Resolution_FOL

> Name: Xi Peng
>
> ID   : 179084

Project Structure

There are totally 7 python files in my project:

FOL.py

CNF.py

preprocess.py

MGU.py

Resolution.py

runner.py

utils.py

I will briefly introduce the function of these files:

- **FOL.py**

  Define the basic data type of these project:

  - [ ] Quantity
  - [ ] Operation
  - [ ] Predicate
  - [ ] Function
  - [ ] Constant
  - [ ] Node

  All the type extend from Node (except node), because I want to transfer the sentence to a binary tree.

- **CNF.py preprocess.py**

  Convert the sentence string to a binary tree, then convert to CNF. The sentence finally will be a 2d list, like [[p1,p2],[p3,p4...],...] p means predicate. The binary tree is the intermediate products. sentence -> binary tree -> 2d list.

- **MGU.py**

  Do the most general unify.

- **Resolution.py**

  Do the resolution.

- **uitls.py**

  Some useful method like print the list, print a tree...

The **input** and **MGU_testCase** file contains some input sentences or test case for me to debug.

Sigma Structure

Input formate

quantifiers: exist, forAll

operators: and, or, implies

negation: not

I use prefix notation like:

```
forAll x forAll y ( implies ( and ( B ( x ) , S ( y , y ) ) , not S ( x , y ) )
)
```

It means:

B(x) & S (y, y) => -S(y,y)

all the operators are binary operator, so there is no and(P(x),P(x),P(x)), just have and(and(P(x),P(x)),P(x)).

There should be a space for each characters in the input sentence.

```
valid    and ( B ( x ) , S ( y , y ) )
invalid  and(B(x), S(y,y)) #because not all characters has a space between each
other
```

Define Sigma Structure

I use four list to store the signature:

variables, predicates, constants, functions

```
self.variables = VARIABLES
self.predicates = PREDICATES
self.constants = CONSTANTS
self.functions = FUNCTIONS
```

You can define the signature manually or use the pre-defined signature.

By manually:

```
if you want to define sigma structure enter 1, otherwise enter any key
1
enter predicate, press 'end' to end
P
Q
end
enter function, press 'end' to end
f
g
end
enter variable, press 'end' to end
x
y
x
end
enter constant, press 'end' to end
a
b
end
```

For pre-defined signature, I use the a-m as constant, n-z as variable, A-M as function, N-Z as predicate.

If the sentence contains symbols that do not belong to signature, it will raise a error:

```
AssertionError: undefined symbol
```

CNF

```
sentence = sentence.split()

sentence = removeImplies(sentence)

sentence = pushNegation(sentence, fol_engine)

sentence = VarConst2Node(sentence, fol_engine)

sentence = defineScope(sentence, fol_engine)

rootNode = All2Node(sentence, fol_engine)

skolemization(rootNode, fol_engine)

convert2CNF(rootNode)
```

After these several steps, the sentence will change to a biniary tree, only the leaf nodes will be the predicate, other nodes will be the 'and' or 'or' operators, the child nodes of 'or' operator has no 'and' node.

I do the redundancy elimination, p & p and p | p will be p

You can run the **CNF.py** module individually to see the results.

```
forAll x ( implies ( F ( x ) , L ( j , x ) ) )
-F ( x_1 ) L ( j x_1 )
```

```
and ( and ( and ( or ( P ( x ) , P ( x ) ) , Q ( x ) ) , Z ( x ) ) , P ( x ) )
P ( x )
Z ( x )
Q ( x )
```

This is the case do the Skolemization:

```
not exist x ( implies ( P ( x ) , forAll y  P ( y ) ) )
P ( x_1 )
-P ( F_16 ( x_1 ) )
```

MGU

Implement it by find the conflict set and unify set.

You can run the **MGU.py** module individually to see the results.

```
input one predicate
P ( y , f ( x , y ) )
input another predicate
P ( g ( z ) , f ( a , z ) )
no unifiable
```

```
input one predicate
P ( y , f ( x , y ) )
input another predicate
P ( b , f ( a , y ) )
P ( b f ( a b ) ) , y => b; x => a;
```

```
input one predicate
P ( g ( x ) , z )
input another predicate
P ( g ( y ) , g ( z ) )
no unifiable
```

```
input one predicate
P ( x , g ( y ) )
input another predicate
P ( g ( y ) , x )
P ( g ( y ) g ( y ) ) , x => g ( y );
```

Resolution

Implement it by BFS, use the query and one clause from KB to check if it can resolve to a new clause, if it not in KB, add to KB, if it is [], return True. If there is no new clause produced or the steps exceed the setting, return False.

User can set the max_loops parameter.

You can run the **runner.py** module individually to see the results.

1. A barber shaves all persons who do not shave themselves.

```
forAll x forAll y ( implies ( and ( B ( x ) , not S ( y , y ) ) , S ( x , y ) ) )
```

2. No barber shaves someone who shaves himself.

```
forAll x forAll y ( implies ( and ( B ( x ) , S ( y , y ) ) , not S ( x , y ) ) )
```

3. There is no barber.

```
forAll x ( not B ( x ) )
```

Show using your algorithm that (3) is a consequence of (1) and (2).

```
if you want to define sigma structure enter 1, otherwise enter any key
1
enter predicate, press 'end' to end
B
S
end
enter function, press 'end' to end
end
enter variable, press 'end' to end
x
y
end
enter constant, press 'end' to end
a
b
end
input KB size:
2
input sentence:
forAll x forAll y ( implies ( and ( B ( x ) , not S ( y , y ) ) , S ( x , y ) ) )
input sentence:
forAll x forAll y ( implies ( and ( B ( x ) , S ( y , y ) ) , not S ( x , y ) ) )
input query:
forAll x ( not B ( x ) )
True
```

other cases:

```
if you want to define sigma structure enter 1, otherwise enter any key
0
input KB size:
0
input query:
implies ( exist x forAll y P ( x , y ) , forAll y exist x P ( x , y )  )
True
```

```
if you want to define sigma structure enter 1, otherwise enter any key
0
input KB size:
0
input query:
implies (  forAll y exist x P ( x , y ) , exist x forAll y P ( x , y ) )
False
```