DHBW
Duale Hochschule
Baden-Württemberg

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

05/17/2025

Time frame:	09/30/2024 - 06/12/2025
Student ID, Course:	6732014, TINF22B
Supervisor at DHBW:	Prof. Dr. Stephan Schulz

Declaration of Authorship

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, 05/17/2025

Jannis Gehring

Table of Contents

List of Figures

List of Tables

Code Snippets

List of Acronyms

APT	alternating path theory
CNF	Conjunctive Normal Form
FOL	First order Logic
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
TPTP	Thousands of Problems for Theorem Provers

1 Introduction

The notion of logic and reasoning has been fundamental to human knowledge acquisition for thousands of years. Even Aristotle stated “All knowledge should be subject to examination and reason.”.

2 Theory

2.1 Propositional logic

2.1.1 Syntax of propositional logic

Propositional logic deals with atomic boolean variables (usually denoted with letters $A, B, C, ...$ or $p, q, r, ...$).

They can be negated ($\neg$) or combined with different junctors ($\wedge$ denoting *and*, $\vee$ denoting *or*) to form complex formulae.
The notion that A *implies* B is defined as $\neg A \vee B$ and noted as $A \rightarrow B$.
Logical *equivalence* is defined as $(A \rightarrow B) \wedge (B \rightarrow A)$ and noted with $A \leftrightarrow B$.
The ranking of operators is $\neg \gg \wedge \gg \vee \gg \rightarrow \gg \leftrightarrow$; where necessary, braces have to be added.

Example: Propositional logic syntax

Suppose a set of variables $\{A, B, C, D\}$. Then A, $B \wedge D$ and $((\neg\neg B \leftrightarrow D \wedge C) \vee 0) \rightarrow C$ are valid propositional logic formulae, whilst $\neg$ and $B \wedge \vee D$ are not.

2.1.2 Semantic of propositional logic

Boolean variables can take the values 0, which is interpreted as *false*, and 1, being interpreted as *true*. The semantics of logical operators can be stated with a simple truth table [1]:

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \to B$	$A \leftrightarrow B$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Table 1 — Truth table for basic operators in propositional logic.

An *interpretation* $\mathcal{I} : \{A, B, C, ...\} \to \{0, 1\}$ is a function mapping a (finite) set of variables onto boolean values. An interpretation that makes a formula true is called a *model* of that formula. A formula is called *satisfiable*, if there exists a model for it, and a *tautology* if all interpretations are models. Likewise, a formula is called *falsifiable*, if there exists an interpretation that is not a model and *unsatisfiable*, if no interpretation is a model.

Example: Propositional logic semantic.

Suppose - a set of variables $\{A, B, C, D\}$ and - an interpretation $\mathcal{I} = \{A \mapsto 1, B \mapsto 0, C \mapsto 0, D \mapsto 1\}$.

Then the formulae
$\alpha = D \vee 1$, $\beta = \neg A \to B$ and $\gamma = \neg C \wedge \neg D \leftrightarrow \neg(C \vee D)$
evaluate to be true under $\mathcal{I}$ whilst
$\delta = 0$, $\varepsilon = C \to B$ and $\zeta = B \vee C \wedge (A \leftrightarrow D)$
evaluate to be false under $\mathcal{I}$.

Note that the formulae α, β, γ, ε and ζ are satisfiable, α and γ even being tautologies. The formulae β, δ, ε and ζ are falsifiable, δ is unsatisfiable.

2.1.3 Resolution in propositional logic

One natural question to ask when dealing with formulae is whether one or more formulae *imply* another formula. One method of solving this problem is with *binary resolution*, formulated by J. A. Robinson in 1965 [2].

If a formula is a conjunction of disjunctions and negations appear only directly before variables it is in Conjunctive Normal Form (CNF). A formula $\alpha = (A \vee \neg B) \wedge ... \wedge (\neg C \vee A)$ in CNF can also be written as a set of *clauses* $M_\alpha = \{\{A, \neg B\}, ..., \{\neg C, A\}\}$, where the elements of clauses are called *literals*. The empty clause is denoted by $\square$.

