

Implementing a graph-based clause-selection strategy for Automatic Theorem Proving in Python

from the course of studies Computer Science

at the Cooperative State University Baden-Württemberg Stuttgart

by

Jannis Gehring

03/27/2025

Time frame: 09/30/2024 - 06/12/2025

Student ID, Course: 6732014, TINF22B

Supervisor at DHBW: Prof. Dr. Stephan Schulz

Declaration of Authorship

Gemäß Ziffer 1.1.13 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg vom 29.09.2017. Ich versichere hiermit, dass ich meine Arbeit mit dem Thema:

Implementing a graph-based clause-selection strategy for Automatic Theorem Proving in Python

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass alle eingereichten Fassungen übereinstimmen.

Stuttgart, 03/27/2025

Jannis Gehring

Table of Contents

1 Theory	1
1.1 First order Logic (FOL)	1
1.1.1 Basic FOL terminology	1
2 PyRes	3
2.1 Python	3
2.2 PyRes and other theorem provers	3
2.3 Architecture	4
2.4 Functionality	4
3 Specification	7
3.1 Formal description	7
3.2 Technical description	7
References	a

Code Snippets

Listing 1	The central functions for the given-clause algorithm in pyres-simple	6
Listing 2	7

List of Acronyms

FOL	First order Logic
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
TPTP	Thousands of Problems for Theorem Provers

Glossary

Exploit	An exploit is a method or piece of code that takes advantage of vulnerabilities in software, applications, networks, operating systems, or hardware, typically for malicious purposes.
Patch	A patch is data that is intended to be used to modify an existing software resource such as a program or a file, often to fix bugs and security vulnerabilities.
Vulnerability	A Vulnerability is a flaw in a computer system that weakens the overall security of the system.

1 Theory

1.1 First order Logic (FOL)

1.1.1 Basic FOL terminology

Terms are the most fundamental building block of FOL formulas. They constitute to elements of the corresponding domain D and consist of variables, functions and constants.

We assume a set $V \subset D$ of *variables*. Variables are usually denoted with the letters x, y, z, x_1, y_2, \dots , or in TPTP syntax: $X1, X2, \dots$

We also assume a set F of *function symbols*. All functions have the form $f : D^n \rightarrow D$, with n being the arity of f . Function symbols usually take the letters f, g, h, \dots . A function and its arity are denoted as $f|_n$.

Constants represent a special case of functions with arity 0 and take the letters a, b, c, \dots

Predicates are of the form $P : D^n \rightarrow \{0, 1\}$. They map (tuples of) domain elements onto truth values. The concept of function-arity is extended to predicates accordingly. They are usually denoted by P, Q, R, S, \dots

An *atom* consists of a predicate P and the corresponding input terms.

A *formula* is either an atom or the combination of atoms with logical operators ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow$) or quantors (\forall, \exists).

A *substitution* is a mapping $\sigma : V^n \rightarrow T^n$ with $n \in \mathbb{N}$ and T denoting the set of all syntactically correct terms in the problem context. It is extended from variables to terms, atoms, literals and clauses accordingly.

A *literal* is either an atom or a negated atom and usually denoted by l_1, l_2, \dots, l_n .

A *clause* C is a set of literals $\{l_1, l_2, \dots, l_n\}$. The boolean value of a clause is the disjunction of its literals truth values. Clauses are denoted by $C, D, E, \dots, C1, C2, \dots, CN$. The empty clause is denoted by \square .

1 Theory

Example: We use the problem “PUZ001-1” of the Thousands of Problems for Theorem Provers (TPTP)-dataset as an example:

- Variables $V = \{x, y\}$
- Functions $F = \{\}$
- Constants $\{a, b, c\}$ (standing for “agatha”, “butler” and “charles”)
- Predicates $\{H, K, L, R\}$ (standing for “hates”, “killed”, “loves”, “richer”)
- The set of clauses (with indexes for references):

$$\begin{aligned} & \{\{L(a)\}_1, \{L(b)\}_2, \{\neg K(x, y), \neg R(x, y)\}_3, \{\neg H(a, x), \neg H(c, x)\}_4, \\ & \{\neg H(x, a), \neg H(x, b), \neg H(x, c)\}_5, \{H(a, a)\}_6, \{H(a, c)\}_7, \{\neg K(x, y), H(x, y)\}_8, \\ & \{\neg H(a, x), H(b, x)\}_9, \{\neg L(x), R(x, a), H(b, X)\}_{10}, \{K(b, a), K(c, a)\}_{11} \end{aligned}$$

2 PyRes

PyRes is an open-source theorem prover for first-order logic. Its name originates from **P**ython, the programming language it is built with, and the **R**esolution calculus, which it implements for solving FOL problems.

2.1 Python

Python is an interpreted high-level programming language. It supports multiple programming paradigms like functional programming and object orientation. Python was created by Guido van Rossum in the early 1990s [1]. Not only is Python easy to learn and read, it also has a lot of packages like Numpy for efficient numerical computations, Pandas for manipulating big datasets, Matplotlib for plotting or TensorFlow and PyTorch for machine learning. This is why it is still among the most used programming languages.

2.2 PyRes and other theorem provers

A lot of modern theorem provers, i.e. E, Vampire and SPASS, are built with low-level languages like C and C++ ([2], [3], [4]). They employ optimized data structures and complex algorithms to increase their performance. Other provers like iprover [5] are implemented in lesser-known languages like OCaml. Both approaches make it harder for new developers to understand the codebase and functionality, hence hindering further development. This also leaves the didactic potential of theorem provers unused.

