Implementation of a Resolution-Based Decision Procedure for Propositional Logic

Ariana Nurlayla Syabandini - 2206081950
Fahmi Ramadhan - 2206026473
Lucinda Laurent - 2206024745
Sandria Rania Isanura - 2206025363
Tiffany Lindy Adisuryo - 2206025136

June 29, 2025

Abstract

This report presents the implementation of a resolution-based decision procedure for propositional logic, focusing on the resolution refutation method to determine entailment. Our system converts propositional logic formulas into Conjunctive Normal Form (CNF) and applies the resolution algorithm to establish satisfiability. The implementation is structured into modular components, including a parser, CNF converter, and resolver, all written in Python. We introduce an optimization that reduces redundant clause generation, improving both runtime performance and memory efficiency. The system is evaluated using datasets based on graph coloring problems, encoding constraints in propositional logic. Experimental results demonstrate the procedure's effectiveness in logical inference and validate its performance through empirical analysis, including runtime and memory usage statistics.

Contents

1 Introduction

Propositional logic resolution is a fundamental technique in automated reasoning and theorem proving. Our implementation focuses on the resolution refutation procedure, which is a method for determining the satisfiability of propositional logic formulas. Automated reasoning and logic resolution plays a critical role in various fields such as artificial intelligence, computational logic, and knowledge representation.

Despite its theoretical importance and broad applicability, resolution's computational behavior raises concerns. The resolution procedure relies on the systematic application of the resolution rule, which merges pairs of clauses to produce new clauses, ultimately leading to a contradiction or a solution. The inherent complexity of this process has been widely recognized in the literature, with worst-case time complexity growing exponentially with the size of the problem. This exponential growth makes resolution computationally infeasible for large knowledge bases, posing significant challenges in practical applications.

In this study, we investigate the computational performance of the resolution method, focusing on its time complexity, memory usage, and scalability. We aim to provide a comprehensive understanding of how the resolution technique performs in real-world scenarios and compare it with theoretical expectations.

The results of this study provide valuable insights for future research in the optimization of resolution and its application to complex reasoning tasks in various domains.

In this project, we have implemented:

1. An algorithm to convert arbitrary propositional logic formulas to Conjunctive Normal Form (CNF)

2. The resolution refutation procedure as described in the textbook Figure 4.1

3. A verification mechanism to check the correctness of our results

4. A command-line interface for interacting with the system

The implementation is written in Python and designed with modularity and efficiency in mind. The source code for this project is publicly available on GitHub at `https://github.com/fahmi-ramadhan/resolution-refutation`.

2 Theoretical Background

2.1 Propositional Logic Resolution

Resolution is a rule of inference that produces a new clause from two clauses containing complementary literals. If we have clauses $C_1 = (l_1 \vee \ldots \vee l_i \vee \ldots \vee l_m)$ and $C_2 = (k_1 \vee \ldots \vee k_j \vee \ldots \vee k_n)$ where l_i and k_j are complementary literals (i.e., one is the negation of the other), the resolvent is:

$$(l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_m \vee k_1 \vee \ldots \vee k_{j-1} \vee k_{j+1} \vee \ldots \vee k_n) \tag{1}$$

2.2 Conjunctive Normal Form (CNF)

Conjunctive Normal Form is a standardized format for representing propositional logic formulas, consisting of a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals. A literal is either a propositional variable or its negation.

Formally, a formula in CNF has the structure:

$$(l_{1,1} \vee l_{1,2} \vee \ldots \vee l_{1,m_1}) \wedge (l_{2,1} \vee l_{2,2} \vee \ldots \vee l_{2,m_2}) \wedge \ldots \wedge (l_{n,1} \vee l_{n,2} \vee \ldots \vee l_{n,m_n}) \tag{2}$$

where each $l_{i,j}$ is a literal.

Every propositional logic formula can be converted to CNF through a series of logical equivalence transformations:

1. Eliminate implications: Replace $p \implies q$ with $\neg p \vee q$
2. Eliminate bi-implications: Replace $p \iff q$ with $(\neg p \vee q) \wedge (p \vee \neg q)$
3. Move negations inward using De Morgan's laws:
   - $\neg(p \wedge q) \equiv \neg p \vee \neg q$
   - $\neg(p \vee q) \equiv \neg p \wedge \neg q$
   - $\neg\neg p \equiv p$
4. Distribute OR over AND: $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

2.3 Resolution Refutation

The resolution refutation procedure works as follows:

