Concordia University

Engineering and Computer Science

COMP 6591 - Knowledge base systems

Project Report - August 2023

Design and implementation of the SLD Resolution Method

Group ID: 19

Team members

Akshaya Barat Bushan - 40220139

Marium Ali - 40226329

Ramana Koduri - 40217265

Ashish Reddy Jaddu - 40217896

Table of Contents

1. Introduction

Identifying the logical correctness of assertions is one of the major difficulties in the field of artificial intelligence. First-Order Logic (FOL) is a fundamental tool for reasoning and inference because it plays a crucial role in describing complex relationships and propositions. With the ultimate goal of improving our understanding of automated reasoning systems, our project, "Design and Implementation of the SLD Resolution Method for First-Order Logic Inference," tackles the task of determining whether a query can be logically inferred from a given set of closed formulas that are implemented in Python.

In the area of automated reasoning, the SLD (Selective Linear Definite) Resolution Method is a potent method. It provides a methodical way to determine whether a query is valid inside a certain set of formulas. The goal of this project is to develop an automated logical reasoning system that can answer the question "Does the set of formulas entail the given query?" by combining theoretical notions with real-world application.

2. Objective

The core objective of our project is to develop a software tool that employs the SLD Resolution Method to determine the logical implication relationship between a set of closed formulas and a query in the First-Order Logic. By combining the principles of resolution-based reasoning with efficient algorithmic implementations, we aim to facilitate the validation of logical relationships in a wide range of scenarios.

3. Methodology

The theorem prover has been developed in Python and is responsible for managing and processing rules and queries through a queue-based algorithm, as well as utilizing data structures and regular expressions. It employs unification techniques to handle variable substitutions and incorporates efficient clause resolution algorithms.

```
Start
|
|-> Input: Number of Queries (n2)
|      Collect Queries
|
|-> Input: Number of Rules (k)
|      Collect Rules
|
|-> Create KB (Knowledge Base)
|      Setup KB
|
|-> For each Query:
|      |
|      |-> Negate Query (if necessary)
|      |      |
|      |      |-> Resolve with KB
|      |             |
|      |             |-> Initialize Queue
|      |             |
|      |             |-> Iterate through Clauses:
|      |                    |
|      |                    |-> Unify Clauses
|      |                    |
|      |                    |-> Apply Resolution Logic
|      |                    |
|      |                    |-> Update KB with New Clauses
|      |
|      |-> Output: Resolvability of Query
|
|-> End
```

3.1 Negating the Goal

In this step, the goal clause is negated to transform the proving task into a refutation task. The negated goal will be resolved with a set of definite clauses to check for contradictions.

3.2 Conversion to CNF

Converting the clauses into Conjunctive Normal Form (CNF) is essential for uniform resolution. CNF ensures that each clause is a conjunction of disjunctions, making it suitable for resolution.

In order to convert a given logical sentence into Conjunctive Normal Form (CNF), we use a function called CNF. It is expected that the input sentence has a "premise => conclusion" structure. We are aware that pq is equal to pq. As a result, the code begins by utilising the "=>" symbol to separate the input statement into its premise and conclusion components. The "&" mark is then used to further separate the premise into its component parts. The code handles negations differently for each of these components: if a component begins with a negation ("~"), the negation is deleted; otherwise, a negativity is added.

In order to produce a disjunction of literals and the temp2 variable, the modified components are then merged using the "|" symbols. The modified expression in CNF is created by using the "|" separator to combine temp2 and the conclusion section.

3.3 Unification

The unification algorithm compares two lists of arguments, arglist1 and arglist2, element by element. It tries to find a consistent set of substitutions for variables within these lists, such that the lists become equal. The algorithm handles different cases based on the types of elements being compared,

- If both elements are variables and not equal, return an empty list since unification is not possible.
- If both elements are equal variables, add a substitution to the theta dictionary.
- If the first element is a variable and the second is not, substitute the variable in the second list and add a substitution to theta.
- If the first element is not a variable and the second is a variable, substitute the variable in the first list and add a substitution to theta.
- If both elements are not variables, substitute the variable in the first list, and if necessary, substitute the variable in the second list. Add substitutions to theta as needed.