Now suppose the formula [3]

$$\begin{aligned} \alpha &= (p \vee q \vee \neg r) \wedge (r \vee \neg s) \\ M_\alpha &= \{\{p, q, \neg r\}, \{r, \neg s\}\}. \end{aligned} \tag{1}$$

Is α satisfiable? For α to be satisfiable, there has to exist a model of α, and for an interpretation of α to be a model, it has to make both clauses in M_α true. The first clause gets true either if $p^\mathcal{J} = 1$ or $q^\mathcal{J} = 0$, the second is true if $s^\mathcal{J} = 0$. r has no effect on the truth-value of the formula, because it appears once positive and once negative; the value of one literal always "canceles out" the value of the other one. Out of these observations, we construct a new clause $\{p, q, \neg s\}$ and add it to our set of clauses. Satisfiability of our previous set M_α is now equivalent to satisfiability of our newly constructed set $M_\alpha{}'$:

$$M_\alpha \equiv \mathrm{Res}(M_\alpha) = M_\alpha{}' = \{\{p, q, \neg r\}, \{r, \neg s\}, \{p, q, \neg s\}\}. \tag{2}$$

If resolution can be applied again, one abbreviates $\mathrm{Res}(\mathrm{Res}(M_\alpha))$ with $\mathrm{Res}^2(M_\alpha)$, $\mathrm{Res}(\mathrm{Res}(\mathrm{Res}(M_\alpha)))$ with $\mathrm{Res}^3(M_\alpha)$ and so on. $\mathrm{Res}^*(M_\alpha)$ denotes the "final" set of clauses for which no more resolution can be applied. If it contains the empty clause, there are no models of $\mathrm{Res}^*(M_\alpha)$, and because $\mathrm{Res}^*(M_\alpha) \equiv M_\alpha$ there are also no models of M_α, which in turn means, that M_α is shown to be unsatisfiable.

In general, the central inferencing rule of resolution [3] can be stated as follows:

$$\begin{gathered} \frac{p_1 \vee ... \vee p_m \vee \mathrm{r} \quad q_1 \vee ... \vee q_m \vee \neg\, \mathrm{r}}{p_1 \vee ... \vee p_m \vee q_1 \vee ... \vee q_m}, \\ \text{or in clause notation:} \\ \frac{\{p_1, ..., p_m, r\} \quad \{q_1, ..., q_m, \neg r\}}{\{p_1, ..., p_m, q_1, ..., q_m\}}. \end{gathered} \tag{3}$$

So how can this be obtained to proof that a set of formulae implies another formula? Suppose one wants to proof that α, β and γ imply δ with:

- $\alpha = \neg q \vee r$
- $\beta = p \vee \neg r$
- $\gamma = \neg q \vee \neg p$
- $\delta = q$

$$(\neg q \vee r) \wedge (p \vee \neg r) \wedge (\neg q \vee \neg p) \vDash q \tag{4}$$

A formula α implies another formula β, if all models of α are also models of β. In this case, one writes $\alpha \vDash \beta$.

2.2 First order Logic

First-order-logic is an extension of propositional logic.

Terms are the most fundamental building block of FOL formulas. They correspond 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 $x, y, z, x_1, y_2, ...$, or in TPTP syntax: $X1, X2, ...$
We also assume a set F of *function symbols*. All functions have the form $f : D^n \to D$, with n being the arity of f. Function symbols usually take the letters $f, g, h, ...$ A function symbol and its arity are denoted as $f|_n$.
Constants represent a special case of functions with arity 0 and take the letters $a, b, c, ...$

Predicates are of the form $P : D^n \to \{0, 1\}$. They map (tuples of) domain elements onto truth values. The concept of function-arity is extended to predicates accordingly. They are usually denoted by $P, Q, R, S, ...$
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 ($\neg, \wedge, \vee, \to, \leftrightarrow$) or quantors ($\forall, \exists$).
A *substitution* is a mapping $\sigma : V^n \to T^n$ 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 $l_1, l_2, ...l_n$.
A *clause* C is a set of literals $\{l_1, l_2, ..., l_n\}$. The boolean value of a clause is the disjunction of its literals truth values. All variables in those literals are implicitly universally quantified. Clauses are denoted by $C, D, E, ..., C1, C2, ..., CN$. The empty clause is denoted by $\square$.