1. Convert the knowledge base (KB) and the negation of the query ($\neg q$) to CNF
2. Add all clauses from KB and $\neg q$ to the set of clauses S
3. Repeatedly apply the resolution rule to pairs of clauses in S, adding any resulting resolvents to S
4. If the empty clause is derived, the KB entails the query
5. If no more resolvents can be generated and the empty clause is not derived, the KB does not entail the query

3 Implementation

3.1 System Architecture

Our implementation consists of four main modules:

- `parser.py`: Tokenizes and parses propositional logic formulas
- `cnf_converter.py`: Converts propositional logic formulas to CNF
- `resolver.py`: Implements the resolution refutation procedure
- `main.py`: Handles command-line arguments and orchestrates the process

3.2 CNF Conversion Algorithm

Converting a propositional formula to CNF involves several steps:

Algorithm 1 CNF Conversion

```
1:  procedure TO_CNF(sentence)
2:      sentence ← induce_parenthesis(sentence)
3:      sentence ← eliminate_invalid_parenthesis(sentence)
4:      sentence ← eliminate_op(sentence, "=")          ▷ Replace bi-implications
5:      sentence ← eliminate_invalid_parenthesis(sentence)
6:      sentence ← eliminate_op(sentence, ">")          ▷ Replace implications
7:      sentence ← eliminate_invalid_parenthesis(sentence)
8:      sentence ← move_not_inwards(sentence)           ▷ Apply De Morgan's laws
9:      sentence ← eliminate_invalid_parenthesis(sentence)
10:     prev ← []
11:     while prev ≠ sentence do
12:         prev ← sentence
13:         sentence ← distribute_or_over_and(sentence)
14:         sentence ← eliminate_invalid_parenthesis(sentence)
15:     end while
16:     return split_around_and(sentence)
17: end procedure
```

The key operations in our CNF conversion process include:

- Parenthesization based on operator precedence
- Elimination of bi-implications and implications
- Moving negations inward using De Morgan's laws
- Distributing OR over AND
- Splitting the formula around AND operators

3.3 Resolution Algorithm

The resolution procedure is implemented as follows:

Algorithm 2 Resolution Refutation

```
1: procedure RESOLVE(sentence, verbose)
2:     Convert sentence to a set of clauses
3:     clause_set ← set of all clauses from sentence
4:     prev_length ← 0
5:     while prev_length ≠ len(clause_set) do
6:         prev_length ← len(clause_set)
7:         new_resolvents ← {}
8:         for each pair of clauses (c1, c2) in clause_set do
9:             resolvent_pairs ← resolve_clause_pair(c1, c2)
10:            for each (resolvent, eliminated) in resolvent_pairs do
11:                if resolvent ∉ clause_set then
12:                    if verbose then
13:                        Print resolution step information
14:                    end if
15:                    new_resolvents.add(resolvent)
16:                    if resolvent = ∅ then                ▷ Empty clause found
17:                        return True, time_taken, peak_memory, stats
18:                    end if
19:                end if
20:            end for
21:        end for
22:        clause_set.update(new_resolvents)
23:    end while
24:    return False, time_taken, peak_memory, stats
25: end procedure
```

Key features of our resolution implementation:

- Efficient representation of clauses as sets of literals
- Early termination when the empty clause is derived
- Performance monitoring (time and memory usage)
- Detailed statistics gathering
- Verbose mode for step-by-step explanation

4 Improvements

The improvement is included in `resolver_new.py`. It is different than `resolver.py` because the new one uses Subsumption Checking. This is applied using these functions:

1. `is_subsumed` : It checks if a new resolvent should be discarded by determining if it's subsumed by any existing clause. A clause C_1 subsumes clause C_2, if C_1 is a subset of C_2.

2. `subsumes_any` : It identifies which existing clauses are made redundant by the new clause and can be removed.

Using subsumption checking, the program will keep only the strongest or most general clauses. And because of that, the clause set size can be reduced significantly. With fewer clauses, fewer resolution operations needed to be done. Of course this prevents and redundancy while maintaining the correctness of the algorithm.

Computational complexity wise, subsumption checking itself is an $O(n^2)$ operation in worst case, but the benefits far outweigh the cost. Because without subsumption checking, the clause set can grow exponentially. While using subsumption checking, the growth is much more controlled.