PyRes, on the other hand, is explicitly built for readability. Extensive documentation and the choice of an interpreted language enable a step-by-step understanding of the functionality. Its architecture and calculus is similar to other theorem provers, making it a suitable entry for understanding a multitude of provers. [6]

Whilst PyRes doesn't have as much extensions for optimization than other provers, this simplicity makes it a good candidate for implementing and evaluating new approaches (like alternating path theory in this case).

2.3 Architecture

PyRes is built with a layered architecture. [6]

The bottom layer is a lexical scanner. The classes `Ident` and `Token` allow storing different symbols of TPTP expressions as variables. The class `Lexer` converts strings into sequences of such tokens, allowing further inspection and processing.

The next layer consists of different classes representing FOL objects. Multiple functions implement basic terms with s-expression-like nested lists. The class `Formula` implements atoms as well as complex formulae through a tree-structure. Terms are also used by the class `Literal` to form literals, which are aggregated to clauses in `Clause`. These themselves are aggregated as sets in `ClauseSet`.

Finally, the classes `SearchParams` and `ProofState` utilize the previously mentioned classes for implementing the given-clause algorithm. Here, the `ProofState` class holds two `ClauseSets`: One for the processed and one for the unprocessed clauses.

Apart from those, there are multiple modularized components: `signature` provides an explicit signature of the formulae, `unification`, `subsumption`, `substitution`, `derivations` and `resolution` implement the corresponding FOL algorithms. `heuristics` and `indexing` provide different algorithms for optimized clause-selection during resolution.

To ensure an easy learning curve, PyRes comes in three consecutive forms:

1. **pyres-simple**, a minimal version for clausal logic.
2. **pyres-cnf**, adding heuristics, indexing and sub-sumption
3. **pyres-fof**, full support for FOL

2.4 Functionality

PyRes functions as a pipeline. First, the problem is parsed and converted to the data types specified in the previous chapter. If needed (and supported by the specified version), equality axioms are added. Then, the actual reasoning takes place.

At the heart of PyRes is the given-clause algorithm. Here, the clauses are divided into two sets, one for unprocessed and another for processed clauses. In the beginning, all

2 PyRes

clauses are unprocessed. The algorithm now iteratively selects one of the unprocessed clauses, the *given-clause*, and computes its factors as well as the resolvents between the given-clause and all processed clauses. These new clauses are now added to the unprocessed-clauses set, whilst the given-clause is moved from the unprocessed clauses to the processed clauses. The algorithm ends either if the given-clause is the empty clause (and therefore a contradiction has been found) or the unprocessed-clauses set is empty. Listing 1 shows the implementation of the given-clause-algorithm in `pyres-simple`.

If the algorithm found a contradiction, the proof is then extracted. At last, the results are printed.

```

1  def processClause(self):
2      """
3      Pick a clause from unprocessed and process it. If the empty
4      clause is found, return it. Otherwise return None.
5      """
6      given_clause = self.unprocessed.extractFirst()
7      given_clause = given_clause.freshVarCopy()
8      print("%s", given_clause)
9      if given_clause.isEmpty():
10         # We have found an explicit contradiction
11         return given_clause
12
13     new = []
14     factors = computeAllFactors(given_clause)
15     new.extend(factors)
16     resolvents = computeAllResolvents(given_clause, self.processed)
17     new.extend(resolvents)
18
19     self.processed.addClause(given_clause)
20
21     for c in new:
22         self.unprocessed.addClause(c)
23     return None
24
25  def saturate(self):
26      """
27      Main proof procedure. If the clause set is found unsatisfiable,
28      return the empty clause as a witness. Otherwise return None.
29      """
30      while self.unprocessed:
31          res = self.processClause()
32          if res != None:
33              return res
34      else:
35          return None

```

Listing 1 — The central functions for the given-clause algorithm in pyres-simple

3 Specification

This chapter serves to specify the context and requirements for the implementation. First, we will establish a formal description of the algorithm, then we will frame the technical details the algorithm will be embedded into.

3.1 Formal description

The algorithm can be defined as a function

$$R_{n,S}(S') : (S, S', n) \mapsto \{c \in S \mid d_S(S', c) \leq n\} \quad (1)$$


with S being a set of all given clauses, $S' \subseteq S$ being the set of clauses, from which the relevance distance is computed (usually containing one clause, the conjecture to prove), $n \in \mathbb{N}$ denoting the maximum relevance distance.

3.2 Technical description

```

1  if params.perform_rel_filter:
2      neg_conjs = cnf.getNegatedConjectures()
3      rel_graph = SetRelevanceGraph(cnf)
4      rel_cnf = rel_graph.get_rel_neighbourhood(neg_conjs,
        params.relevance_distance)

```

 Python

References

- [1] “Python - History and License.”
- [2] S. Schulz, “eprover.” GitHub, 2024.
- [3] “vampire.” [Online]. Available: <https://github.com/vprover/vampire>
- [4] “tspass.” [Online]. Available: <https://github.com/michel-ludwig/tspass>
- [5] “iprover.” [Online]. Available: <https://github.com/edechter/iprover>
- [6] S. Schulz and A. Pease, “Teaching Automated Theorem Proving by Example: PyRes 1.2,” in *Automated Reasoning*, N. Peltier and V. Sofronie-Stokkermans, Eds., Cham: Springer International Publishing, 2020, pp. 158–166.