VIETNAM NATIONAL UNIVERSITY, HCM
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

SUBJECT: CSC14003 - Artificial Intelligence

Report: Lab02: PL Resolution

Teacher : Bùi Tiến Lên
Trần Quốc Huy
Phạm Trọng Nghĩa
Student : Hồ Thế Phúc – 21127670
Class : 21CLC8

Contents

I. Check list:

No.	Criteria	Degree of completion
1	Read the input data and successfully store it in some data structures.	100%
2	The output file strictly follows the lab specifications.	100%
3	Implement the propositional resolution algorithm.	100%
4	Provide a complete set of clauses and exact conclusion.	100%
5	Five test cases: both input and output files.	100%
6	Discussion on the algorithm’s efffiency and suggestions.	100%

II. function explanation:

II.1. main:

1. It prompts the user to input the paths to the input and output files.

2. Reads the input clauses (knowledge base and queries) from the specified input file using the `read_input_file` function.

3. Initializes two empty lists: `results` to store the results of theorem proving for each query, and `loopsres` to store the resolution steps (loops) for each query.

4. For each query in the `queries` list, it calls the `pl_resolution` function to perform resolution-based theorem proving using the knowledge base and the current query. The resulting `result` and `loops` lists are appended to `results` and `loopsres` respectively.

5. Determines the overall conclusion based on the `results`. If all query results are True (satisfiable), it sets `conclusion` to "YES", otherwise "NO".

6. Opens the specified output file in write mode using a `with` statement.

7. Iterates through the `loopsres` list, which contains the resolution steps for each query:
    a. Writes the number of clauses in the resolution step to the output file.
    b. Formats and writes each clause to the output file.
    c. If the clause is empty, it writes an empty line.

8. Writes the `conclusion` (either "YES" or "NO") to the output file.

In summary, the `main` function orchestrates the entire process of reading input clauses, performing resolution-based theorem proving for multiple queries, and writing the resolution steps and conclusions to an output file

II.2. Read input file:

`read_input_file`, is designed to read clauses from an input file and organize them into two separate lists: one for knowledge base (KB) clauses and another for query clauses. The input file is expected to have a specific structure with the following format:

1. The first line contains an integer representing the number of query clauses.
2. The next 'num_query_clauses' lines contain query clauses.
3. The next line contains an integer representing the number of KB clauses.
4. The next 'num_kb_clauses' lines contain KB clauses.

Here's a step-by-step breakdown of how the function works:

1. Initialize two empty lists, `kb` and `queries`, to store KB and query clauses respectively.

2. Open the input file located at `file_path` in read mode using a `with` statement. The `with` statement ensures that the file is properly closed after reading, even if an exception occurs.

3. Read the first line of the file, which contains the number of query clauses. Convert the value to an integer using `int()` and store it in the variable `num_query_clauses`.

4. Use a loop to read the next `num_query_clauses` lines, each representing a query clause:
    a. Read a line from the file and strip any leading/trailing whitespace using `.strip()`.
    b. Check if the string 'OR' exists in the query clause.
    c. If 'OR' is present, split the query clause using ' OR ' as the delimiter. Create a list of lists, where each inner list contains a single element from the split clause. Append this list to the `queries` list.
    d. If 'OR' is not present, split the query clause using ' OR ' as the delimiter and directly append the resulting list to the `queries` list.

5. Read the next line of the file, which contains the number of KB clauses. Convert the value to an integer using `int()` and store it in the variable `num_kb_clauses`.

6. Use a loop to read the next `num_kb_clauses` lines, each representing a KB clause:
    e. Read a line from the file and strip any leading/trailing whitespace using `.strip()`.
    f. Split the KB clause using ' OR ' as the delimiter and append the resulting list to the `kb` list.

7. Close the file.

8. Return the `kb` and `queries` lists as the output of the function.

II.3. Resolve:

`resolve`, is implementing a part of the Resolution Refutation inference rule used in logic and artificial intelligence. This rule is commonly applied in automated theorem proving, particularly in propositional logic and first-order logic, to check the validity of logical formulas or to derive new formulas from a knowledge base.

Here's a step-by-step explanation of how the function works:

1. The function takes two arguments, `c1` and `c2`, which represent two clauses that you want to resolve.

2. It starts by converting `c1` and `c2` into lists using the `list()` function. This is done to ensure that the clauses can be modified during the resolution process.

3. Two empty lists are created: `resolvents` to store the resolved clauses, and `flag` to indicate whether any resolution was successful.

4. The function iterates through each literal `l1` in clause `c1`:
    a. It checks if `l1` is negated (starts with '-') and extracts the root literal without the negation.
    b. For each literal `l2` in clause `c2`, the same process is repeated to extract the root literal `root_l2`.
    c. It checks if the root literals of `l1` and `l2` are the same and if one of them is negated while the other is not. This is a crucial step in the resolution process, where complementary literals (one negated and one non-negated) can be resolved.

5. If a pair of complementary literals is found, a resolution is possible. The `flag` is set to `True` to indicate success.
    d. A copy of `c1` is made to preserve the original clause.
    e. Elements from `c2` are added to the copy using the `extend()` method.
    f. The two complementary literals `l1` and `l2` are removed from the copy.
    g. The copy is converted into a set and then back into a list, which effectively removes duplicate literals.
    h. The resulting list is assigned to `resolvent` and added to the `resolvents` list.

6. Once all possible resolutions have been attempted, the function returns both the list of resolved clauses (`resolvents`) and a boolean `flag` indicating whether any resolution was successful.

