Lab 02. PL-Resolution

Information

- **Student ID:** 18127185
- **Name:** Bùi Vũ Hiếu Phụng
- **Checklist:**
  - [x] Read input and store knowledge base
  - [x] Convert to CNF (just AND, OR clauses)
  - [x] Implement the PL-Resolution algorithm
  - [x] Write output following the lab specifications
  - [x] Test cases
  - [x] Report & discussion

Quick start

- Save input file in `input/` folder
- Change directory to `src/` folder
- Run `python main.py`
- Receive output files based on input at `output/` folder

Description

Knowledge base structure

Clauses

- In lab description, we can easily see that clauses are already in CNF form. So we can express these clauses as list of strings which means that
  - Between elements (literals) in the list is the logical disjunction (OR)
  - Between lists is the logical conjunction (AND)
- In the case of α (query) has multiple clauses, $\neg\alpha$ will not in CNF form, so we have to implement a function to convert $\neg\alpha$ to CNF form

Knowledge base

- **Class KnowledgeBase** (KB) includes clauses in format that was mentioned above and some methods to process clauses in PL-Resolution
- Also there are some other notes (from lab description)
  - Literals within a clause (both in input data and output data) are sorted following the alphabetical order
  - Clause in which two complementary literals appear can be discarded

Functions and methods

- `KB.getNegative_atom(atom)` : get negative form of a literal.

  For example: `KB.getNegative_atom('A') = '-A'` or `KB.getNegative_atom('-A') = 'A'`

- `KB.checkComplementary(clause)` : Detect the complementary case line: $A \vee B \vee -B$ by loop through every literal and check if its negative form exists in the clause or not

- `KB.addClause(clause)` : add clause to KB only if it is not in the current KB and is not an complementary clause

- `KB.normClause(clause)` : Remove duplicate literals and sort them lexicographically

- `KB.remove_eval(clauses)` : Remove always True clauses
  - For example:

    ```
    ['-A', '-C']        (1)
    [' A', '-B']        (2)
    ['-B', '-C']        (3)
    ['-C']              (4)
    [' A', '-B', 'C']   (5)
    ```

- Because $-C$ is True, the (1) and (3) clauses will become $-A \vee True$ and $-B \vee True$ which means that we can discard both of them. At the same time $A \vee -B$ is true then $A \vee -B \vee C$ is always. As a result, this KB will become:

  ```
  [' A', '-B']
  ['-C']
  ```

- `KB.getNegative_query(query)`
  - To get $\neg\alpha$, we get negative of every literal in every clause and save in lists to present AND clauses
    - For example: $\text{negative}(A \vee B \vee C) = \neg A \wedge \neg B \wedge \neg C$ will be present as:

      ```
      [['-A'], ['-B'], ['-C']]
      ```

- If there are more than one clauses in α, means that $\neg\alpha$ will have multiple AND clauses connected by OR. To convert them to CNF, we pass them to `KB.toCNF(clauses)` which we will discuss about next
    - For example: $\text{negative}((A \vee B) \wedge (\neg A \vee C)) = (\neg A \wedge \neg B) \vee (A \wedge \neg C)$

      At this step, this clause will be present as:

      ```
      [[['-A'], ['-B']] , [['A'], ['-C']]]
      ```

- `KB.toCNF(clauses)`
    - We do not have to care about implication, equivalence and negation. Here we implement the distributive law (OR over AND) to convert $\neg\alpha$ in the case mentioned in the above paragraph.
    - Firstly, we product every list in the clauses arguments
    - Secondly, flat list to present *disjunction*, normalize and discard complementary or duplicate clauses
        - For example:

          ```
          [['-A'], ['-A']] -> ['-A', '-A'] -> discard
          [['-A'], ['-C']] -> ['-A', '-C']
          [['-B'], [' A']] -> ['-B', ' A'] -> ['A', '-B']
          [['-B'], ['-C']] -> ['-B', '-C']
          ```

- `KB.resolve(clause_i, clause_j)`: Implementation of Resolution Inference Rule
    - Loop for every literal in `clause_i`, if its negative form exists in `clause_j`, resolve them
    - Discard complementary or duplicate clauses

- `KB.PL_Resolution(query)`: Implementation of PL-Resolution

  We have to prove $KB \models \alpha$. Using PL-Resolution, we have to show $KB \wedge \neg\alpha$ unsatisfiable
    - Get negative of query ($\neg\alpha$), add to temporary KB
    - Continuously loops through:
        - Resolve every pair temp KB
        - At the end of a loop
            - If there is no new clause generated, return *False*
            - Else:
                - Check if resolvents contains empty clause or not, if yes, return *True*. If no, continue the loop

---

Discussion

- Pseudo-code:

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
    for each pair of clauses C_i, C_j in clauses do
      resolvents ← PL-RESOLVE(C_i, C_j)
      if resolvents contains the empty clause then return true
      new ← new ∪ resolvents
    if new ⊆ clauses then return false
    clauses ← clauses ∪ new
```

- As we can see, PL-Resolution will loop through every pair of clauses. If $KB \wedge \neg\alpha$ is large (for example: n), we have $nC2$ clauses. Even a medium sized problem the number of resolvents increases rapidly

  $\rightarrow$ In programming, it may take exponent time to solve PL-Resolution

- Some methods are needed to optimize this algorithm: Reduce the size of KB and alpha
  - Checking complementary clauses
  - Preprocessing clauses
  - Checking tautologies
  - Subsumptions (Backward & Forward)
  - Selecting the shortest usable clause

References

- https://www.doc.ic.ac.uk/~kb/MACTHINGS/SLIDES/2013Notes/6LSub4up13.pdf
- https://fossies.org/linux/sympy/sympy/logic/boolalg.py