
    t_RPAR = r'\)'
    # FUTURE OPERATORS
    t_NEXT = r'X'
    t_UNTIL = r'U'
    t_EVENTUALLY = r'E'
    t_GLOBALLY = r'G'
    # PAST OPERATOR
    t_PASTNEXT = r'Y'
    t_PASTUNTIL = r'S'
    t_PASTEVENTUALLY = r'O'
    t_PASTGLOBALLY = r'H'

    t_ignore = r' '+'\n'

    def t_TERM(self, t):
        r'[a-z]+'
        t.type = MyLexer.reserved.get(t.value, 'TERM')
        return t # Check for reserved words

    def t_error(self, t):
        print("Illegal character '%s' in the input formula" % t.value[0])
        t.lexer.skip(1)

    # Build the lexer
    def build(self,**kwargs):
        self.lexer = lex.lex(module=self, **kwargs)
```

Firstly, we have defined the reserved words within a dictionary so to match each reserved word with its identifier. Secondly, we have defined the tokens list with all possible tokens that can be produced by the lexer. The tokens list is always required for the implementation of a lexer. Then, each token has to be specified by writing a regular expression rule. If the token is simple it can be specified using only a string. Otherwise, for non trivial tokens we have to write the regular expression in a class method as for our token `TERM`. In that case, defining the token rule as a method is useful also when we would like to perform other actions. After that, we have a method to handle unrecognized tokens and, finally, we can build the lexer.

1.2.2 Parser.py

In the `Parser.py` module we can find the declaration of `MyParser` class which implements the parsing component of `PLY`. The `MyParser` class operates after the Lexer has split the input string into known tokens. The main feature of the parser is to interpret and build the appropriate data structure for the given input. To this extent, the most important aspect of a parser is the definition of the *syntax*, usually specified in terms of a BNF grammar, that should be unambiguous. Furthermore, Yacc, the parsing component of `PLY`, implements a parsing technique known as LR-parsing or shift-reduce parsing. In particular, this parsing technique works on a bottom up fashion that tries to recognize the right-hand-side of various grammar rules. Whenever a valid right-hand-side is found in the input, the appropriate action code is triggered and the grammar symbols are replaced by the grammar symbol on the left-hand-side and so on until there is no more rule to apply. The parser implementation is shown in Listing 1.2

Listing 1.2. `Parser.py` module

```python
import ply.yacc as yacc
from ltlf2dfa.Lexer import MyLexer

class MyParser(object):

    def __init__(self):
        self.lexer = MyLexer()
        self.lexer.build()
        self.tokens = self.lexer.tokens
        self.parser = yacc.yacc(module=self)
        self.precedence = (

            ('nonassoc', 'LPAR', 'RPAR'),
            ('left', 'AND', 'OR', 'IMPLIES', 'DIMPLIES', 'UNTIL', \
             'PASTUNTIL'),
            ('right', 'NEXT', 'EVENTUALLY', 'GLOBALLY', \
            'PASTNEXT', 'PASTEVENTUALLY', 'PASTGLOBALLY'),
            ('right', 'NOT')
        )

    def __call__(self, s, **kwargs):
```

```
        return self.parser.parse(s, lexer=self.lexer.lexer)

    def p_formula(self, p):
        '''
        formula : formula AND formula
                | formula OR formula
                | formula IMPLIES formula
                | formula DIMPLIES formula
                | formula UNTIL formula
                | formula PASTUNTIL formula
                | NEXT formula
                | EVENTUALLY formula
                | GLOBALLY formula
                | PASTNEXT formula
                | PASTEVENTUALLY formula
                | PASTGLOBALLY formula
                | NOT formula
                | TRUE
                | FALSE
                | TERM
        '''

