Student: Nguyễn Quốc Thắng

ID: 22127385

REPORT

PROPOSITIONAL LOGIC - RESOLUTION

1. Overview of the program structure

This program implements the Propositional Logic Resolution (PL- Resolution) algorithm in Python. The main functions of the program are:

- Read Input: The program reads input data from a text file – "input.txt", consisting of the alpha clause and other clauses. The alpha clause represents the set of axioms of the logic system. Other clauses provide additional information to the knowledge base.

- Construct Knowledge Base: The input data is used to construct a knowledge base, which is a collection of logical clauses. Each clause is represented as a *'Clause'* object containing a list of propositional symbols and their negation states.

- Perform PL-Resolution: The PL-Resolution algorithm is applied to the knowledge base to infer new clauses. The algorithm works by combining existing clauses to generate new ones, eliminating contradictions, and drawing conclusions.

- Write Output: The program writes the results of the resolution process to a text file – "output.txt". The output includes the number of new clauses generated at each step and the entailment decision (true or false).

2. Details about functions and installation methods

2.1. Class Literal:

- **Purpose:** Represents a single propositional symbol with negation.
- **Attributes:**
  - symbol (string): the propositional symbol (e.g., "A", "B").
  - negation (bool): True if the literal is negated, False otherwise.
- **Important Methods:**
  - __repr__(self): Returns a string representation of the literal (e.g., "A" or "-B").
  - __eq__(self, literal): Compares two literals for equality based on symbol and negation.
  - __hash__(self): Defines a unique hash value for the literal (used for efficient storage in sets).

- __lt__(self, literal): Defines the comparison order between literals (negated literals come first).

+ negate(self): Inverts the negation state of the literal (e.g., "A" becomes "-A").

+ is_opposite(self, literal): Checks if two literals are opposite (same symbol, different negation).

2.2. Class Clause:

- Purpose: Represents a disjunction (OR) of literals.

- Attribute:

literals (list[Literal]): A list containing the literals within the clause.

- Flatform Function:

+__repr__(self): Returns a string representation of the clause (e.g., "A | B | -C").

+__eq__(self, clause): Compares two clauses for equality based on the set of literals they contain.

+__hash__(self): Defines a unique hash value for the clause (used for efficient storage in sets).

+__lt__(self, clause): Defines the comparison order between clauses (based on number of literals and then lexicographic order of literals).

+ is_empty(self): Checks if the clause contains no literals.

+ is_meaningless(self): Checks if the clause contains a literal and its negation (logically unsatisfiable).

+ add_literal(self, literal): Adds a literal to the clause's list of literals.

+ clean(self): Removes duplicate literals and sorts the remaining literals in the clause.

+ negate_literals(self): Negates all literals within the clause (flips the negation state of each literal).

+ parse_clause(string_clause) : Parses a string representation of a clause into a Clause object.

+ clone_without_literal(self, literal=None): Creates a copy of the clause excluding a specific literal (useful for generating resolvents).

+ resolve(clause1: Clause, clause2: Clause) : Implements PL-Resolution. Takes two clauses, finds complementary literals (one positive, one negative for the same symbol), and generates new clauses (resolvents) by excluding those literals.

2.2. KnowledgeBase Class:

- **KnowledgeBase Class:**
  - The KnowledgeBase class represents a knowledge base of logical clauses.
  - It is used for automated reasoning and knowledge representation.
  - Key components:
    self.clauses: A list to store logical clauses.
- **Function** build_knowledge_base(alpha_string, clause_strings):

```
def build_knowledge_base(self, alpha_string, clause_strings):
        alpha_string = alpha_string.strip()
        alpha_literals = alpha_string.split('OR')
        for alpha_literal in alpha_literals:
          clause = Clause.parse_clause(alpha_literal)
          clause.negate_literals()
          self.add_clause(clause)
        for clause_str in clause_strings:
          clause = Clause.parse_clause(clause_str)
          clause.clean()
    self.add_clause(clause)
```