II.4. supporting functions:

a. `remove_duplicates(clauses)`:
    This function takes a list of clauses as input and returns a new list containing only the unique clauses, removing any duplicates. It iterates through the input list and checks if the current clause is already in the `unique_clauses` list. If not, it adds the

clause to the list of unique clauses. This function helps in simplifying the knowledge base by eliminating redundant clauses.

b. `contains_complementary_literals(literals)`:
   This function takes a list of literals as input and checks if the list contains complementary literals, which are literals and their negations. It does this by iterating through the list of literals and checking if the negation of each literal exists in the list. If a complementary pair is found, the function returns `True`; otherwise, it returns `False`. This function is used to determine whether a clause contains complementary literals, which is a key step in the resolution process.
c. `remove_complementary_literals(clauses)`:
   This function takes a list of clauses as input and returns a new list containing only the clauses that do not have complementary literals. It iterates through the input list of clauses and checks if each clause contains complementary literals using the `contains_complementary_literals` function. If a clause does not contain complementary literals, it is added to the `unique_clauses` list. This function is used to remove clauses that are trivially resolved due to containing complementary literals.

II.5.PL_resolution:

It's designed to determine the satisfiability of a knowledge base (KB) with respect to a set of query clauses. Here's a breakdown of how the `pl_resolution` function works:

1. The function takes two arguments: `kb` (knowledge base) and `query` (list of query clauses).

2. The function starts by creating a copy of the knowledge base (`kb`) called `clauses`.

3. It processes the query clauses to prepare them for resolution:
   a. If a query is a list (nested), it inverts the literals by adding a negation sign '-' to the beginning of each literal.
   b. If a query is a string and starts with '-', it removes the negation sign.
   c. If a query is a string and doesn't start with '-', it adds a negation sign.

4. The modified query clauses are appended to the `clauses` list.

5. The main loop of the resolution process begins (`while True`).

6. Inside the loop, it iterates over all pairs of clauses in the `clauses` list and attempts to resolve them:
   d. The `resolve` function is called on each pair of clauses to find possible resolvents.
   e. If resolvents are found (`flag` is `True`), duplicates are removed and complementary literals are eliminated from the resolvents.
   f. If resolvents are empty and `flag` is still `True`, this indicates a contradiction, and the function returns an empty list (unsatisfiable) and a list of resolution steps (`loops`).
   g. Otherwise, the processed resolvents are added to the `new_clauses` list.

7. After iterating through all pairs of clauses, the new clauses (`new_clauses`) are processed:
    h. Duplicates are removed.
    i. Clauses already present in the original `clauses` are filtered out.
    j. Complementary literals are removed.

8. If all clauses in `new_clauses` are already present in the original `clauses`, it means no further resolutions can be done, and the loop breaks.

9. The processed `new_clauses` are added to the `clauses` list, duplicates are removed, and the loop continues.

10. Once the loop ends, the function returns an empty list and the list of resolution steps (`loops`), indicating the steps taken during the resolution process.

III. Discussion on the algorithm's efficiency and suggestions to improve:

The pl_resolution algorithm implements a resolution-based theorem proving approach for propositional logic. While resolution is a sound inference rule (meaning if it derives a result, that result is true), its efficiency can vary depending on the complexity of the knowledge base and the queries. Let's discuss the algorithm's efficiency considerations:

1. Time Complexity: The primary factor affecting the algorithm's time complexity is the nested loop structure. The algorithm iterates through all possible pairs of clauses in the knowledge base to find resolvents. If there are N clauses, this leads to an O(N^2) complexity. For each pair of clauses, the resolve function is called, which itself involves iterating through each literal in both clauses. This can further contribute to the overall time complexity. The process continues until no new clauses are generated, which could result in a relatively high number of iterations.

2. Space Complexity: The algorithm's space complexity is affected by the storage of clauses, resolvents, and intermediate data structures. The clauses list stores both the knowledge base and modified query clauses, and new clauses are added to it during resolution. The new_clauses list temporarily holds newly generated clauses for each iteration. The loopsres list accumulates the resolution steps for each query. All of these lists could lead to additional space consumption, but their sizes might vary during execution.

3. Clause Complexity: The complexity of the input clauses also significantly impacts the algorithm's efficiency. Large and complex clauses can lead to more resolvents being generated, increasing the time required for each iteration. Conversely, smaller and simpler clauses might lead to faster resolution.

4. Duplication and Filtering: The algorithm frequently calls functions like remove_duplicates and remove_complementary_literals to filter out duplicates and trivially resolved clauses. While these operations are necessary for correctness, they also add overhead, especially when dealing with large sets of clauses.

5. Early Termination: In some cases, the algorithm might be able to determine unsatisfiability or satisfiability earlier in the process, potentially before exhausting all iterations. For example, if a contradiction is found during resolution, the algorithm can terminate early.

6. Optimizations: Depending on the specific use case, there might be ways to optimize the algorithm. For instance, certain types of clauses or literals can be indexed for faster lookup. Advanced data structures or heuristics can be employed to prioritize which pairs of clauses to resolve, potentially avoiding unproductive resolutions.

7. Complexity of Queries: The number and complexity of queries can also influence the algorithm's efficiency. Resolving each query against the knowledge base takes time, and if there are many queries, the total time taken can increase.

8. Implementation Language and Data Structures: The efficiency of the algorithm can also depend on the programming language being used and the inherent efficiency of the data structures used to represent clauses and literals.

In conclusion, while the resolution-based algorithm is sound for propositional logic, its efficiency can be highly variable based on factors such as input clause complexity, query complexity, and the efficiency of the implementation. In practice, for larger and more complex problems, various optimizations and advanced data structures might be needed to improve the algorithm's performance.

IV. References:

1."Artificial Intelligence: A Modern Approach" by Stuart Russell and Peter Norvig.
2. GitHub - kieuconghau/pl-resolution: HCMUS - Artificial Intelligence - Lab 2: Popositional Logic - Resolution