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", "lives", "richer")
- The set of clauses (with indexes for references):

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

2.3 Clause selection

Clause selection is the problem of identifying and selecting those clauses of a logical problem, that are necessary and sufficient for a full proof. On the one hand, failing to identify necessary clauses prevents a successfull proof, on the other hand, selecting irrelevant clauses can slow down the proof algorithm.

In general, one has to differentiate between two types of clause selection: clause selection *before* saturation and clause selection *during* saturation. While the first one functions like a filter in the prover-pipeline, the second one serves as a heuristic for finding proofs faster. In a worst case scenario, the first one can prevent a successful output by removing important clauses, whilst the second can only delay the consideration of a needed clause. Oftentimes, the same principle idea can be applied to implement both kinds of selections. This coursework focuses on the first kind of selection (for the full pipeline, see Figure 2). For that, there are different approaches:

Meng and Paulson [4] introduced an approach based on the sharing of symbols. They essentially computed a score for each clause (i.e. the number of relevant symbols divided by the total number of symbols) and accepted all clauses, whose score exceeds some pass mark $0 < p < 1$. All symbols of accepted clauses are then regarded as relevant and the procedure repeats iteratively. The passmark is increasing (and therefore getting stricter) every iteration with the formula $p_{i+1} = p_i + \frac{1-p_i}{c}$, c being a parameter for convergence. They found $p = 0.6$ and $c = 2.4$ to be effective. Although the approach is farely simply, it increased the number of problems solved for a given time limit for E [5], SPASS [6] and Vampire [7].

Pudlak [8] introduced an approach where relevance is computed using finite models. In his algorithm, he starts with M_0 being a model of $\{\neg C_{conj}\}$ (C_{conj} being the conjecture). Then, a premise C_0 avoiding M_0 is selected, and a new model $M_1 \vDash \{\neg C_{conj}, C_0\}$ is constructed. This procedure is repeated until no model can be found. The set of premises $\{C_0, C_1, ..., C_n\}$ is now treated as a candidate for proving the theory. This idea was implemented in SPASS [6] by Sutcliffe and Puzis. The algorithm is able to reuse interpretations in different proofs. It also does not become ineffective concerning

memory when proofs take more time, a problem other provers suffer from. On the other hand, the number of computed interpretations can get really high, making it ineffective for problems with large numbers of premises.

2.4 Clause selection with APT

2.4.1 Terminology

For describing alternating path theory (APT), the following terminology is introduced. It is based on Plaisted's work on alternating paths [9].

The relation $\equiv$ denotes syntactic identity, meaning $A \equiv A$, $\neg A \equiv \neg A$, $A \equiv \neg\neg A$, $\neg\neg A \equiv A$ for all atoms A.

Two literals L and M are *complementary unifiable* if there are substitutions σ and τ so that $\sigma(L) \equiv \neg\tau(M)$. This leads to the central definition of APT:

Definition 2.4.1.1 (Alternating Path):
An *alternating path* in a set of clauses S from C_1 to C_n is a sequence

$$C_1, p_1, C_2, p_2, ..., C_{n-1}, p_{n-1}, C_n \tag{5}$$

with

- $C_i \in S$ for all i,
- $p_i = (L_i, M_{i+1})$ being a tuple of literals with $L_i \in C_i$ and $M_{i+1} \in C_{i+1}$,
- L_i and M_{i+1} being complementary unifiable and
- $L_i \not\equiv M_i$.

The name "alternating" comes from the notion, that the path alternates between connecting two clauses through the unifiability of its' literals and switching literals inside a clause. Oftentimes one omits the p_i when denoting an alternating path.