  - **Purpose:** Constructs the knowledge base from an alpha clause (query) and a list of clause strings.
  - **Steps:**
    - Process the alpha clause:
      - Split the alpha_string by "OR" to get individual literals.
      - Parse each literal into a Clause object.
      - Negate the literals (since alpha is negated).
      - Add these negated literals to the knowledge base.
    - Process the clause strings:
      - Parse each clause string into a Clause object.
      - Clean the clause (remove duplicates and tautologies).
      - Add the cleaned clause to the knowledge base.
- **Function** pl_resolution():

```
        def pl_resolution(self):
            input_clauses = set(self.clauses)
            output_clauses = []
            is_unsatisfiable = False

            while True:
              new = set()
              for (clause1, clause2) in combinations(input_clauses, 2):
                resolvents, is_empty = Clause.resolve(clause1, clause2)
                new.update(resolvents)
                is_unsatisfiable |= is_empty

              diff_clauses = new.difference(input_clauses)
              output_clauses.append(diff_clauses)
              input_clauses.update(new)

              if is_unsatisfiable:
                return True, output_clauses
              if not diff_clauses:
          return False, output_clauses
```

- **Purpose:** Performs PL-Resolution on the knowledge base.
- **Steps:**
  1. Initialize:
     - input_clauses: Set of clauses in the knowledge base.
     - output_clauses: List to store derived clauses.
     - is_unsatisfiable: Flag to track unsatisfiability.
  2. Iteratively combine pairs of clauses:
     For each pair (clause1, clause2) in input_clauses:
     - Resolve them to get new clauses (resolvents).
     - Update new with the resolvents.
     - Set is_unsatisfiable if an empty clause is derived.
  3. Update knowledge base:
     - Add new clauses to input_clauses.
     - Append the difference between new and input clauses to output_clauses.
  4. Termination conditions
     - If is_unsatisfiable, return True and output_clauses.
     - If no new clauses, return False and output_clauses.

2.3. Input-Output Function:

```
def read_input(input_file):
    with open(input_file, 'r') as file:
        alpha = file.readline().strip()
        num_clauses = int(file.readline().strip())
        clauses = [file.readline().strip() for _ in range(num_clauses)]
    return alpha, clauses

def write_output(output_file, output_clauses, is_entailed):
    with open(output_file, 'w') as file:
        for clauses in output_clauses:
            file.write('{}\n'.format(len(clauses)))
            for clause in clauses:
                file.write('{}\n'.format(clause))
        if is_entailed:
            file.write("YES\n")
        else:
            file.write("NO\n")
```

1. read_input(input_file):

   - **Purpose:** Reads input data from a file.
   - **Parameters:**
     - input_file: The path to the input file.
   - **Steps:**
     1. Opens the specified file in read mode.
     2. Reads the first line (alpha clause) and removes leading/trailing whitespace.
     3. Reads the second line (number of clauses) and converts it to an integer.
     4. Reads the remaining lines (clause strings) and stores them in a list.
     5. Returns the alpha clause and the list of clause strings.

2. write_output(output_file, output_clauses, is_entailed):

   - **Purpose:** Writes output data to a file.
   - **Parameters:**
     - output_file: The path to the output file.
     - output_clauses: A list of lists, where each inner list represents a set of derived clauses.

- is_entailed: A boolean indicating whether the knowledge base entails the query.

– **Steps:**

1. Opens the specified file in write mode.
2. Writes the length of each set of clauses followed by the individual clauses.
3. If is_entailed is True, writes “YES”; otherwise, writes “NO”.

2.4. Main Function:

```
def main(input_file, output_file):
    alpha, clauses = read_input(input_file)
    kb = KnowledgeBase()
    kb.build_knowledge_base(alpha, clauses)
    is_entailed, output_clauses = kb.pl_resolution()
    write_output(output_file, output_clauses, is_entailed)
```

The purpose of the main function is to orchestrate the entire process of solving a logical inference problem using the provided input data.