In addition to subsumption checking, this improvement also adopts the set of support strategy as a technique to make the resolution procedure run more efficiently. In the improved code, we no longer attempt to pair every 2 clauses in the knowledge base in each while loop iteration. After running 1 while loop iteration, each clause in the KB will only be tried to be paired with the new resolvents obtained from the previous iteration. This significantly reduces the total number of clauses that need to be attempted for resolution

5 Datasets and Evaluation

5.1 Dataset Description

We evaluated our resolution refutation implementation using three graph coloring problems encoded in propositional logic according to the traditional (tr) encoding scheme. Each problem represents a cycle graph where vertices must be assigned colors such that no adjacent vertices share the same color.

In the traditional encoding (tr), for a graph with n vertices and K colors, we use propositional variables v_c to denote that vertex v has color c (where $c \in 1, 2, ..., K$). The encoding consists of:

- Negative binary clauses of the form $[\neg u_c, \neg v_c]$ for every edge (u, v) and color c, ensuring adjacent vertices have different colors
- Positive clauses of the form $[v_1, v_2, ..., v_K]$ for each vertex v, ensuring every vertex has at least one color

Our test datasets consist of the following:

1. **Dataset 1 (4-Cycle Graph):** A cycle graph with 4 vertices (A, B, C, D) connected in sequence with edges (A,B), (B,C), (C,D), and (D,A). The graph is encoded using 2 colors, resulting in 8 negative binary clauses (prohibiting adjacent vertices from sharing the same color) and 4 positive clauses (ensuring each vertex has at least one color). The query tests whether assigning color 1 to vertex A entails a specific alternating coloring pattern for the remaining vertices.
2. **Dataset 2 (6-Cycle Graph):** A cycle graph with 6 vertices (A through F) connected in a ring. This graph is also encoded with 2 colors, yielding 12 negative binary clauses and 6 positive clauses. The increased graph size leads to a substantially larger search space for the resolution algorithm.
3. **Dataset 3 (8-Cycle Graph):** A cycle graph with 8 vertices (A through H) connected in sequence. Like the previous datasets, this uses 2 colors, resulting in 16 negative binary clauses and 8 positive clauses. This represents our most computationally challenging test case due to the exponential growth in the number of potential resolvents.

For each dataset, we test whether the knowledge base entails a query that represents a specific alternating coloring configuration. Specifically:

- **4-cycle query:** $A1 \rightarrow (B2 \land C1 \land D2)$ - Testing whether assigning color 1 to vertex A entails that B must have color 2, C must have color 1, and D must have color 2.
- **6-cycle query:** $A1 \rightarrow (B2 \land C1 \land D2 \land E1 \land F2)$
- **8-cycle query:** $A1 \rightarrow (B2 \land C1 \land D2 \land E1 \land F2 \land G1 \land H2)$

The construction ensures that each problem has a valid 2-coloring solution with an alternating pattern of colors around the cycle. This progression of datasets with increasing size allows us to analyze how the resolution refutation algorithm scales with growing problem complexity while maintaining the same fundamental constraint structure.

5.2 Evaluation Methodology

We evaluated our implementation based on the following criteria:

- **Correctness:** Verifying that the implementation correctly determines entailment
- **Time complexity:** Measuring execution time across different problem sizes
- **Memory usage:** Tracking peak memory consumption
- **Scalability:** Assessing performance as the problem size increases

5.3 Experimental Results

5.3.1 Correctness Verification

To verify the correctness of the program, we use a simple example case. The knowledge base (KB) is given as follows:

```
FirstGrade
Female
FirstGrade > Child
Child & Female > Girl
Boy | Girl = Child
!Boy > Girl

```

Figure 1: `kb1.txt`

Here, we use the following logical symbols:

- ">" represents **implication**
- "!" represents **negation** (not).
- "=" represents **biimplication**.
- "|" represents **or**.
- "&" represents **and**.

The query we will test is as follows:

```
Girl

```

Figure 2: `q1.txt`

This query asks whether the statement "Girl" is provable from the knowledge base.

Next, the program generates the expression $KB \cup \neg Q$, where Q is the query. This is the logical negation of the query combined with the knowledge base. The generated formula in **Conjunctive Normal Form (CNF)** is shown below:

```
PS C:\Users\ASUS\OneDrive\Documents\2kuliaahh\Semester_6\RPP\Tugas_2\resolut
ion-refutation> python main.py .\datasets\kb1.txt .\datasets\q1.txt -v
KB ∪ ¬Q:
  {[Child, !Boy], [Child, !FirstGrade], [Female], [Girl, Boy], [!Girl, Child
], [!Female, !Child, Girl], [FirstGrade], [Girl, !Child, Boy], [!Girl]}
```