In this coursework and the implementation, the *length* of an alternating path is equal to the number of clauses (including the start) minus one. Note that Plaisted omitted

this subtraction of one in his work [9]. Differing from his definition seemed more intuitive, because then, the conjecture clauses have an alternating path distance of zero to themselves. This length is roughly analogue to the concept of a norm in a vector space, leading to a metric-like definition for clauses in S:

Definition 2.4.1.2 (Relevance distance):
The *relevance distance* d_S is defined

1. between clauses $\{C_1, C_2\} \subseteq S$ as the length of the shortest path between those. If there is no alternating path between C_1 and C_2, their distance to one another is ∞.
2. between a subset $T \in S$ and any clause $C \in S$ as the shortest path from a clause in T to C:

$$d_S(T, C) = \min\{d_S(D, C) : D \in T\} \tag{6}$$

If $d_S(C_1, C_2) \neq \infty$, C_1 and C_2 are *relevance connected* in S. A set of clauses $S' \subseteq S$ is relevance connected, if every pair of two clauses in S is relevance connected.

Definition 2.4.1.3 (Relevance neighbourhood):
The *relevance neighbourhood* from $T \subseteq S$ regarding the relevance distance n is defined as

$$R_{n,S}(T) = \{C \in S : d_S(T, C) \leq n\} \tag{7}$$

The last definition required for formulating the central theorem of clause-selection using APT is that of a *set of support*:

Definition 2.4.1.4 (Set of Support):
Let S be a unsatisfiable set of clauses. $S' \subseteq S$ is called a set of support for S, if it shares at least one clause with every unsatisfiable subset of S.

2.4.2 Functionality

Using this terminology, Plaisted [9] concludes the following theorem:

Theorem 2.4.2.1:
Let S be an unsatisfiable set of clauses. If

- $S' \subseteq S$ is a support set for S,
- there is a length n refutation from S and
- $m \geq 2n - 2$,

then $R_{m,S}(S')$ is unsatisfiable.

Flipping this on its head leads to the following theorem proving method:

For a given m, compute $R_{m,S}(S')$ and test $R_{m,S}(S')$ for satisfiability.

There are two practical problems to this approach: On the one hand, the computation of $R_{m,S}(S')$ isn't guaranteed to terminate; on the other hand, unsatisfiability of $R_{m,S}(S')$ proves unsatisfiability for S, but satisfiability for $R_{m,S}(S')$ does not prove satisfiability for S.

Plaisted solves both problems by defining an algorithm where all values of m are tried in parallel, interleaving the computations. If one concludes unsatisfiability, S is proven to be unsatisfiable. [9]

For the implmenetation in PyRes, m is going to be set manually by the user via command line (see also Section 4.2). If $R_{m,S}(S')$ is found to be satisfiable, but there

were clauses missing during saturation due to clauses selection with APT, PyRes doesn't return with "Satisfiable", but with "GaveUp", to indicate this uncertainty.

3 PyRes

PyRes is an open-source theorem prover for first-order logic. Its name originates from **Py**thon, the programming language it is built with, and the **Res**olution calculus, which it implements for solving First order Logic (FOL) problems.

3.1 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 [10]. 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 among the most used programming languages.

3.2 PyRes and other theorem provers

A lot of modern theorem provers, i.e. E [5], Vampire [7] and SPASS [6], are built with low-level languages like C and C++ ([11], [12], [13]). They employ optimized data structures and complex algorithms to increase their performance. Other provers like iprover [14] are implemented in lesser-known languages like OCaml. While those languages ensure soundness and efficiency, both approaches make it hard for new developers to understand the codebase and functionality, hence hindering further developement. 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. [15]

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).

3.3 Architecture

PyRes is built with a layered architecture. [15]

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 to implement the given-clause algorithm. Here, the `ProofState` class holds two `ClauseSet`s: 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 increasingly more complex versions:

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 with equality.

3.4 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 problem is being clausified to CNF, before the actual reasoning takes place.

At the heart of PyRes is the given-clause algorithm [15]. Here, the clauses are divided into two sets, one for unprocessed and another for processed clauses. In the beginning, all 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 set of unprocessed clauses, 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 set of unprocessed clauses 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. Figure 1 illustrates this pipeline as a flow-chart.

