Alma Mater Studiorum - University of Bologna

COMPUTER SCIENCE AND ENGINEERING - DISI

ARTIFICIAL INTELLIGENCE

Labeled Prolog: a computational model in 2p-Kt

Master degree thesis

Supervisor

Prof.Roberta Calegari

Co-supervisor

Prof. Giovanni Ciatto

Candidate

Giuseppe Boezio

ACADEMIC YEAR 2021-2022- THIRD SESSION

Abstract

Content of the abstract

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chapter 2

Background notions

2.1 The Prolog Language

2.1.1 Brief History

Prolog stands for *PROgrammation en LOGique* and it emerged during 1970s as a way to use logic as a programming language. The early developers of this language were Robert Kowalski, Maarten van Emden and Alain Colmelauer. The programming language, Prolog, was born of a project aimed not at producing a programming language but at processing natural languages; in this case, French [4]. The project gave rise to a preliminary version of Prolog at the end of 1971 and a more definitive version at the end of 1972. Prolog gained a lot of attraction from the computing society as it was the very first logic programming language. The language still holds considerable importance and popularity among the logic programming languages and comes with a range of commercial as well as free implementations.
Prolog is used for different kind of tasks such as:

- theorem proving [3]
- expert systems [9]
- knowledge representation [6]
- automated planning [10]
- natural language processing [8]

2.1.2 Concepts

Syntax and semantics of Prolog are described in ISO standard ISO/IEC 13211. Prolog is a logic programming language; this means that it is used to describe known facts and relationships about a problem and less about prescribing the sequence of steps taken by a computer to solve the problem. When a computer is programmed in Prolog, the actual way the computer carries out the computation is specified partially by the logic declarative semantics of Prolog, partly by what new facts can be inferred from the given ones, and only partly by explicit control information supplied by the programmer.

Prolog is used to solve problems which involve objects and relations among them. The main features of the programming language are:

- specifyng some *facts* about some objects and their relationships
- defining some *rules* about objects and their relationships
- asking *questions* about objects and their relationships

Prolog programs are built from terms. A term is either a constant, a variable or a structure.

2.1.2.1 Constants

A constant is a sequence of characters which denotes a specific object or relationship. A constant can be an atom or a number. All constants begin with a lower case letter.

Example

a is an atom
12 is a number

2.1.2.2 Variables

A variable looks like an atom except it has a name beginning with capital letter or underline signed. A variable should be thought of as standing for some objects we are unable or unwilling to name at the time we write the program.

Example

X and *Answer* are valid names for variables

2.1.2.3 Structures

A structure is a collection of other objects called *components*. Structures help to organize the data in a program because they permit a group of related information to be treated as a single object instead of separate entities. A structure is written in Prolog by specifying its *functor* and its *components*. The coponents are enclosed in round brackets and separated by commas. The functor is written just before the opening round brackets.

Example

owns(john,book) is a structure having owns as functor and, john and book as components

2.1.2.4 Facts

A fact is a relation among objects which are all *ground*. This means that a fact is a *structure* which does not contain any variables among its components. A fact is written as a structure followed by a dot (.). The names of the objects that are enclosed within the round brackets in each fact are called *arguments*. The name of the relationship which comes just before the round brackets is called *predicate*.

Example

king(john,france). is a fact

2.1.2.5 Rules

A rule is a disjunction of predicates, where at most one is not negated, written in the following way:

$$Head : -Body.$$

where Head is a predicate, Body is a conjunction of predicates and the symbol ":-" means that the body implies the head. This kind of structure is called Horn clause. A Prolog program can be seen as a list (because order matters) of Horn clauses called *theory*. Facts could be seen as Rules having Body equals to true. Rules are used to describe some complex relations among objects of the domain of discourse and differently from facts can contain variables.

Example

motherOf(X,Y) :- parentOf(X,Y),female(X).

The aforementioned description of Prolog language has been adapted from [2].

2.1.2.6 Unification

A substitution is a function which associates a variable to a given term. The most general unifier (m.g.u.) is the substitution which allows to transform two terms making them equals such that all other substitutions can be obtained through a composition with this one. The unification is a process whereby two structures are made equals via substitution and it is used several times during the Prolog resolution process.