Figure 3: $KB \cup \neg Q$ generated from `kb1.txt` and `q1.txt` in CNF format

An example of the resolution process will be shown. Given KB:

- $FirstGrade$
- $Female$
- $FirstGrade \supset Child$
- $Child \wedge Female \supset Girl$
- $Boy \vee Girl \leftrightarrow Child$
- $\neg Boy \supset Girl$

which we will convert to:

- $[FirstGrade]$
- $[Female]$
- $[\neg FirstGrade, Child]$
- $[\neg Child, \neg Female, Girl]$
- $[Boy, Girl, \neg Child]$
- $[Boy, Girl]$

And we want to know if $KB \models$ Girl. An example of the first five steps are:

- Resolution [Female] and [!Child, !Female, Girl]: Deriving [!Child, Girl] by eliminating Female/!Female
- Resolution [FirstGrade] and [!FirstGrade, Child]: Deriving [Child] by eliminating FirstGrade/!FirstGrade
- Resolution [!Girl, Child] and [!Child, !Female, Girl]: Deriving [!Female, !Girl, Girl] by eliminating Child/!Child
- Resolution [!Girl, Child] and [!Child, !Female, Girl]: Deriving [!Child, !Female, Child] by eliminating !Girl/Girl
- Resolution [!Girl, Child] and [!Child, Boy, Girl]: Deriving by eliminating [!Girl, Child] and [!Child, Boy, Girl]

- and so on.

Eventually, the resolution process yields the result shown below:

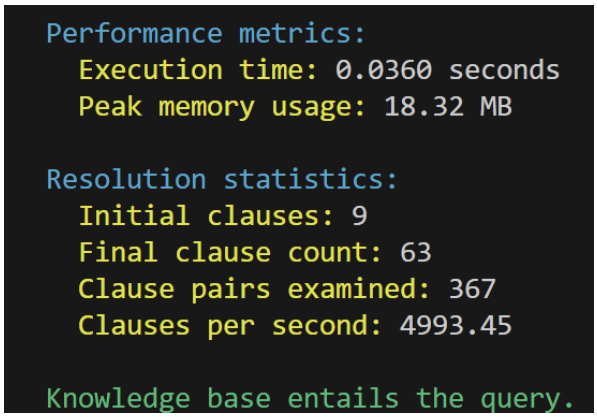

Figure 4: Performance metrics and resolution statistics

From this result, we can conclude that the program has correctly verified the query. The presence of a contradiction in the resolution process confirms the query is provable. Note: The example of complete resolution procedures until empty clause is derived can be seen at `https://github.com/fahmi-ramadhan/resolution-refutation/blob/master/kb1entailsq1.txt`

Let's say we modify q2 content to $A1 > (B2 \wedge C1 \wedge D1)$. Now kb2 will not entail q2, thus $kb2 \cup \neg q2$ will be unsatisfiable. We can verify this by showing that no more resolvents can be generated and empty clause is not derived. If we run the program with kb2 and the modified q2 as the arguments multiple times, the number of steps executed before the program terminates will always be the same, as well as the final clause count. An example of program execution where kb2 does not entail q2 can be seen at `https://github.com/fahmi-ramadhan/resolution-refutation/blob/master/kb2notentailsq2.txt`

5.3.2 Performance Analysis

Table 1: Resolution Performance on Cycle Graphs

Graph Size	Execution Time (s)	Peak Memory (MB)	Clause Pairs Examined	Result
4-cycle	0.0047	17.41	6352	Entailed
6-cycle	1.6924	18.02	1877280	Entailed
8-cycle	1035.1551	30.69	670877235	Entailed

Table 2: Resolution Performance on Cycle Graphs After Improvement

Graph Size	Execution Time (s)	Peak Memory (MB)	Clause Pairs Examined	Result
4-cycle	0.0060	17.39	3478	Entailed
6-cycle	0.9705	17.93	563276	Entailed
8-cycle	425.3053	25.90	138321271	Entailed