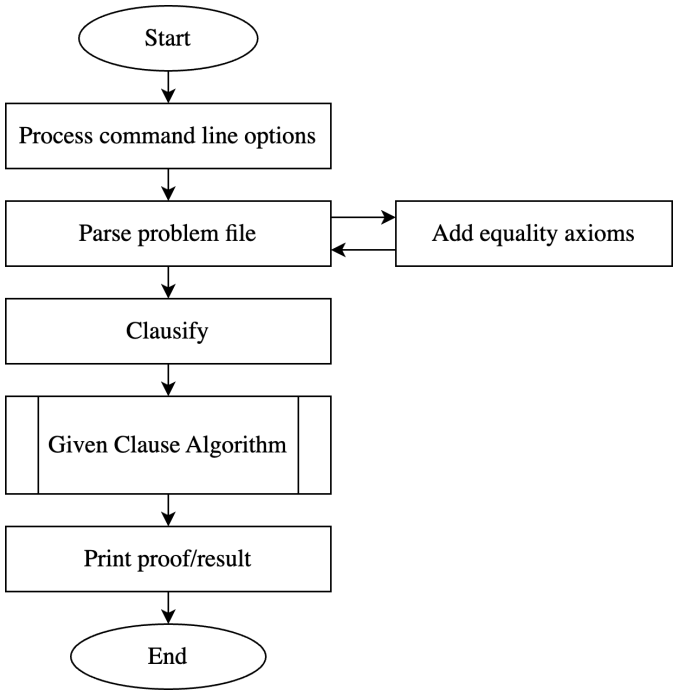

Figure 1 — Simple pipeline of PyRes's functionality.

```python
def processClause(self):
    """
    Pick a clause from unprocessed and process it. If the empty
    clause is found, return it. Otherwise return None.
    """
    given_clause = self.unprocessed.extractFirst()
    given_clause = given_clause.freshVarCopy()
    print("%", given_clause)
    if given_clause.isEmpty():
        # We have found an explicit contradiction
        return given_clause

    new = []
    factors    = computeAllFactors(given_clause)
    new.extend(factors)
    resolvents = computeAllResolvents(given_clause, self.processed)
    new.extend(resolvents)

    self.processed.addClause(given_clause)

    for c in new:
        self.unprocessed.addClause(c)
    return None

def saturate(self):
    """
    Main proof procedure. If the clause set is found unsatisfiable,
    return the empty clause as a witness. Otherwise return None.
    """
    while self.unprocessed:
        res = self.processClause()
        if res != None:
            return res
    else:
        return None
```

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

4 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.

4.1 Formal specification

The algorithm is should calculate the function

$$R_{n,S}(S') : (S, S', n) \mapsto \{c \in S \mid d_S(S', c) \leq n\} \tag{8}$$

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 and $d_S(S', c)$ being the minimal distance of S' to the clause c.

4.2 Technical specification

The implementation of APT functions as a filter preceding the actual solving algorithm. Therefore, the only changes made to existing PyRes steps are the command line specification and the output of the result.

Whilst the set of all clauses S and the set of conjectures S' are defined by the problem file, the relevance distance n can be specified with the command line argument `--relevance-distance/-r`. If a relevance distance is specified, clause selection with APT is performed before saturation. Figure 2 illustrates the new pipeline:

