ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO LAB 02
Logic Resolution

Môn: Cơ sở Trí tuệ nhân tạo - CSC14003

Lớp: 21_22

Tên: Lê Nguyễn

Mã số sinh viên: 21120511

Thành phố Hồ Chí Minh - 2023

Mục lục

1 Hoàn thành công việc **2**

2 Hiểu về source code **2**
2.1 Xử lý dữ liệu input 2
2.2 Xử lý $\neg\alpha$ 3
2.3 Xử lý thuật toán Resolution 6
2.4 Xử lý output 8

3 Các test case **9**
3.1 Test case 1 9
3.2 Test case 2 9
3.3 Test case 3 11
3.4 Test case 4 12
3.5 Test case 5 12

1 Hoàn thành công việc

Bảng 1: Bảng công việc

Tên công việc	Mức độ hoàn thành
Xử lý dữ liệu input	100%
Lưu dữ liệu vào cấu trúc dữ liệu phù hợp	100%
Đưa ra 5 test cases (không quá dễ)	100%
File output tuân theo cấu trúc được đưa ra trong assigment	100%
Áp dụng được thuật toán resolution (Davis-Putnam)	100%

2 Hiểu về source code

2.1 Xử lý dữ liệu input

Để dễ dàng thao tác với input (ở đây là các clause) ta sẽ lưu dưới dạng `List` thay vì `String`. Do các input ở dạng CNF và không có AND (trừ α clause) và NOT đều được đưa về tối giản, ví dụ: `"-A AND -B"` là tối giản còn `"-(A OR B)"` chưa tối giản. Vì vậy ta có thể lưu như sau: `"-A OR B OR C"` sẽ được lưu thành `["-A", "B", "C"]`, đồng thời ta cũng sort lại các *literal* (đơn giản là sort `List`).

Ví dụ: ta có input như sau:

```
-A OR B
-C OR B
A OR C OR -B
-B
```

sẽ được lưu trữ thành:

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

Đối với α clause, ta sẽ mở rộng ra hơn, thay vì chỉ có duy nhất 1 clause, α có thể có nhiều clause nối liền với nhau, ví dụ: `"(-A OR B) AND (-B OR C)"` là một input α hợp lệ còn `"(A OR B) AND -(B OR C)"` là một input α không hợp lệ. Để xử lý α đầu tiên ta sẽ tách AND sau đó đến tách OR.

Ví dụ: ta có α như sau:

```
(-A OR B) AND (-B OR C)
```

sẽ được lưu trữ thành:

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

Ngoài ra, theo đề bài, ta sẽ xử lý thêm trường hợp giữa các *literal* (`"A"`, `"-B"`, ...) và *operator* (`"AND"`, `"OR"`) có nhiều hơn một khoảng trắng, lúc này khi đọc file ta sẽ tách `String` theo khoảng trắng và sau đó gộp lại:

```
# input bị sai format
input = "-A       OR                    B"
# tách input dựa trên khoảng trắng
split = ["-A", "OR", "B"]
# gộp lại bằng 1 khoảng trắng
format = "-A OR B"
```

Vậy đối với input như sau:

```
-A
4
-A OR          B
-C          OR B
A OR    C   OR -B
-B
```

sẽ được lưu trữ thành:

```
alpha = [["-A"]]
KB = [
    ["-A", "B"],
    ["B", "-C"],
    ["A", "-B", "C"],
    ["-B"]
]
```

Hai hàm `read_input()` và `__format_input()` sẽ thực hiện các công việc trên.

2.2 Xử lý $\neg\alpha$

Dựa theo quy tắc De Morgan, ta biết:

$$\neg(A \vee B) = \neg A \wedge \neg B$$

Vì vậy để xử lý $\neg\alpha$, ta thay những literal `"A"` thành `"-A"` hoặc ngược lại. Lấy ví dụ $\alpha = (A \vee B) \wedge (\neg C \vee \neg D)$ hay `alpha = [["A", "B"], ["-C", "-D"]]`. Chạy hàm `__negate_clause()` ta sẽ có kết quả `__negate_clause(alpha) = [["-A", "-B"], ["C", "D"]]` lúc này ta hiểu rằng các `List` con bên trong sẽ được nối bởi `"AND"` và bên ngoài là `"OR"` thay vì ngược lại như ở lúc đầu ta xử lý input.