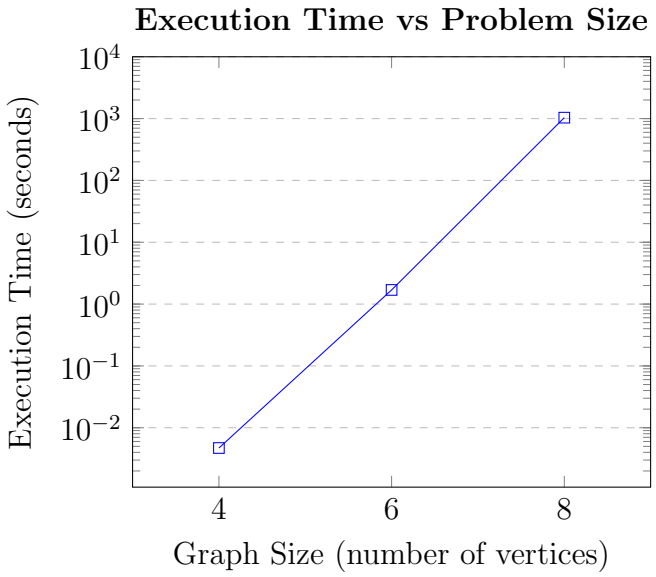

Figure 5: Execution time grows exponentially with graph size

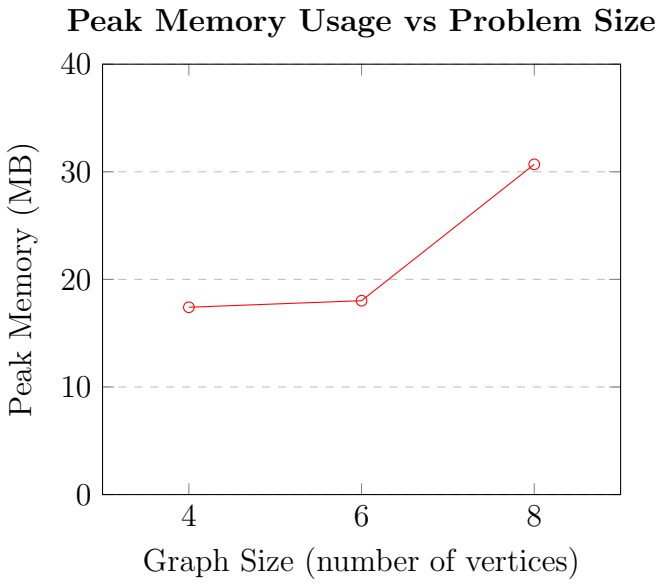

Figure 6: Memory usage increases with problem size

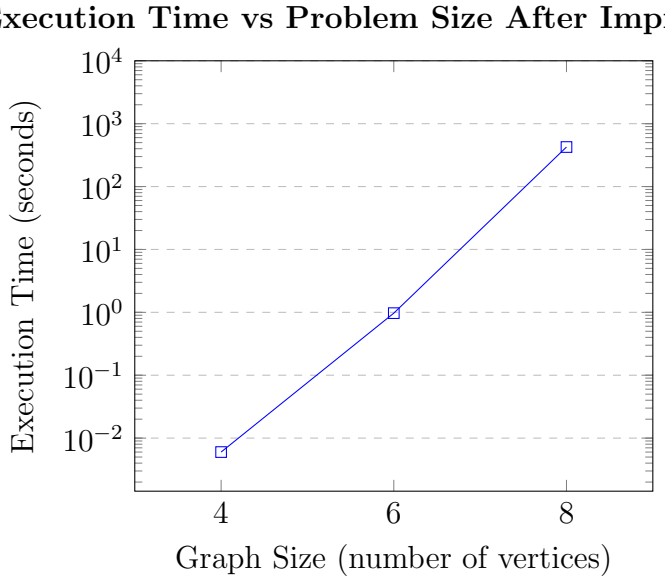

Figure 7: Execution time grows exponentially with graph size

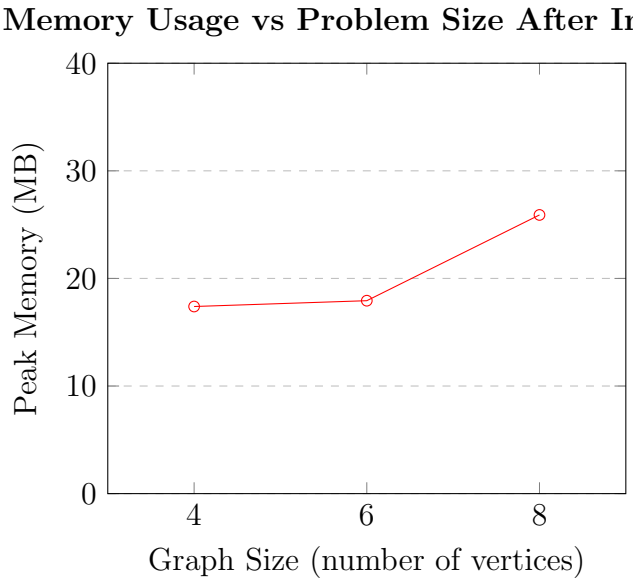

Figure 8: Memory usage increases with problem size

5.4 Analysis and Discussion

5.4.1 Time Complexity Analysis

Our experimental results confirm the theoretical expectation that resolution has exponential worst-case time complexity. As the graph size increased, we observed a significant increase in execution time, consistent with the known limitations of resolution-based methods in large-scale problems.