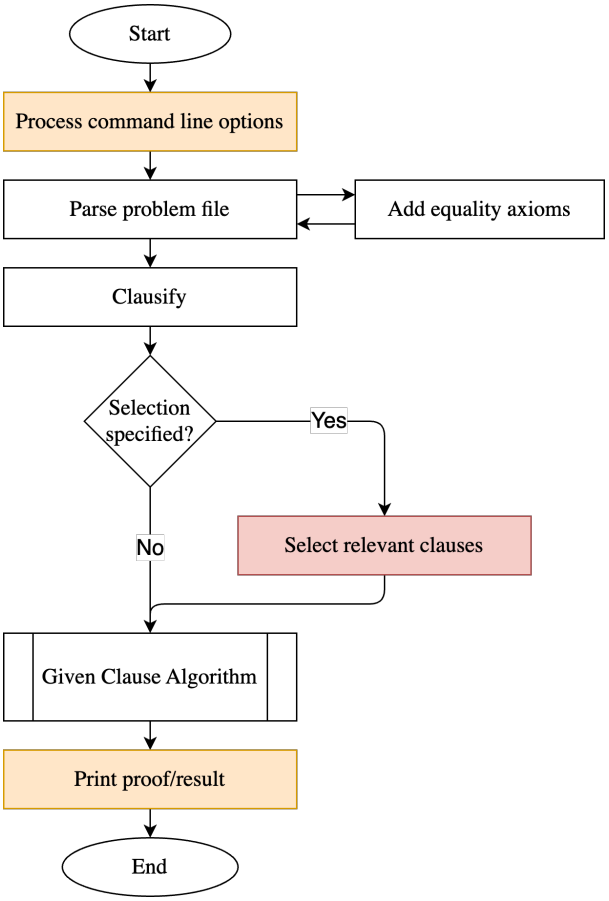

Figure 2 — PyRes pipeline with optional clause-selection step.
Orange denotes changed, red new steps.

Independent of implementation, the process of selecting relevant clauses features two steps: Firstly, the graph has to be constructed; secondly, the relevance neighbourhood

of the negated conjectures has to be selected. Each step is performed by a seperate Python function.

To provide a unified interface for different implementations of APT, both steps are aggregated in an abstract base class named `RelevanceGraph`. Each concrete implementation is created as a child class of `RelevanceGraph`, supplying a function body for both `construct_graph` and `get_rel_neighbourhood`.

Listing 2 contains the definition of the `RelevanceGraph` abstract base class, Listing 3 contains the actual code for clause selection in the PyRes main file.

```python
from abc import ABC, abstractmethod
from clausesets import ClauseSet

class RelevanceGraph(ABC):

    def __init__(self, clause_set: ClauseSet):
        self.construct_graph(clause_set)

    @abstractmethod
    def construct_graph(self, clause_set: ClauseSet):
        pass

    @abstractmethod
    def get_rel_neighbourhood(self, from_clauses: ClauseSet, distance:
    int):
        pass
```

Listing 2 — Base Class for different implementations: `RelevanceGraph`

```python
if params.perform_rel_filter:
    neg_conjs = cnf.getNegatedConjectures()
    rel_graph = SetRelevanceGraph(cnf)
    rel_cnf = rel_graph\
        .get_rel_neighbourhood(neg_conjs, params.relevance_distance)
```

Listing 3 — Main steps of performing clause selection, independent of implementation. `SetRelevanceGraph` is substituted with the implementations' class name.

5 Implementation

5.1 Universal Set Approach

5.1.1 Data structures

With this approach, the data structures have different layers:

The most basic data structures are the classes `Node` and `Edge`. Each node contains a clause and one literal of that clause, clauses and literals are implemented with the already available, corresponding classes. The distinction between nodes by which the clause is entered end exited is made implicitly by aggregating them in sets (more on that later). An edge simply consists of two nodes.

For both classes, a string representation for printing and debugging has been implemented.

```python
class Node:
    def __init__(self, literal: Literal, clause: Clause) -> None:
        self.literal: Literal = literal
        self.clause: Clause = clause

    def __repr__(self) -> str:
        return f"<{self.clause.name},{self.literal}>"


class Edge:
    def __init__(self, node1: Node, node2: Node) -> None:
        self.node1: Node = node1
        self.node2: Node = node2

    def __repr__(self) -> str:
        return f"Edge: {self.node1} - {self.node2}"
```

Listing 4 — Implementation of nodes and edges in the universal set approach.

The next layer aggregates those classes into multiple sets: For algorithmic simplicity, two sets of nodes are created: one refers to all nodes by which a clause is entered (`in_nodes`), the other refers to all nodes by which a clause is exited (`out_nodes`). Edges are aggregated in a single set (hence *universal* set approach).

5.1.2 Graph construction algorithm

Nodes are constructed by iterating over all clauses and every literal of those clauses and creating one node for `in_nodes` and one for `out_nodes`. This algorithm is therefore $\mathcal{O}(|S| \cdot |L|)$, with L denoting the set of all literals in clauses in S.