1. Reading Input.
2. Building the Knowledge Base.
3. Performing PL-Resolution.
4. Writing Output.

3. Test

STT	Input	Output
1	input1.txt 1 -A 2 4 3 -A OR B 4 B OR -C 5 A OR -B OR C 6 -B	output1.txt 1 3 2 -C 3 B 4 -A 5 4 6 A OR -B 7 A OR C 8 {} 9 -B OR C 10 YES

<table>
<tr>
<td>2</td>
<td>input2.txt
1 -A OR B
2 4
3 -A OR B OR C
4 -B OR -C
5 -C OR D
6 -D OR E</td>
<td>output2.txt
1 4
2 -A OR B OR D
3 -C OR E
4 B OR C
5 -A OR C
6 8
7 -A OR E
8 -A OR B OR E
9 B OR D
10 -A OR -C OR D
11 -A OR -B
12 -A OR D
13 C
14 B OR E
15 3
16 D
17 -A OR -C OR E
18 E
19 0
20 NO</td>
</tr>
<tr>
<td>3</td>
<td>input3.txt
1 -C OR D
2 3
3 -A OR B
4 -B OR -C
5 C OR -D</td>
<td>output3.txt
1 3
2 -B
3 -B OR -D
4 -A OR -C
5 2
6 -A
7 -A OR -D
8 0
9 NO</td>
</tr>
</table>

4	input4.txt 1 -A OR B OR C 2 4 3 -A OR -B OR C 4 -B OR -C 5 -A OR -D 6 -C OR -D OR E	output4.txt 1 4 2 -B OR C 3 -D 4 -A OR -B OR -D OR E 5 -A OR -B 6 1 7 -B OR -D OR E 8 0 9 NO
5	input5.txt 1 -B 2 3 3 -A OR B 4 B OR -C 5 A OR -B OR C	output5.txt 1 1 2 A OR C 3 2 4 A OR B 5 B OR C 6 0 7 NO

4. Advantages and disadvantages of PL-resolution

- **Advantages:**
  - **Completeness:** If a solution exists, the resolution algorithm will find it.
  - **Space-Efficient**: Compared to other methods, resolution significantly reduces search space.
  - **Ease of Implementation**: The algorithm is straightforward without complex inference rules.
- **Disadvantages:**
  - **Soundness**: It is not entirely sound; if no solution exists, resolution does not guarantee this result.
  - **Limited to Propositional Logic:** Cannot be extended to handle hybrid logic or higher-order logic.
  - **Challenges with Many Variables:** With numerous variables, the number of resolvents grows substantially.

5. Proposed Solutions

- We can Combining Modus Ponens with Resolution.
- We can use Modus Ponens during the resolution process:

Modus ponens inference rule (for Horn Form)

$$\frac{\alpha_1, \ldots, \alpha_n, \alpha_1 \wedge \cdots \wedge \alpha_n \implies \beta}{\beta}$$

- Whenever we derive a resolvent, check if it matches the form of an implication (P → Q).
- If so, apply Modus Ponens to infer additional conclusions.

This integration enhances both completeness and accuracy.

– **Benefits:**
  - **Completeness:** By combining resolution with Modus Ponens, we cover more inference patterns, increasing the likelihood of finding a solution.
  - **Accuracy:** Modus Ponens ensures that valid implications are correctly applied during the inference process.
– Example:

  Suppose we have the following premises:

  1. (P → Q)

  2. (P)

  We apply resolution to derive a new clause: (Q).

  Now, using Modus Ponens, we infer: (Q).

6. Summary

STT	Đặc tả tiêu chí	Hoàn thành (%)
1	Đọc dữ liệu đầu vào và lưu trong cấu trúc dữ liệu phù hợp	100
2	Cài đặt giải thuật hợp giải trên logic mệnh đề	100
3	Các bước suy diễn phát sinh đủ mệnh đề và kết luận đúng	100
4	Tuân thủ mô tả định dạng của đề bài	100
5	Báo cáo test case và đánh giá	100

- END -