The theta dictionary maintains the variable substitutions and the algorithm updates the argument lists as necessary during the process. The result of the unification is a list containing the updated argument lists and the theta dictionary, which represents the consistent variable substitutions that make the two argument lists equivalent.

3.4 Resolution

The system iteratively resolves queries using knowledge base rules according to the resolution principle. It finds complementary clause pairs, performs unification, and produces new clauses. This process keeps going until either a preset limit is reached or a resolution results in an empty phrase (which denotes a contradiction).

3.4.1 Query Processing and Negation

- Starting with a set of queries that need to be answered, the algorithm loops over them.
- If a query hasn't previously been negated, it does so for each one. The algorithm, for instance, will negate the query "P(x) | Q(y)" to "P(x) & Q(y)". They are divided into separate rules for each "&" in the sentence.

3.4.2 Clause Selection and Unification

- The algorithm creates a queue to hold clauses for processing.

- It selects clauses from the knowledge base that can potentially be resolved with the negated query. These selected clauses are pushed into the queue.
- The algorithm then enters a loop where it dequeues clauses and processes them.
- For each clause, it extracts the predicate and arguments and then searches the knowledge base for matching rules (clauses) that can be unified with the current clause.
- Unification is performed to find substitution bindings for the variables in the query and the selected rule. This helps align the arguments and variables.

3.4.3 Resolution Process

- The algorithm aims to resolve the negated query with the selected clauses using the computed substitution bindings.
- It iterates through each clause in the selected rule (if there's more than one clause) and performs resolution steps.
- The resolution process involves merging the selected clause and the negated query while keeping track of substitutions made during unification.
- The algorithm continues to merge and simplify clauses using the substitutions until either a resolution is achieved (which indicates that the query is true) or no further resolutions can be made.

3.4.4 Output Determination

- The algorithm monitors the process of resolution. It concludes that the query is accurate in the knowledge base if it successfully merges clauses to an empty clause (signifying a resolution).
- The process ends after concluding that the query is false in the knowledge base if no resolution is made and the queue is empty.

3.4.5 Efficiency and Termination

- The algorithm has termination and efficiency techniques.
- In order to prevent infinite loops or computations that take too long, it places a cap on the number of iterations and a time restriction on each query.

- The method stops and treats the question as unresolved if the resolution procedure takes too long or the iteration limit is reached.

3.4.6 Results and Reporting

- The algorithm keeps track of each query's resolution in the knowledge base as true or false.
- The resolution() function ends with the return of a list containing the outcomes.

4. Experimental Results

We outline the experimental findings from using the SLD resolution technique for query responding in this section. Our test's primary objective was to assess the effectiveness and efficiency of the resolution process in figuring out whether certain queries inside a knowledge base might be satisfied.

4.1 Experimental Setup

We processed a set of queries against an established knowledge base using a customized implementation of the SLD resolution algorithm, which follows a set of support and unit resolutions. The code, which was developed in Python, had a number of techniques for clause selection, query negation, unification, and resolution. The tests were carried out on a standard laptop with 8 GB RAM and a 2.4 GHz processor.

4.2 Results

We put the SLD resolution algorithm through its paces on a collection of queries of varying complexity. The algorithm delivered the following test outcomes,

Query Satisfiability: The algorithm was successful in determining whether the questions were satisfiable. According to the specified knowledge base, the resolutions showed whether the given queries were true or false.

Efficiency: For comparatively simple queries to complex queries, the algorithm performed efficiently. The fact that the resolution process ended in a timely manner

shows that the algorithm was successful in identifying resolutions or determining unsatisfiability.

Complex Queries: The algorithm successfully conducted stages for unification and resolution when dealing with complex queries that contained many clauses and predicates. However, the algorithm ran into resolution challenges for some complex queries, increasing the execution duration.