Tiếp theo dựa vào phép phân phối, ta sẽ phân phối $\wedge$ trên $\vee$, dựa vào ví dụ trên, ta có được:

$$\begin{aligned}
&(\neg A \wedge \neg B) \vee (C \wedge D) \\
=&(\neg A \vee (C \wedge D)) \wedge (\neg B \vee (C \wedge D)) \\
=&(\neg A \vee C) \wedge (\neg A \vee D) \wedge (\neg B \vee C) \wedge (\neg B \vee D)
\end{aligned}$$

Biểu diễn dưới dạng `List`, ta được:

```
[["-A", "C"], ["-A", "D"], ["-B", "C"], ["-B", "D"]]
```

Vậy làm sao để chuyển từ `[["-A", "-B"], ["C", "D"]]` thành được như trên ? Ta sẽ dùng hàm `__distributeAndOr()`. Để hiểu rõ hơn cách hàm `__distributeAndOr()` hoạt động ta sẽ dùng ví dụ sau:

```
# thực hiện phân phối
[["A", "B"], ["C", "D"], ["E", "F"]]
```

- Đầu tiên, ta chia biểu thức trên thành 2 phần:

  ```
  LHS = ["A", "B"]
  RHS = __distributeAndOr([["C", "D"], ["E", "F"]])
  ```

  trong đó `LHS` là phần bên trái (phần tử đầu tiên của biểu thức) còn `RHS` là phần bên phải của biểu thức. Ý tưởng phía trên được hiểu như sau:

  $$\begin{aligned}&(A \land B) \lor (C \land D) \lor (E \land F)\\ =&(A \land B) \lor ((C \land D) \lor (E \land F))\end{aligned}$$

  Ta sẽ thực hiện trong ngoặc trước, và thực hiện trong ngoặc ta lại tiếp tục phân phối lần nữa, vì vậy ta tiếp tục chạy đệ quy cho vế phải (`RHS`). Ta chạy cho đến khi `RHS` chỉ còn 1 phần tử, vậy sẽ dừng đệ quy khi:

  ```
  LHS = ["C, D"]
  RHS = ["E, F"]
  ```

- Bước tiếp theo, ta sẽ tiến hành phân phối từng phần tử của `LHS` cho từng phần tử của `RHS`, ý tưởng sẽ được hiểu như sau:

  $$\begin{aligned}&(C \land D) \lor (E \land F)\\ =&(C \lor (E \land F)) \land (D \lor (E \land F))\end{aligned}$$

  Các biểu thức ví dụ như $C \lor (E \land F)$ là các biểu thức tối giản và dễ để phân phối bằng cách kết hợp C với E, tiếp tục kết hợp C với F, vậy ta sẽ dùng hàm `__distributeAtom()` để làm việc đó. Nếu ta đưa input là `"C"` và `"D"` thì nó trả về `["C", "D"]` bởi vì nó hiểu là $C \lor D$, còn nếu đưa input là `"C"` và `["E", "F"]` nó sẽ trả về `["C", "E", "F"]` bởi vì nó hiểu là $C \lor (E \lor F)$. Tương tự như vậy, ta được kết quả khi gọi `__distributeAtom()` để kết hợp từng phần tử của `LHS` với từng phần tử của `RHS`, ta được:

  ```
  # lần chạy thứ nhất
  result = [["C", "E"], ["C", "F"]]
  # lần chạy thứ hai, do LHS chỉ có 2 phần tử là "C", "D"
  result = [["D", "E"], ["D", "F"]]
  ```

- Sau đó ta kết hợp các kết quả phân phối và `__distributeAndOr()` sẽ trả về:

  ```
  # RHS = ["C", "E"], ["C", "F"], ["D", "E"], ["D", "F"]
  __distributeAndOr([["C", "D"], ["E", "F"]]) = [
      ["C", "E"], ["C", "F"], ["D", "E"], ["D", "F"]
  ]
  ```

- Tiếp tục thực hiện phân phối từng phần tử của `LHS = ["A", "B"]` cho từng phần tử `RHS`. Cuối cùng ta được:

```
[
    ["A", "C", "E"],
    ["A", "C", "F"],
    ["A", "D", "E"],
    ["A", "D", "F"],
    ["B", "C", "E"],
    ["B", "C", "F"],
    ["B", "D", "E"],
    ["B", "D", "F"],
]
```

- Ta có thể tóm gọn lại code như sau:

```python
def __distributeAtom(literal, clause):
    if type(clause) == str:
        return [literal, clause]
    else:
        return [literal] + clause

def __distributeAndOr(clause):
    if len(clause) == 1:
        return clause[0]

    lhs = clause[0]
    rhs = self.__distributeAndOr(clause[1:])

    result = []
    for i in range(0, len(lhs)):
        for j in range(0, len(rhs)):
            result.append(__distributeAtom(lhs[i], rhs[j]))

    return result
```

Vậy là ta đã thực hiện thành công phân phối giữa `"AND"` và `"OR"` khi thực hiện $\neg\alpha$. Thế nhưng chúng ta vẫn còn hai vấn đề cần giải quyết.