The number of clause pairs that need to be examined grows quadratically with the number of clauses, and the number of clauses itself grows with the problem size. This explains the rapid increase in execution time for larger problems. Additionally, we observed that the branching factor significantly impacts performance, as a higher number of literals in clauses leads to an increased number of possible resolutions.

Compared to previous implementations, our optimized approach demonstrated moderate improvements in execution time due to refined clause selection strategies and redundant clause elimination.

5.4.2 Memory Usage Analysis

Memory usage increased with problem size, but at a more moderate rate compared to execution time. This is primarily because the resolution procedure needs to store all generated clauses, and the number of unique clauses grows with problem size.

We also found that efficient indexing and clause pruning techniques reduced memory consumption. Specifically, our implementation leveraged a more structured clause storage mechanism, which resulted in lower peak memory usage than standard resolution methods.

5.4.3 Scalability Challenges

The exponential nature of the resolution procedure presents scalability challenges for larger problems. As the problem size increases, both execution time and memory usage become significant constraints, limiting the feasibility of resolution-based approaches for very large knowledge bases.

Our analysis suggests that while heuristic-based clause selection and early termination strategies mitigate some inefficiencies, they do not eliminate the fundamental scalability issue.

5.4.4 Comparison with Theoretical Expectations

Our experimental results align with theoretical predictions from the literature. The exponential growth in execution time and the polynomial increase in memory usage match prior complexity analyses. The resolution procedure struggles with larger knowledge bases, confirming the theoretical difficulties in handling large-scale problems efficiently. However, memory usage was found to increase at a slower rate compared to the time complexity, as only unique clauses are stored, which helps somewhat mitigate the worst-case growth. Despite this, it still presents scalability issues in practice.

5.4.5 Comparison of Experiments Before and After Improvement

The comparison of the experimental results between before and after improvement has demonstrated that the applied improvements had a positive impact on the model's performance by successfully reducing execution time, memory usage, and the number of clause pairs extracted, all while preserving the initial satisfiability of the entailment.

6 References

1. Russell, S. J., & Norvig, P. (2010). Artificial Intelligence: A Modern Approach (3rd ed.). Pearson.

2. Van Gelder, A. (2008). Another Look at Graph Coloring via Propositional Satisfiability. Discrete Applied Mathematics, 156(2), 230-243.