Edges between nodes of the same clause are constructed by checking, for each combination of an `in_node` with an `out_node`, whether their clauses are equal *and* their literals are unequal. This case corresponds to the switching of literals of the same clause in an alternating path; an edge is created.

Edges between nodes of different clauses correspond to the potential resolution between those. Therefore, for each combination of an `in_node` with an `out_node`, their literals have to be evaluated. For the resolution calculus to apply, the literal's signs have to be different. Also, there has to exist a possible unifier for both literals' atoms, which is checked with the already available `mgu` function of `unification.py`.

Both edges of same and different clauses take $\mathcal{O}(|S| \cdot |L|)$ time to compute.

5.1.3 Neighbourhood computation algorithm

The relevance neighbourhood is computed as follows:

First, the set of conjectures is mapped to the corresponding set of nodes.

```python
def get_rel_neighbourhood(self, from_clauses: ClauseSet,
distance: int):

    neighbourhood = self.clauses_to_nodes(from_clauses)
    for _ in range(2 * distance - 1):
        new_neighbours = self.get_neighbours(neighbourhood)
        neighbourhood |= new_neighbours

    clauses = self.nodes_to_clauses(neighbourhood)
    return clauses

def get_neighbours(self, subset: set[Node]):
    neighbouring_edges = {
        edge for edge in self.edges
        if self.edge_neighb_of_subset(edge, subset)
    }
    self.edges -= neighbouring_edges
    neighbouring_nodes = {
        edge.node2 if edge.node1 in subset else edge.node1
        for edge in neighbouring_edges
    }
    return neighbouring_nodes

@staticmethod
def edge_neighb_of_subset(edge: Edge, subset: set[Node]):
    return (edge.node1 in subset) != (edge.node2 in subset)
```

Listing 5 —

5.2 Adjacency Set Approach

6 Evaluation

6.1 Experimental setup

References

[1] J. Barwise, "An Introduction to First-Order Logic," *HANDBOOK OF MATHEMATICAL LOGIC*, vol. 90. in Studies in Logic and the Foundations of Mathematics, vol. 90. Elsevier, pp. 5–46, 1977. doi: https://doi.org/10.1016/S0049-237X(08)71097-8.

[2] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *J. ACM*, vol. 12, no. 1, pp. 23–41, Jan. 1965, doi: 10.1145/321250.321253.

[3] K.-U. Witt, "Mathematische Grundlagen der Informatik," *Mengen, Logik, Rekursion*. Springer Vieweg Wiesbaden, 2013. doi: https://doi.org/10.1007/978-3-658-03079-7.

[4] J. Meng and L. C. Paulson, "Lightweight relevance filtering for machine-generated resolution problems," *Journal of Applied Logic*, vol. 7, no. 1, pp. 41–57, 2009, doi: https://doi.org/10.1016/j.jal.2007.07.004.

[5] S. Schulz, "E - a brainiac theorem prover," *AI Commun.*, vol. 15, no. 2, 3, pp. 111–126, Aug. 2002, [Online]. Available: https://dl.acm.org/doi/abs/10.5555/1218615.1218621

[6] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski, "SPASS Version 3.5," in *Automated Deduction – CADE-22*, R. A. Schmidt, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 140–145. doi: 10.1007/978-3-642-02959-2_10.

[7] L. Kovács and A. Voronkov, "First-Order Theorem Proving and Vampire," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–35. doi: 10.1007/978-3-642-39799-8_1.

[8] P. Pudlák, "Semantic Selection of Premisses for Automated Theorem Proving.," *ESARLT*, vol. 257, 2007.

References

[9] D. A. Plaisted, "Properties and Extensions of Alternating Path Relevance - I." [Online]. Available: https://arxiv.org/abs/1905.08842

[10] "Python - History and License."

[11] S. Schulz, "Eprover GitHub repository." [Online]. Available: https://github.com/eprover/eprover

[12] "Vampire GitHub repository." [Online]. Available: https://github.com/vprover/vampire

[13] "TSPASS GitHub repository." [Online]. Available: https://github.com/michel-ludwig/tspass

[14] "iProver GitHub repository." [Online]. Available: https://github.com/edechter/iprover

[15] 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.