4.3 Execution Results:

Input 1:

- **Facts and Rules**

  King(x) & Greedy(x) => Evil(x)
  King(x)
  Greedy(x)

- **Query/ Goal:**

  Evil(John)

Output 1:

```
ashish/Desktop/Logic-Programming-Resolution-master/Final.py
Enter number of queries: 1
Enter 1 queries
Evil(John)
Enter number of rules: 3
Enter 3 rules
King(x) & Greedy(x) => Evil(x)
King(x)
Greedy(x)
***** Resolution Start *****
Query after negation=> ~Evil(John)
KB=>  {'~King': {1: [['~King(x)|~Greedy(x)|Evil(x)', ['~King(x)', '~Greedy(x)', 'Evil(x)'], 0,
 ['x']]]}, '~Greedy': {1: [['~King(x)|~Greedy(x)|Evil(x)', ['~King(x)', '~Greedy(x)', 'Evil(x)
'], 1, ['x']]]}, 'Evil': {1: [['~King(x)|~Greedy(x)|Evil(x)', ['~King(x)', '~Greedy(x)', 'Evil
(x)'], 2, ['x']]]}, 'King': {1: [['King(x)', ['King(x)'], 0, ['x']]]}, 'Greedy': {1: [['Greedy
(x)', ['Greedy(x)'], 0, ['x']]]}}




Is the query resolvable: TRUE
```

Input 2:

- **Facts and Rules**

  P(x) => P(f(x))

- **Query/ Goal:**

  P(A)

Output 2:

```
ashish/Desktop/Logic-Programming-Resolution-master/Final.py
Enter number of queries: 1
Enter 1 queries
P(A)
Enter number of rules: 1
Enter 1 rules
P(x) => P(f(x))
***** Resolution Start *****
Query after negation=> ~P(A)
KB=>  {'~P': {1: [['~P(x)|P(f(x))', ['~P(x)', 'P(f(x))'], 0, ['x']]]}, 'P': {2: [['~P(x)|P(f(x
))', ['~P(x)', 'P(f(x))'], 1, ['f', 'x)']]]}}
Is the query resolvable: FALSE
```

Input 3:

- **Facts and Rules**

  Buffalo(x) & Pig(y) => Faster(x, y)
  Pig(x) & Slug(y) => Faster(x, y)
  Faster(x, y) & Faster(y, z) => Faster(x, z)
  Buffalo(Bob)
  Pig(Pat)
  Slug(Steve)

- **Query/ Goal:**

  Faster(Pat, Steve)

Output 3:

```
ashish/Desktop/Logic-Programming-Resolution-master/Final.py
Enter number of queries: 1
Enter 1 queries
Faster(Pat, Steve)
Enter number of rules: 5
Enter 5 rules
Buffalo(x) & Pig(y) => Faster(x, y)
Pig(x) & Slug(y) => Faster(x, y)
Faster(x, y) & Faster(y, z) => Faster(x, z)
Buffalo(Bob)
Pig(Pat)
Slug(Steve)***** Resolution Start *****
Query after negation=> ~Faster(Pat,Steve)
KB=  {'~Buffalo': {1: [['~Buffalo(x)|~Pig(y)|Faster(x,y)', ['~Buffalo(x)', '~Pig(y)', 'Faster
(x,y)'], 0, ['x']]]}, '~Pig': {1: [['~Buffalo(x)|~Pig(y)|Faster(x,y)', ['~Buffalo(x)', '~Pig(y
)', 'Faster(x,y)'], 1, ['y']], ['~Pig(x)|~Slug(y)|Faster(x,y)', ['~Pig(x)', '~Slug(y)', 'Faste
r(x,y)'], 0, ['x']]]}, 'Faster': {2: [['~Buffalo(x)|~Pig(y)|Faster(x,y)', ['~Buffalo(x)', '~Pi
g(y)', 'Faster(x,y)'], 2, ['x', 'y']], ['~Pig(x)|~Slug(y)|Faster(x,y)', ['~Pig(x)', '~Slug(y)'
, 'Faster(x,y)'], 2, ['x', 'y']], ['~Faster(x,y)|~Faster(y,z)|Faster(x,z)', ['~Faster(x,y)', '
~Faster(y,z)', 'Faster(x,z)'], 2, ['x', 'z']]]}, '~Slug': {1: [['~Pig(x)|~Slug(y)|Faster(x,y)'
, ['~Pig(x)', '~Slug(y)', 'Faster(x,y)'], 1, ['y']]]}, '~Faster': {2: [['~Faster(x,y)|~Faster(
y,z)|Faster(x,z)', ['~Faster(x,y)', '~Faster(y,z)', 'Faster(x,z)'], 0, ['x', 'y']], ['~Faster(
x,y)|~Faster(y,z)|Faster(x,z)', ['~Faster(x,y)', '~Faster(y,z)', 'Faster(x,z)'], 1, ['y', 'z']
]]}, 'Buffalo': {1: [['Buffalo(Bob)', ['Buffalo(Bob)'], 0, ['Bob']]]}, 'Pig': {1: [['Pig(Pat)'
, ['Pig(Pat)'], 0, ['Pat']]]}}
Is the query resolvable: FALSE
```