2.1.2.7 Resolution

The resolution in Prolog happens in the following way:the interpreter tries to verify whether a conjunction of predicates (the goal) provided by the user can be derived from the current program or not and in the case it could, it provides a computed answer substitution (c.a.s.) which is a set of substitutions which allow to make true the user's goal.
Prolog resolution process is called SLD (Selective Linear Definite clause resolution) and works as follows:
SLD resolution implicitly defines a search tree of alternative computations, in which the initial goal clause is associated with the root of the tree. For every node in the tree and for every definite clause in the program whose positive literal unifies with the selected literal in the goal clause associated with the node, there is a child node associated with the goal clause obtained by SLD resolution. A leaf node, which has no children, is a success node if its associated goal clause is the empty clause. It is a failure node if its associated goal clause is non-empty but its selected literal unifies with no positive literal of definite clauses in the program. SLD resolution is non-deterministic in the sense that it does not determine the search strategy for exploring the search tree. Prolog searches the tree depth-first, one branch at a time, using backtracking when it encounters a failure node. Depth-first search is very

efficient in its use of computing resources, but is incomplete if the search space contains infinite branches and the search strategy searches these in preference to finite branches: the computation does not terminate. The SLD resolution search space is an or-tree, in which different branches represent alternative computations.[5]

2.1.3 2p-Kt

2p-Kt is a general, extensible, and interoperable ecosystem for logic programming and symbolic AI written in Kotlin which supports the Prolog ISO standard.
2p-kt is the evolution of another project called tuProlog [1]. To support reusability

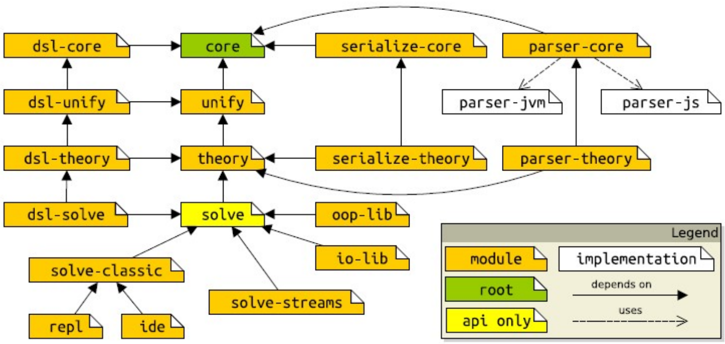

Figure 2.1: 2P-Kt project map. LP functionalities are partitioned into some loosely-coupled and incrementally-dependent modules.

2p-Kt is divided into several modules described as follows:

- **:core**: exposes data structures for knowledge representation via terms and clauses, other than methods supporting their manipulation
- **:unify**: used to compare and manipulate logic terms through logic unification
- **:theory**: in-memory storage of clauses into ordered (e.g. queues) or unordered (e.g. multisets) data structures, and their efficient retrieval via pattern-matching
- **:serialize-* and :parser-***: used to perform ancillary operations such as serialization and parsing
- **:solve**: aspects which are orthogonal w.r.t. any particular resolution strategy —e.g. errors management, extensibility via libraries, I/O, etc
- **:solve-***: modules which implement a specific resolution strategy

- **:repl and :ide**: provide CLI and GUI

The structure of the project can be seen in figure 2.1.
2p-Kt provides a well-grounded technological basis for implementing/experimenting/extending the many solutions proposed in the literature—e.g., abductive inference, rule induction, probabilistic reasoning and labelled LP.

2.2 Constraint Programming

Constraint Programming is a paradigm for solvig combinatorial problems where different constraints are imposed on feasible solutions for different decision variables, each having its own domain. Constraints are relations among variables which limit the values decision variables can assume in feasible solutions [11]

2.2.1 Brief History

In artificial intelligence interest in constraint satisfaction developed in two streams. In some sense a common ancestor of both streams is Ivan Sutherland's groundbreaking 1963 MIT Ph.D. thesis, "Sketchpad: A man-machine graphical communication system". In one stream, the versatility of constraints led to applications in a variety of domains, and associated programming languages and systems. This stream we can call the language stream. In 1964 Wilkes proposed that algebraic equations be allowed as constraint statements in procedural Algol-like programming languages, with relaxation used to satisfy the constraints. Around 1967, Elcock developed a declarative language, Absys, based on the manipulation of equational constraints.

2.2.2 Concepts

There are mainly two types of constraint programming problems: CSP (Constraint Satisfaction Problem) and COP (Constraint Optimization Problem).