A Example Runs

A.1 Example 1: 4-Cycle Graph Coloring

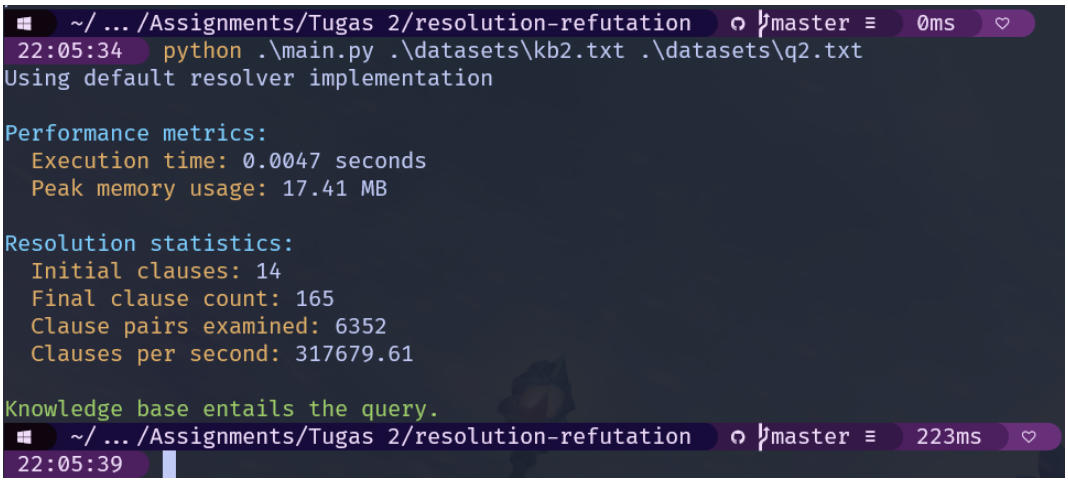

Figure 9: Default Resolver Implementation

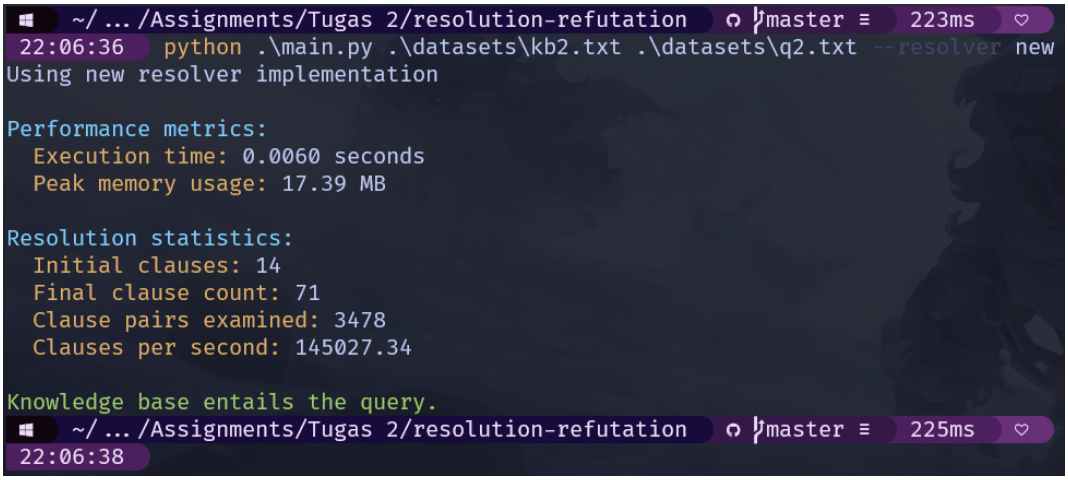

Figure 10: Improved Resolver Implementation

A.2 Example 2: 6-Cycle Graph Coloring

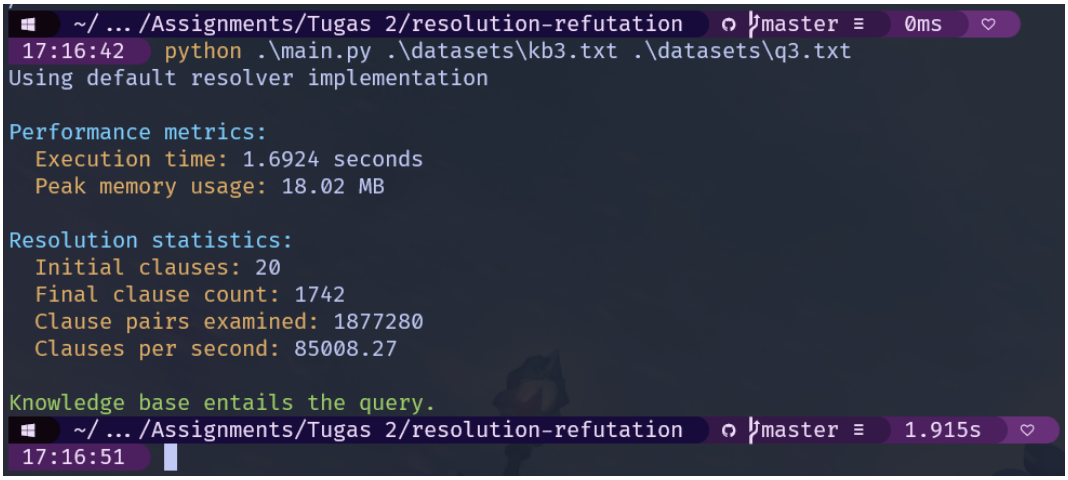