Input 4:

- **Facts and Rules**

  C(x) => L(x, ca(y))
  L(x, ca(y)) => ~nf(x)
  Et(x,p(y)) => nf(x)
  B(x, p(y)) => Et(x,p(y)) | Car(x, p(y))
  B(J, p(y))

- **Query/ Goal:**

  C(J) => Car(J,p(y))

Output 4:

```
ashish/Desktop/Logic-Programming-Resolution-master/Final.py
Enter number of queries: 1
Enter 1 queries
C(J) => Car(J,p(y))
Enter number of rules: 5
Enter 5 rules
C(x) => L(x, ca(y))
L(x, ca(y)) => ~nf(x)
Et(x,p(y)) => nf(x)
B(x, p(y)) => Et(x,p(y)) | Car(x, p(y))
B(J, p(y))
***** Resolution Start *****
Query after negation=> C(J)
KB=>  {'~C': {1: [['~C(x)|L(x,ca(y))', ['~C(x)', 'L(x,ca(y))'], 0, ['x']], ['~C(x)|L(x,ca(y))'
, ['~C(x)', 'L(x,ca(y))'], 0, ['x']]]}, 'L': {3: [['~C(x)|L(x,ca(y))', ['~C(x)', 'L(x,ca(y))']
, 1, ['x', 'ca', 'y)']], ['~C(x)|L(x,ca(y))', ['~C(x)', 'L(x,ca(y))'], 1, ['x', 'ca', 'y)']]]}
, '~L': {3: [['~L(x,ca(y))|~nf(x)', ['~L(x,ca(y))', '~nf(x)'], 0, ['x', 'ca', 'y)']], ['~L(x,c
a(y))|~nf(x)', ['~L(x,ca(y))', '~nf(x)'], 0, ['x', 'ca', 'y)']]]}, '~nf': {1: [['~L(x,ca(y))|~
nf(x)', ['~L(x,ca(y))', '~nf(x)'], 1, ['x']], ['~L(x,ca(y))|~nf(x)', ['~L(x,ca(y))', '~nf(x)']
, 1, ['x']]]}, '~Et': {3: [['~Et(x,p(y))|nf(x)', ['~Et(x,p(y))', 'nf(x)'], 0, ['x', 'p', 'y)']
], ['~Et(x,p(y))|nf(x)', ['~Et(x,p(y))', 'nf(x)'], 0, ['x', 'p', 'y)']]]}, 'nf': {1: [['~Et(x,
p(y))|nf(x)', ['~Et(x,p(y))', 'nf(x)'], 1, ['x']], ['~Et(x,p(y))|nf(x)', ['~Et(x,p(y))', 'nf(x
)'], 1, ['x']]]}, '~B': {3: [['~B(x,p(y))|Et(x,p(y))|Car(x,p(y))', ['~B(x,p(y))', 'Et(x,p(y))'
, 'Car(x,p(y))'], 0, ['x', 'p', 'y)']], ['~B(x,p(y))|Et(x,p(y))|Car(x,p(y))', ['~B(x,p(y))', '
Et(x,p(y))', 'Car(x,p(y))'], 0, ['x', 'p', 'y)']]]}, 'Et': {3: [['~B(x,p(y))|Et(x,p(y))|Car(x,
p(y))', ['~B(x,p(y))', 'Et(x,p(y))', 'Car(x,p(y))'], 1, ['x', 'p', 'y)']], ['~B(x,p(y))|Et(x,p
(y))|Car(x,p(y))', ['~B(x,p(y))', 'Et(x,p(y))', 'Car(x,p(y))'], 1, ['x', 'p', 'y)']]]}, 'Car':
 {3: [['~B(x,p(y))|Et(x,p(y))|Car(x,p(y))', ['~B(x,p(y))', 'Et(x,p(y))', 'Car(x,p(y))'], 2, ['
x', 'p', 'y)']], ['~B(x,p(y))|Et(x,p(y))|Car(x,p(y))', ['~B(x,p(y))', 'Et(x,p(y))', 'Car(x,p(y
))'], 2, ['x', 'p', 'y)']]]}, 'B': {3: [['B(J,p(y))', ['B(J,p(y))'], 0, ['J', 'p', 'y)']], ['B
(J,p(y))', ['B(J,p(y))'], 0, ['J', 'p', 'y)']]]}, '~Car': {3: [['~Car(J,p(y))', ['~Car(J,p(y))
'], 0, ['J', 'p', 'y)']]]}}
Is the query resolvable: TRUE
```