A *constraint satisfaction problem* (CSP) involves finding solutions to a constraint network, that is, assignments of values to its variables that satisfy all its constraints. Constraints specify combinations of values that given subsets of variables are allowed to take.
A constraint can be specified extensionally by the list of its satisfying tuples, or intensionally by a formula that is the characteristic function of the constraint.

A *constraint optimization problem* (COP) is basically the same as a CSP but in addition to the aforementioned constraints there is another one which consists of finding a solution which minimizes or maximizes a certain function.

2.2.3 Constraint Logic Programming

Constraint Logic Programming (CLP) began as a natural merger of two declarative paradigms: constraint solving and logic programming [7].
Viewing the subject rather broadly, constraint logic programming can be said to involve the incorporation of constraints and constraint "solving" methods in a logic-based language. This characterization suggests the possibility of many interesting languages, based on different constraints and different logics. However, to this point, work on CLP has almost exclusively been devoted to languages based on Horn clauses.
Prolog can be said to be a CLP language where the constraints are equations over the algebra of terms (also called the algebra of finite trees, or the Herbrand domain). The equations are implicit in the use of unification.

2.2.4 SWI Prolog - CLP libraries

SWI-Prolog is an implementation of the Prolog language which is strong in education because it is free and portable, but also because of its compatibility with textbooks and its easy-to-use environment.
SWI-Prolog is used as an embedded language where it serves as a small rule subsystem in a large application. The syntax and set of built-in predicates is based on the ISO standard [15].

2.2.4.1 CLP(X) libraries

CLP(X) stands for constraint logic programming over the domain X. Plain Prolog can be regarded as CLP(H), where H stands for Herbrand terms.
SWI Prolog supports:

- **CLP(FD)** for integers
- **CLP(B)** for boolean variables
- **CLP(Q)** for rational numbers

- **CLP(R)** for floating point numbers

All constraints contained in these libraries will be deeply explained in chapter 3.

CLP(FD) has two main usages:

- declarative integer arithmetics
- solving combinatorial problems such as planning, scheduling and allocation tasks

The predicate of this library can be classified as:

- *arithmetic constraints*
- *membership constraints*
- *enumeration predicates*
- *combinatorial constraints*
- *reification predicates*
- *reflection predicates*

Practical usage of these constraints can be found in [14].

CLP(Q) and CLP(R) share basically the same constraints except for *bb_inf* constraint which is used to find a minimum in the case of mixed integer programming.

CLP(B) can be used to model and solve combinatorial problems such as verification, allocation and covering tasks. Benchmarks and usage examples of this library are available from [13] and [12].

Chapter 3

CLP in 2p-Kt

3.1 Requirements

As described in section 2.1.3, 2P-Kt is an exensible framework which has different mechanisms which could be used to add to the standard Prolog other features; in this case we deal with Constraint Logic Programing (CLP).
The main purpose of the following project is to implement CLP libraries in 2P-Kt having the following requirements:

- the interface of the predicates exposed by the libraries must be as close as possible to the one used by SWI Prolog for CLP
- there are not strict requirements about how to implement libraries but a good solution would exploit on existing classes in 2P-kt
- the different libraries could contain different predicates wrt SWI Prolog only if it is not possible to find another solution which is compatible with the current framework

3.2 Design

3.2.1 Common aspects

Four different libraries have been realized:

- **clp-core** for basic functionality of the other libraries
- **clpfd** for integer variables
- **clpqr** for rational and real variables

- **clpb** for boolean variables

clpqr contains basically the predicates of **clpq** and **clpr** because there is any distinction between rational and reals in 2P-Kt.

Libraries will be described in the following sections highlighting common and/or different aspects with respect to the SWI Prolog counterpart.

3.2.2 Constraint Logic Programming over Finite Domains

For a better explaination predicates will be divided in groups as described in section 2.2.4.1.