        if len(p) == 2: p[0] = p[1]
        elif len(p) == 3:
            if p[1] == 'E': # E(a) == true UNITL A
                p[0] = ('U','T', p[2])
            elif p[1] == 'G': # G(a) == not(eventually (not A))
                p[0] = ('~',('U', 'T', ('~',p[2])))
            elif p[1] == 'O': # O(a) = true SINCE A
                p[0] = ('S','T', p[2])
            elif p[1] == 'H': # H(a) == not(pasteventually(not A))
                p[0] = ('~',('S', 'T', ('~',p[2])))
            else:
                p[0] = (p[1], p[2])
        elif len(p) == 4:
            if p[2] == '->':
                p[0] = ('|', ('~', p[1]), p[3])
            elif p[2] == '<->':
                p[0] = ('&', ('|', ('~', p[1]), p[3]), ('|', ('~', p[3]),\
                p[1]))
            else:
                p[0] = (p[2],p[1],p[3])
        else: raise ValueError
```

```


    def p_expr_group(self, p):
        '''
        formula : LPAR formula RPAR
        '''
        p[0] = p[2]

    def p_error(self, p):
        raise ValueError("Syntax error in input! %s" %str(p))
```

As we can see, as soon as the parser is instantiated it builds the lexer, gets the tokens and defines their precedence if needed. Then, we have defined methods of the `MyParser` class that are in charge of constructing the syntax tree structure from tokens found by the lexer in the input string. In our case, we have chosen to use as data structure a tuple of tuples as it is the one of the simplest data structure in Python. In general, a tuple of tuples represents a tree where each node represents an item in the formula.

For instance, the LTL$_f$ formula $\varphi = G(a \rightarrow Xb)$ is represented as $('\sim', ('U', 'T', ('\sim', ('|', ('\sim', 'a'), ('X', 'b')))))$ as depicted in Figure 1.1. Finally, as in the `MyLexer` class, we

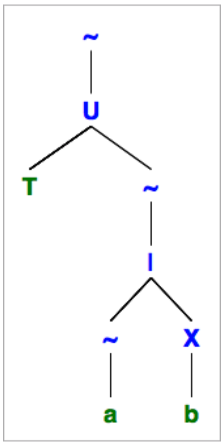

Figure 1.1. The syntax tree generated for the formula "$G(a \sim Xb)$". Symbols are in green while operators are in blue.

have to handle errors defining a specific method.

LTL$_f$2DFA can be used just for the parsing phase of an LTL$_f$ formula as shown in Listing 1.3.

Listing 1.3. How to use only the parsing phase of LTL$_f$2DFA.

```
from ltlf2dfa.Parser import MyParser

formula = "G(a->Xb)"
parser = MyParser()
```

```
parsed_formula = parser(formula)

print(parsed_formula) # syntax tree as tuple of tuples
```

1.2.3 Translator.py

The `Translator.py` module contains the majority of APIs that the LTL$_f$2DFA package exposes. Indeed, this module consists of a Translator class which concerns the core feature of the package: the translation of an LTL$_f$ formula into a DFA. Since the package takes advantage of the MONA tool for the formula conversion, the `Translator` class has to translate the given formula in the syntax recognized by MONA, create the input program for MONA and, finally, invoke MONA to get back the resulting DFA in the Graphviz[2] format. The main methods of the `Translator` class are:

- `translate()`, which starting from the formula syntax tree generated in the parsing phase translates it into a string using the syntax of MONA (Klarlund and Møller, 2001);
- `createMonafile(flag)`, which, as the name suggests, creates the program *.mona* that will be given as input to MONA. The flag parameter is going to be `True` of `False` whether we need to compute also DECLARE assumptions or not;
- `invoke_mona()`, which invokes MONA in order to obtain the DFA.

1.2.4 DotHandler.py

[2]Graphviz is open source graph visualization software. For further details see https://www.graphviz.org

Bibliography

Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: new techniques for WS1S and WS2S. In *Proc. 10th International Conference on Computer-Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 516–520. Springer-Verlag, June/July 1998.

Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.