Input 5:

- **Facts and Rules**

  Cat(x) => Likes(x,fish)
  Cat(y) & Likes(y,z) => Eats(y,z)
  Cat(jo)

- **Query/ Goal:**

  Eats(jo,fish)

```
ashish/Desktop/Logic-Programming-Resolution-master/Final.py
Enter number of queries: 1
Enter 1 queries
Eats(jo,fish)
Enter number of rules: 3
Enter 3 rules
Cat(x) => Likes(x,fish)
Cat(y) & Likes(y,z) => Eats(y,z)
Cat(jo)
***** Resolution Start *****
Query after negation=> ~Eats(jo,fish)
KB=>  {'~Cat': {1: [['~Cat(x)|Likes(x,fish)', ['~Cat(x)', 'Likes(x,fish)'], 0, ['x']], ['~Cat(
y)|~Likes(y,z)|Eats(y,z)', ['~Cat(y)', '~Likes(y,z)', 'Eats(y,z)'], 0, ['y']]]}, 'Likes': {2:
[['~Cat(x)|Likes(x,fish)', ['~Cat(x)', 'Likes(x,fish)'], 1, ['x', 'fish']]]}, '~Likes': {2: [[
'~Cat(y)|~Likes(y,z)|Eats(y,z)', ['~Cat(y)', '~Likes(y,z)', 'Eats(y,z)'], 1, ['y', 'z']]]}, 'E
ats': {2: [['~Cat(y)|~Likes(y,z)|Eats(y,z)', ['~Cat(y)', '~Likes(y,z)', 'Eats(y,z)'], 2, ['y',
 'z']]]}, 'Cat': {1: [['Cat(jo)', ['Cat(jo)'], 0, ['jo']]]}}
Is the query resolvable: TRUE
```

5. Agile Development Phases and Progression

Stage 1: Project Initiation (Sprint 0)

Goal: Understanding the project scope, objectives, and requirements.

Tasks:

- Talked about and defined the project's objectives with the team.
- Specified the project's scope, including the First-Order Logic system that will be used.
- Recognized essential attributes and features, such as input processing, CNF conversion, knowledge base creation, resolution procedure, and output generation.

Stage 2: Setting Up the Basics (Sprint 1)

Goal: Establishing the foundation of the project by setting up essential components.

Tasks:

- Implemented fundamental input management, enabling users to submit rules and queries.
- Created a function for converting input sentences into CNF.
- Developed data structures to keep rules and queries.
- Started constructing the knowledge base setup system.