3.2.2.1 Arithmetic Constraints

All constraints supported by SWI Prolog are supported; constraints are the followings:

Constraint	Explaination
Expr1 #= Expr2	Expr1 equals Expr2
Expr1 #Ēxpr2	Expr1 is not equal to Expr2
Expr1 #>= Expr2	Expr1 is greater than or equal to Expr2
Expr1 #=< Expr2	Expr1 is less than or equal to Expr2
Expr1 #> Expr2	Expr1 is greater than Expr2
Expr1 #< Expr2	Expr1 is less than Expr2

Table 3.1: Arithmetic constraints in clpfd

Expression	Explaination
integer	Given value
variable	Unknown integer
-Expr	Unary minus
Expr + Expr	Addition
Expr * Expr	Multiplication
Expr - Expr	Subtraction
Expr Êxpr	Exponentiation
min(Expr,Expr)	Minimum of two expressions
max(Expr,Expr)	Maximum of two expressions
Expr mod Expr	Modulo induced by floored division
abs(Expr)	Absolute value
Expr div Expr	Floored integer division

Table 3.2: Arithmetic expressions in clpfd

Expr1 and *Expr2* are **arithmetic expressions**. *Expr rem Expr* and *Expr // Expr* are not supported. *rem* is modulo induced by truncated division whereas // is truncated integer division.

3.2.2.2 Membership Constraints

These constraints are used to specify the admissible domains of variables. The predicates are:

- **Var in Domain**: Var is an element of Domain; Domain is either an integer or an interval (expressed as Lower..Upper)
- **Vars in Domain**: The variables in the list Vars are elements of Domain

It is not current supported (union of domains) as expression for building a domain.

3.2.2.3 Enumeration predicates

These predicates are used to customize the search to find a feasible assignments of all variables such that all constraints are satisfied.
The predicates are **labeling/2 and label/1**.

labeling(Options, Vars)

Assign a value to each variable in Vars; Options is a list of options that let exhibit some control over the search process. Several categories of options exist:

- **variable selection strategy**: it can the order in which the variable occurs (*leftmost*, it is the default), the leftmost variable with smallest domain (*ff*), the variables with smallest domains, the leftmost one participating in most constraints (*ffc*), the leftmost variable whose lower bound is the lowest (*min*) or the leftmost variable whose upper bound is the highest (*max*)
- **value order**: elements of the chosen variable's domain in ascending order (*up*, it is the default) or domain elements in descending order (*down*)
- **branching strategy**: For each variable X, a choice is made between X = V and X $\#\bar{V}$, where V is determined by the value ordering options. This option is called *step*, it is the default and the only branching option supported

At most one option of each category can be specified, and an option must not occur repeatedly.
The order of solutions can be influenced with:

- min(Expr)

- max(Expr)

This generates solutions in ascending/descending order with respect to the evaluation of the arithmetic expression Expr. Labeling Vars must make Expr ground. If several such options are specified, they are interpreted from left to right.
This predicate does not support as options the following branching strategies:

- *enum*: For each variable X, a choice is made between X = V_1, X = V_2 etc., for all values V_i of the domain of X. The order is determined by the value ordering options.
- *bisect*: For each variable X, a choice is made between X #=< M and X #> M, where M is the midpoint of the domain of X.

label(Vars)

Equivalent to labeling([], Vars).

3.2.2.4 Global constraints

A global constraint expresses a relation that involves many variables at once. The implemented constraints are the followings:

- **all_distinct(Vars)**: True iff Vars are pairwise distinct
- **sum(Vars, Rel, Expr)**: The sum of elements of the list Vars is in relation Rel to Expr. Rel is one of #=, #\=, #<, #>, #=< or #>=
- **scalar_product(Cs, Vs, Rel, Expr)**: True iff the scalar product of Cs and Vs is in relation Rel to Expr. Cs is a list of integers, Vs is a list of variables and integers. Rel is #=, #\=, #<, #>, #=< or #>=
- **lex_chain(Lists)**: Lists are lexicographically non-decreasing
- **tuples_in(Tuples, Relation)**: True iff all Tuples are elements of Relation. Each element of the list Tuples is a list of integers or finite domain variables. Relation is a list of lists of integers
- **serialized(Starts, Durations)**: Describes a set of non-overlapping tasks. Starts = $[S_1, ..., S_n]$, is a list of variables or integers, Durations = $[D_1, ..., D_n]$ is a list of non-negative integers. Constrains Starts and Durations to denote a

set of non-overlapping tasks, i.e.: S_i + D_i =< S_j or S_j + D_j =< S_i for all 1 =< i < j =< n