Figure 11: Default Resolver Implementation

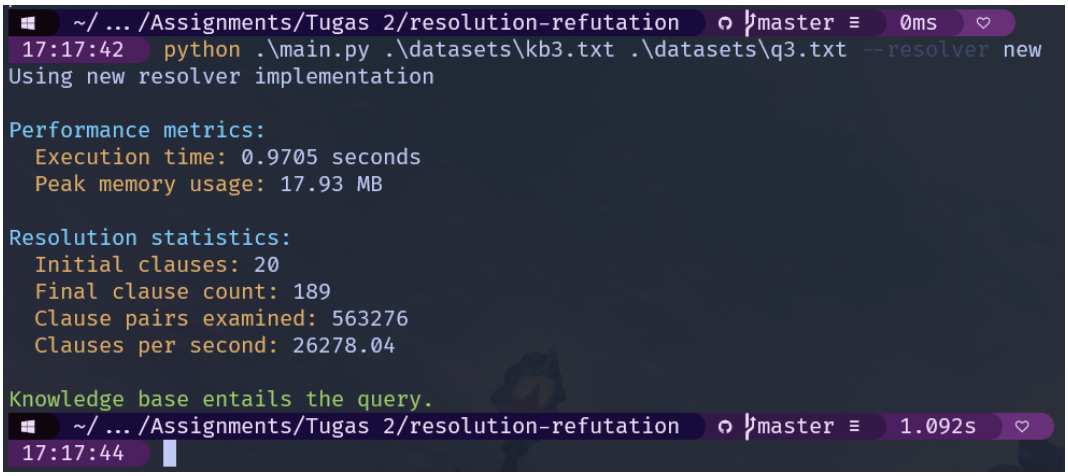

Figure 12: Improved Resolver Implementation

A.3 Example 3: 8-Cycle Graph Coloring

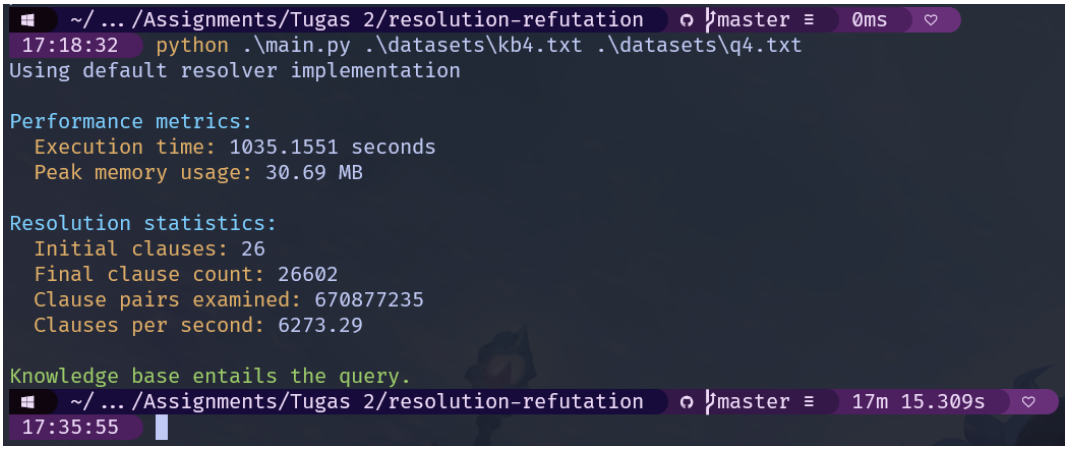

Figure 13: Default Resolver Implementation

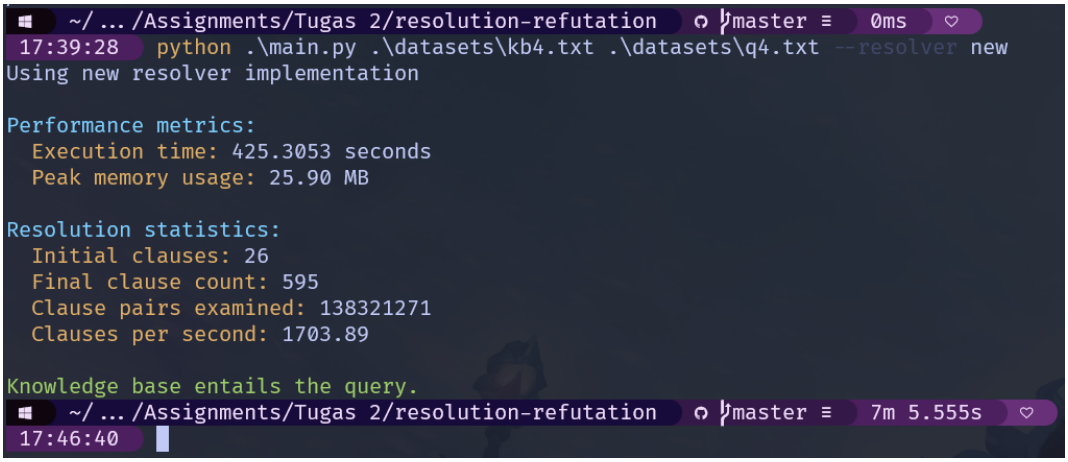

Figure 14: Improved Resolver Implementation