Stage 3: Resolution Process (Sprint 2)

Goal: Focusing on the core of the project - the resolution process.

Tasks:

- Used the unification algorithm to accomplish variable binding and for identifying substitutes.
- Improving the resolution procedure by repeatedly using unification and resolution to generate new clauses.
- Created a queue-based management system to control the resolution procedure.
- Integrated the knowledge base and the resolution process.

Stage 4: Output Generation (Sprint 3)

Goal: Finalizing the core functionality and enabling users to receive results.

Tasks:

- Completed the resolution procedure to determine whether or not the query may be resolved.
- Created a method for output production that will display the resolution's outcome.
- Thoroughly evaluated and troubleshoot the resolution procedure.

Stage 5: Knowledge Base and Optimization (Sprint 4)

Goal: Enhance the efficiency of the knowledge base and overall system performance.

Tasks:

- Optimized the knowledge base setup mechanism for faster retrieval and resolution of clauses.
- Implemented techniques for handling duplicate sentences and repeated resolutions.
- Conducted performance testing to identify bottlenecks and areas for improvement.

Stage 6: User Experience and Refinement (Sprint 5)

Goal: Enhance user interaction and improve code quality.

Tasks:

- Enhanced the user interface by providing clear instructions and input validation.
- Refactored the codebase to improve readability, modularity, and maintainability.
- Reviewed and improved variable names, comments, and documentation.
- Ensured that the code adheres to best practices and style guidelines.

Stage 7: Testing and Validation (Sprint 6)

Goal: Thoroughly test the system and ensure its accuracy.

Tasks:

- Conducted unit tests for individual functions and components.
- Performed integration testing to ensure smooth interactions between modules.
- Validated the theorem prover's results against known logical principles.

Stage 8: Finalization and Deployment (Sprint 7)

Goal: Prepare for project completion and potential deployment.

Tasks:

- Reviewed the project against initial objectives and requirements.
- Addressed any remaining issues and bugs identified during testing.
- Prepared the final version of the codebase, ensuring it is well-documented and organized.

Maintaining an agile strategy and ensuring the project's success requires constant communication with team members, frequent testing, and incremental changes.

6. Conclusion and Future Work

Our project provides a workable implementation of a resolution-based theorem-proving method made to handle First-Order Logic (FOL) assertions, in order to wrap up. The project demonstrates a practical approach to automated reasoning and theorem proving within the context of formal knowledge representation by translating input queries and rules into the Conjunctive Normal Form (CNF) and utilizing SLD resolution. The knowledge base (KB) is well organized, the user input is handled with care, and the unification algorithm is capable of finding suitable replacements for clause resolution. Up until a resolution or refutation is reached, the resolution engine iteratively moves through the knowledge base and queries, utilizing unification to construct new phrases. The pursuit of improved efficiency through optimization, integration with cutting-edge unification techniques, and investigation of domain-specific knowledge integration represent future development potential. Error handling, the addition of effective data structures, and potential parallelization are all potential improvements. The project could develop into a more robust and adaptable tool for logic-based problem-solving across multiple domains by taking these improvements into account.

7. References

1. Course Slides
2. Wikipedia contributors. (Year, Month Day). SLD resolution. In Wikipedia. URL: https://en.wikipedia.org/wiki/SLD_resolution
3. YeGoblynQueenne. July 14, 2020. Real World Programming in SWI-Prolog. Online forum comment. In Hacker News. URL: https://news.ycombinator.com/item?id=23830332
4. Smith, Donald & Hickey, Tim. (1999). Multi-SLD resolution. 10.1007/3-540-58216-9_43. URL: https://www.researchgate.net/publication/2509897_Multi-SLD_resolution
5. Nilsson, U., & Maluszynski, J. (2000). LOGIC, PROGRAMMING AND PROLOG (2nd ed.). URL: https://www.ida.liu.se/~ulfni53/lpp/bok/bok.pdf
6. Maher, Michael. (2009). COMP4415 Lecture Notes on Logic Programming 1. URL:http://www.cse.unsw.edu.au/~cs4415/2010/resources/LP1.pdf