- **element(N, Vs, V)**: The N-th element of the list of finite domain variables Vs is V
- **global_cardinality(Vs, Pairs)**: Global Cardinality constraint. Vs is a list of finite domain variables, Pairs is a list of Key-Num pairs, where Key is an integer and Num is a finite domain variable. The constraint holds iff each V in Vs is equal to some key, and for each Key-Num pair in Pairs, the number of occurrences of Key in Vs is Num
- **circuit(Vs)**: True iff the list Vs of finite domain variables induces a Hamiltonian circuit. The k-th element of Vs denotes the successor of node k
- **cumulative(Tasks, Options)**: Schedule with a limited resource. Tasks is a list of tasks, each of the form task(S_i, D_i, E_i, C_i, T_i). S_i denotes the start time, D_i the positive duration, E_i the end time, C_i the non-negative resource consumption, and T_i the task identifier. Each of these arguments must be a finite domain variable with bounded domain, or an integer. The constraint holds iff at each time slot during the start and end of each task, the total resource consumption of all tasks running at that time does not exceed the global resource limit. Options is a list of options. Currently, the only supported option is *limit(L)* which is the global resource limit
- **cumulative(Tasks)**: Like the previous one but with *L = 1*
- **disjoint2(Rectangles)**: True iff Rectangles are not overlapping. Rectangles is a list of terms of the form F(X_i, W_i, Y_i, H_i), where F is any functor, and the arguments are finite domain variables or integers that denote, respectively, the X coordinate, width, Y coordinate and height of each rectangle.
- **chain(Zs, Relation)**: Zs form a chain with respect to Relation. Zs is a list of finite domain variables that are a chain with respect to the partial order Relation, in the order they appear in the list. Relation must be #=, #=<, #>=, #< or #>

Notes

Wrt global constraints provided by CLP(FD) library of SWI Prolog the following aspects are different:

- the index of the predicate **circuit/1** starts from 1 and not from 0 for implementation issues
- the predicate **all_different/1** has not been supported because it has the same usage of **all_distinct** but it has a weaker propagation which cannot be simulated
- predicates **automaton/3** and **automaton/8** have not been implemented because of the fact that these predicates are rarely used and tedious to use
- the predicate **global_cardinality/3** can be used but actually it throws an exception because the Option parameter cannot be supported

3.2.2.5 Reification Predicates

All relational constraints discussed in 3.2.2.1 can be reified. This means that their truth value is itself turned into a clpfd variable, so that it is possible to reason about whether a constraint holds or not. These predicates are reifiable themselves.

- # **Q**: Q does not hold
- **P** #<==> **Q**: P and Q are equivalent
- **P** #==> **Q**: P implies Q
- **P** #<== **Q**: Q implies P
- **P** #/ **Q**: P and Q hold
- **P** # **Q**: P or Q hold
- **P** # **Q**: Either P holds or Q holds, but not both
- **zcompare(Order, A, B)**: reify an arithmetic comparison of two integers

3.2.2.6 Reflection Predicates

Reflection predicates let obtain, in a well-defined way, information that is normally internal to this library. Predicates are:

- **fd_var(Var)**: True iff Var is a clpfd variable
- **fd_inf(Var, Inf)**: Inf is the infimum of the current domain of Var
- **fd_sup(Var, Sup)**: Sup is the supremum of the current domain of Var

- **fd_size(Var, Size)**: Reflect the current size of a domain. Size is the number of elements of the current domain of Var
- **fd_dom(Var, Dom)**: Dom is the current domain (see 3.2.2.3) of Var
- **fd_degree(Var, Degree)**: Degree is the number of constraints currently attached to Var

3.2.3 Constraint Logic Programming over Rationals and Reals

This library is very different from the previous one because variable definitions and constraints can be stated in the same structure. The main predicate is **{}(Constraints)** which allows to add the constraints given by Constraints to the constraint store.