- Vấn đề đầu tiên là có các literal bị trùng nhau sau khi ta thực hiện phân phối, vì vậy cần loại bỏ các literal bị trùng. Ví dụ như:

$$\begin{aligned} & A \vee A \vee B \\ =& A \vee B \end{aligned}$$

  Để loại bỏ (thuật ngữ gọi là *factoring*) ta dùng hàm `__factoring()`. Cơ chế cũng rất đơn giản, ta chỉ cần đưa `List` về `Set` thì python sẽ tự động loại bỏ phần tử trùng nhau, thế nhưng nó sẽ bị mất đi thứ tự, vì vậy trước khi trả về, ta sẽ tiến hành sort lại các literal.

- Thứ hai là vấn đề về **valid clause**, valid clause là những clause luôn mang giá trị True, ví dụ như:

$$\begin{aligned} & A \vee (\neg A) \vee B \\ =& True \vee B \\ =& True \end{aligned}$$

vì vậy nó sẽ không có ý nghĩa gì nếu ta thêm vào KB, thế nên ta phải loại nó đi sau khi thực hiện phân phối. Ta sử dụng hàm `__is_valid_clause()` để check xem một clause có là valid clause hay không, hàm này đơn giản chỉ check xem trong clause có các literal trái dấu nhau không, nếu có sẽ trả về `True`. Cứ mỗi lần hàm `__is_valid_clause()` trả về `False` thì ta sẽ thêm clause tương ứng vào `List` để ta sử dụng, còn lại sẽ bỏ đi.

Thế là ta đã thực hiện xong việc $\neg\alpha$, đây là việc quan trọng nhất vì ta cần nó đúng để thuật toán Resolution chạy đúng.

2.3 Xử lý thuật toán Resolution

Thuật toán sẽ chủ yếu dựa vào **Luật hợp giải** (*resolution rule*), ta có thể hiểu như sau:

$$\frac{a_1 \vee a_2 \vee ... \vee c \qquad b_1 \vee b_2 \vee ... \vee (\neg c)}{a_1 \vee a_2 \vee ... \vee b_1 \vee b_2 \vee ...}$$

Đơn giản là khi hợp giải 2 clause lại với nhau, các literal trái dấu nhau sẽ bị loại bỏ, còn lại sẽ được nối rời với nhau để tạo thành clause mới. Thế nhưng ta phải chú ý điều này, ví dụ cho 2 câu sau:

$$\begin{aligned} & A \vee (\neg B) \vee C \\ \text{và } & (\neg A) \vee B \vee C \end{aligned}$$

Giả sử ta sẽ hợp giải 2 câu trên theo A và $\neg A$ trước, ta sẽ được $(\neg B) \vee B \vee C = True$ là một câu hợp lệ, hoặc ta hợp giải 2 câu trên theo B và $\neg B$ trước, ta sẽ được $(\neg A) \vee A \vee C = True$ cũng là một câu hợp lệ, vậy 2 câu trên không hợp giải được.

Do đó, ta phải thêm điều kiện vào, hai câu chỉ hợp giải được khi và chỉ khi nó chỉ có duy nhất 1 cặp literal trái dấu nhau, ta sẽ check điều kiện này trong hàm `__can_resolution()`. Nếu điều kiện thoả mãn hay `__can_resolution()` trả về `True`, ta sẽ áp dụng luật hợp giải này bằng hàm `__apply_resolution()` để trả về kết quả sau khi hợp giải.