Constraints can be defined using the following grammar:

<*Constraints*>	::=	<*Constraint*>	single constraint
	|	<*Constraint*> , <*Constraints*>	conjunction
	|	<*Constraint*> ; <*Constraints*>	disjunction
<*Constraint*>	::=	<*Expression*> < <*Expression*>	less than
	|	<*Expression*> > <*Expression*>	greater than
	|	<*Expression*> =< <*Expression*>	less or equal
	|	<=(<*Expression*>, <*Expression*>)	less or equal
	|	<*Expression*> >= <*Expression*>	greater or equal
	|	<*Expression*> =\= <*Expression*>	not equal
	|	<*Expression*> =:= <*Expression*>	equal
	|	<*Expression*> = <*Expression*>	equal
<*Expression*>	::=	<*Variable*>	Prolog variable
	|	<*Number*>	Prolog number
	|	+<*Expression*>	unary plus
	|	-<*Expression*>	unary minus
	|	<*Expression*> + <*Expression*>	addition
	|	<*Expression*> - <*Expression*>	substraction
	|	<*Expression*> * <*Expression*>	multiplication
	|	<*Expression*> / <*Expression*>	division
	|	abs(<*Expression*>)	absolute value
	|	sin(<*Expression*>)	sine
	|	cos(<*Expression*>)	cosine
	|	tan(<*Expression*>)	tangent
	|	exp(<*Expression*>)	exponent
	|	pow(<*Expression*>)	exponent
	|	<*Expression*> ^ <*Expression*>	exponent
	|	min(<*Expression*>, <*Expression*>)	minimum
	|	max(<*Expression*>, <*Expression*>)	maximum

Figure 3.1: clpqr constraints BNF

All constraints are supported except for *<Expression>* =\=*<Expression>* (not equal). This libraries contain also the following predicates:

- **entailed(Constraint)**: Succeeds if Constraint is necessarily true within the current constraint store. This means that adding the negation of the constraint to the store results in failure
- **inf(Expression, Inf)**: Computes the infimum of Expression within the current state of the constraint store and returns that infimum in Inf. This predicate does not change the constraint store
- **sup(Expression, Sup)**: Computes the supremum of Expression within the current state of the constraint store and returns that supremum in Sup. This predicate does not change the constraint store
- **minimize(Expression)**: Minimizes Expression within the current constraint store. This is the same as computing the infimum and equating the expression to that infimum
- **maximize(Expression)**: Maximizes Expression within the current constraint store. This is the same as computing the supremum and equating the expression to that supremum
- **bb_inf(Ints, Expression, Inf, Vertex, Eps)**: It computes the infimum of Expression within the current constraint store, with the additional constraint that in that infimum, all variables in Ints have integral values. Vertex will contain the values of Ints in the infimum. Eps denotes how much a value may differ from an integer to be considered an integer
- **bb_inf(Ints, Expression, Inf, Vertex)**: it behaves as the previous one but not use an error margin
- **bb_inf(Ints, Expression, Inf)**: as the previous one but without returning the values of the integers
- **dump(Target, Newvars, CodedAnswer)**: Returns the constraints on Target in the list CodedAnswer where all variables of Target have been replaced by NewVars. This operation does not change the constraint store

Notes

Expression	Explaination
0	false
1	true
variable	unknown truth value
Expr	logical NOT
Expr + Expr	logical OR
Expr * Expr	logical AND
Expr # Expr	exclusive OR
Expr =:= Expr	equality
Expr =Ēxpr	disequality (same as #)
Expr =< Expr	less or equal (implication)
Expr >= Expr	greater or equal
Expr < Expr	less than
Expr > Expr	greater than
+(Exprs)	n-fold disjunction
*(Exprs)	n-fold conjunction

Table 3.3: Admissible boolean expressions in clpb

Eps of the predicate **bb_inf/5** cannot be fully supported, the only admissible value is 0.

3.2.4 Constraint Logic Programming over Boolean Variables

All predicates of this library are based on the concept of *boolean expression.* A *boolean expression* is one of:

Supported predicates are:

- **sat(Expr)**: True iff the Boolean expression Expr is satisfiable
- **taut(Expr, T)**: If Expr is a tautology with respect to the posted constraints, succeeds with T = 1. If Expr cannot be satisfied, succeeds with T = 0. Otherwise, it fails
- **labeling(Vs)**: Assigns truth values to the variables Vs such that all constraints are satisfied

- **sat_count(Expr, Count)**: Count the number of admissible assignments. Count is the number of different assignments of truth values to the variables in the Boolean expression Expr, such that Expr is true and all posted constraints are satisfiable

- **weighted_maximum(Weights, Vs, Maximum)**: Enumerate weighted optima over admissible assignments. Maximize a linear objective function over Boolean variables Vs with integer coefficients Weights. This predicate assigns 0 and 1 to the variables in Vs such that all stated constraints are satisfied, and Maximum is the maximum of $sum(Weight_i * V_i)$ over all admissible assignments

- **random_labeling(Seed, Vs)**: Select a single random solution. An admissible assignment of truth values to the Boolean variables in Vs is chosen in such a way that each admissible assignment is equally likely. Seed is an integer, used as the initial seed for the random number generator

3.3 Implementation

3.4 Case study

3.4.1 Design

3.4.2 Implementation

Chapter 4

Labeled Prolog

4.1 Model

4.1.1 Labeled Variables

4.1.2 Labeled Terms

4.2 Implementation

Chapter 5

CLP as Labeled Prolog

Chapter 6

Conclusions and future work

Acknowledgements

First, I would like to express my deepest gratitude to Professor Calegari and Professor Ciatto for all the support they provided me during the internship and thesis redaction processes.

Second, I would like to thank my family, my friends, my former classmates and all people who believed in me during this long study path.

Last but not least, I would like to thank myself to have been able to never give up during these two years.

Bologna, 03 February 2023 Giuseppe Boezio

Bibliography

[1] Giovanni Ciatto, Roberta Calegari, and Andrea Omicini. "2P-Kt: A logic-based ecosystem for symbolic AI". In: *SoftwareX* 16 (2021), p. 100817. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2021.100817. URL: https://www.sciencedirect.com/science/article/pii/S2352711021001126 (cit. on p. 7).

[2] W. F. Clocksin and Chris Mellish. "Programming in Prolog". In: *Springer Berlin Heidelberg*. 1987 (cit. on p. 6).

[3] Helder Coelho and Luis Moniz Pereira. "Automated reasoning in geometry theorem proving with Prolog". In: *Journal of Automated Reasoning* 2.4 (1986), pp. 329–390 (cit. on p. 3).

[4] Alain Colmerauer and Philippe Roussel. "The Birth of Prolog". In: *SIGPLAN Not.* 28.3 (Mar. 1993), pp. 37–52. ISSN: 0362-1340. DOI: 10.1145/155360.155362. URL: https://doi.org/10.1145/155360.155362 (cit. on p. 3).

[5] Jean H. Gallier. "Logic for Computer Science: Foundations of Automatic Theorem Proving". In: 1985 (cit. on p. 7).

[6] Michael Gelfond and Nicola Leone. "Logic programming and knowledge representation—the A-Prolog perspective". In: *Artificial Intelligence* 138.1-2 (2002), pp. 3–38 (cit. on p. 3).

[7] Joxan Jaffar and Michael J. Maher. "Constraint logic programming: a survey". In: *The Journal of Logic Programming* 19-20 (1994). Special Issue: Ten Years of Logic Programming, pp. 503–581. ISSN: 0743-1066. DOI: https://doi.org/10.1016/0743-1066(94)90033-7. URL: https://www.sciencedirect.com/science/article/pii/0743106694900337 (cit. on p. 9).

[8] Adam Lally and Paul Fodor. "Natural language processing with prolog in the ibm watson system". In: *The Association for Logic Programming (ALP) Newsletter* 9 (2011) (cit. on p. 3).

[9] Dennis Merritt. *Building expert systems in Prolog*. Springer Science & Business Media, 2012 (cit. on p. 3).

[10] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. "Resolving model inconsistencies using automated regression planning". In: *Software & Systems Modeling* 14.1 (2015), pp. 461–481 (cit. on p. 3).

[11] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. USA: Elsevier Science Inc., 2006. ISBN: 9780080463803 (cit. on p. 8).

[12] Markus Triska. "Boolean constraints in SWI-Prolog: A comprehensive system description". In: *Science of Computer Programming* 164 (2018). Special issue of selected papers from FLOPS 2016, pp. 98–115. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2018.02.001. URL: http://www.sciencedirect.com/science/article/pii/S0167642318300273 (cit. on p. 10).

[13] Markus Triska. "The Boolean Constraint Solver of SWI-Prolog: System Description". In: *FLOPS*. Vol. 9613. LNCS. 2016, pp. 45–61 (cit. on p. 10).

[14] Markus Triska. "The Finite Domain Constraint Solver of SWI-Prolog". In: *FLOPS*. Vol. 7294. LNCS. 2012, pp. 307–316 (cit. on p. 10).

[15] Jan Wielemaker. *SWI Prolog*. 1987. URL: https://www.swi-prolog.org/ (cit. on p. 9).