Giờ ta sẽ đến với thuật toán hợp giải, thuật toán hợp giải chính là hàm `pl_resolution()`. Để hiểu rõ cách hoạt động ta sẽ tìm hiểu ví dụ sau:

```
alpha = [["-A"]]
KB = [
    ["-A", "B"],
    ["B", "-C"],
    ["A", "-B", "C"],
    ["-B"]
]
```

- Đầu tiên, ta $\neg\alpha$, ta được:

```
negate_alpha = [["A"]]
```

- Tiếp theo, ta thêm các clause trong $\neg\alpha$ vào KB, ta được:

```
KB = [
    ["-A", "B"],
    ["B", "-C"],
    ["A", "-B", "C"],
    ["-B"],
    ["A"]
]
```

  Lưu ý, khi thêm clause vào KB, ta phải check xem clause đó đã có mặt trong KB chưa, nếu chưa thì ta sẽ thêm vào, việc này giúp tránh được các clause bị trùng nhau.

- Ở giai đoạn tiếp theo, với mỗi clause trong KB, ta sẽ hợp giải với những câu còn lại trong KB sinh ra clause mới, sau đó thêm các câu mới vào lại KB và chạy đến lần lặp tiếp theo. Ta chạy đến khi nào ta hợp giải ra rỗng (tức là ra kết quả KB có thể entail được α) hoặc trong một lần lặp nào đó, không có một clause mới nào được sinh ra (nghĩa là KB không thể entail được α).

  – Ở lần lặp đầu tiên, ta cho `["-A", "B"]` hợp giải với các câu còn lại trong KB, ta thấy hợp giải được với `["-B"]` và sinh ra `["-A"]` và ta đưa câu mới vào một `List` phụ, gọi là `temp`. Đi tiếp ta thấy `["-A", "B"]` hợp giải được với `["A"]` sinh ra `["B"]`, tiếp tục thêm vào `temp`. Chọn tiếp `["B", "-C"]` ta hợp giải được với `["-B"]` cho ra `["-C"]` và tiếp tục thêm vào `temp`. Còn `["A", "-B", "C"]`, `["-B"]` và `["A"]` không thể hợp giải với các câu nào nữa. Do đó ta được kết quả hợp giải sau lần lặp đầu tiên là:

```
temp = [["-A"], ["B"], ["-C"]]
```

  – Cuối cùng ta thêm kết quả của lần lặp đầu tiên vào KB, lúc này KB trở thành:

```
KB = [
    ["-A", "B"],
    ["B", "-C"],
    ["A", "-B", "C"],
    ["-B"],
    ["A"],
    ["-A"],
    ["B"],
    ["-C"]
]
```

  Khi đó ta sẽ bước qua lần lặp thứ 2, cũng lặp lại như trên, cho mỗi câu trong KB hợp giải với các câu còn lại để cho ra một `List temp`, lúc ta được `temp` là:

```
temp = [["-B", "C"], ["A", "C"], ["A", "-B"], []]
```

  Lúc này ta thấy một `List` rỗng, tức là đã hợp giải thành công và ta `break` vòng lặp này để thoát ra và cho ra giá trị `True`.

Ta có thể tóm gọn lại bằng code sau:

```
def pl_resolution(self):
    neg_alpha = self.negate(self.alpha)
    axioms = self.KB

    # add negate alpha to KB
    for arg in neg_alpha:
        if arg not in axioms:
            axioms.append(arg)

    # start resolution algorithm
    while True:
        # before the first loop, output_clauses has no generated clause
        self.output_clauses.append([])
        # resolve each clause in axioms
        for i in range(0, len(axioms)):
            for j in range(0, len(axioms)):
                if self.__can_resolution(axioms[i], axioms[j]):
                    resolve = self.__apply_resolution(axioms[i], axioms[j])
                    if (resolve not in self.output_clauses[-1]
                        and resolve not in axioms):
                        # append resolve to ouput_clause
                        self.output_clauses[-1].append(resolve)
                    # add all resolve clause to axioms (or KB)
                    if len(resolve) == 0:
                        self.can_entail = True
                        return
        # if we went through all iterations
        # but we couldn't derive any new sentences
        # then KB cannot entail alpha
        if len(self.output_clauses[-1]) == 0:
            self.can_entail = False
            return
        # add all resolve clause to axioms (or KB)
        axioms += self.output_clauses[-1]
```

Đánh giá một chút về cách giải quyết này, ta thấy phải chạy rất lâu bởi vì ta cần 3 vòng lặp để giải quyết, vậy độ phức tạp có thể lên đến $O(mn^2)$ với n là số clause trong KB cộng với α và m là số lần mà ta phải chạy lại thuật toán trên KB mới. Thế nhưng nó là cách dễ hiểu và dễ thực hiện nhất, thế nên trong các trường hợp KB và α không quá lớn, thì cách này sẽ phù hợp hơn. Thuật toán này gần giống với **Thuật toán Davis Putnam**, để tối ưu hơn nữa ta có thể dùng **Thuật toán Davis–Putnam–Logemann–Loveland (DPLL)**.

2.4 Xử lý output

Ta sẽ xử lý output khá là đơn giản, ta tạo một biến toàn cục gọi là `self.output_clause`, `self.output_clause` sẽ chứa kết quả hợp giải (đơn giản hơn là cứ mỗi lần lặp, ta thêm `temp` vào `self.output_clause`) của những lần lặp trong quá trình resolution. Khi chạy xong thuật

toán resolution, ta chỉ cần lặp hết các phần tử của `self.output_clause` để in ra file output. Ngoài ra để biết được sau quá trình resolution, ta có entail được α hay không, ta dùng biến `self.can_entail`, nếu là `True` ta in ra **YES**, ngược lại in ra **NO**.

3 Các test case

3.1 Test case 1

Input:

```
(A OR B) AND (-A OR C) AND (A OR -C OR -D) AND B
5
-A OR C OR D
-A OR B OR -C
B OR D
B OR -C OR D
A OR D
```

Output:

```
9
-A OR B OR D
C OR D
-A OR -B OR D
-B OR C OR D
-A OR -C
-A OR -C OR D
A OR C OR D
-C OR D
A OR -C OR D
5
-A OR D
A OR B OR D
-B OR D
B OR C OR D
D
0
NO
```

3.2 Test case 2

Input:

```
(A OR B OR D) AND (B OR -C OR D)
5
-A OR B OR -C
-B OR C OR D
A OR D
B OR C
-C OR A OR D
```

Output:

```
17
B OR -C OR D
-A OR B
-A OR -C
-A OR -C OR -D
-A OR B OR -D
C OR D
A OR -B OR D
-A OR -B OR C
-B OR D
A OR -B
A OR C
A
A OR B OR D
C
-B OR -C OR D
A OR -B OR -C
A OR -C
28
-A OR B OR D
-A OR -C OR D
B OR -C
-A OR C OR D
-A OR -B OR D
A OR C OR D
B OR D
-C OR D
A OR B
-A
-B OR -C
B OR C OR D
-A OR -B OR -C
-B OR -C OR -D
-C OR -D
-A OR -B OR -D
-A OR C OR -D
A OR -B OR C
A OR B OR C
A OR -B OR -D
A OR -D
-A OR D
B
-C
-A OR B OR C
B OR C OR -D
B OR -D
B OR -C OR -D
```

```
5
D
A OR B OR -C
A OR C OR -D
-B OR C OR -D
{}
YES
```

3.3 Test case 3

Input:

```
-A AND D AND (C OR -B)
4
-A OR B OR -D
B OR -C
A OR -B OR C
-B OR D
```

Output:

```
6
B OR -D
B OR -C OR -D
-C OR D
A OR C OR -D
A OR -B OR -D
A OR -B OR -C
7
-A OR B OR -C
B OR C OR -D
A OR -C
A OR -B OR D
A OR -B
A OR B OR -C
A OR -D
2
A OR -C OR D
-A OR -C OR D
2
-B OR -C OR D
B OR -C OR D
0
NO
```

3.4 Test case 4

Input:

```
-A AND (C OR B)
6
-A OR B OR -C
B OR -C
-A OR -B OR D
-B OR D
B
-D
```

Output:

```
8
-A OR -C OR D
-C OR D
-A OR D
-A OR -B
-B OR -C OR D
D
-B
A
4
-A OR -C
-C
-A
{}
YES
```

3.5 Test case 5

Input:

```
C OR -D
4
-A OR B OR -D
B OR -C
A OR -B OR C
-B OR D
```

Output:

```
3
-A OR B
-C OR D
A OR -B
4
-A OR B OR -C
A OR -C
```

```
A OR -B OR D
-A OR D
4
B OR -C OR -D
A OR -C OR D
-B OR C OR D
-A OR -C OR D
6
B OR -C OR D
-A OR C OR D
A OR -C OR -D
-B OR -C OR D
A OR B OR -C
-A OR -B OR D
4
-A OR B OR C
-A OR B OR D
A OR -B OR -D
A OR -B OR -C
0
NO
```