CS6770 KRR Programming Assignment

Problem statement :

Group D - 8.ALC Taxonomy Builder - Subsumption Hierarchy

Team Number : 2
Team Members: Hanumantappa Budihal (CS21M022)
Pratyush Dash (CS21M046)

Introduction :

Logic-based knowledge representations implement mathematical logic, a subfield of mathematics dealing with formal expressions, reasoning, and formal proof.

Description logics (DL) constitute a family of formal knowledge representation languages that provide a logical underpinning for OWL ontologies. Many description logics are more expressive than propositional logic, which deals with declarative propositions and does not use quantifiers, and more efficient in decision problems than first-order predicate logic, which uses predicates and quantified variables over nonlogical objects.

A crucial design principle in description logic is to establish favorable trade-offs between expressivity and computational complexity to suit different applications. The expressivity of each description logic is determined by the supported mathematical constructors. There are crisp general-purpose DLs, spatial, temporal, and spatiotemporal DLs, probabilistic and possibilistic DLs, and fuzzy DLs.

ALC :

The Attributive Language (AL) family of description logics allows atomic negation, concept intersection, universal restrictions, and limited existential quantification. A core AL-based description logic is the Attributive (Concept) Language with Complements (ALC), in which, unlike AL, the complement of any concept is allowed, not only the complement of atomic concepts.

ALC concept expressions can include concept names, concept intersection, concept union, complement, existential and universal quantifiers, and individual names.

Subsumption Hierarchy:

The subsumption algorithm determines subconcept-superconcept relationships: a concept C is subsumed by a concept D w.r.t. a TBox if, in each model of the TBox, each instance of C is also an instance of D. Such an algorithm can be used to compute the taxonomy of a TBox, i.e., the subsumption hierarchy of all those concepts introduced in the TBox.

Problem statement :

Implement a taxonomy builder. Given an ALC KB in Negation Normal Form generated by assignment 6, reuse the ALC Tableau implementation from assignment 7 and write a program to build the taxonomy (subsumption hierarchy) of concepts defined in the KB.

Implementation :

1. **Input file read** : used **xml.etree.ElementTree** library to read the xml file and return the tree value.

2. Create a **dictionary** which will have class names as **keys** and the super classes of the corresponding classes are the **values** for that particular key.
   Dictionary = {key : value} = {class : set of super classes of this class}

3. Iterate all the class tag members in the input file and get the child of each **“Class”** tag.

```
for child in root.findall("Class"):
```

   A “Class” tag can have three kinds of child tags - “CONCEPT”, “SubClassOf” and “EquivalentTo”

4. The “CONCEPT” tag, which is immediate children of “Class” tags, will give the name of the class.
   Simply extract the name of the class and add it to the dictionary as a key and empty set as its value.

```
key = child.find('CONCEPT').text  # get the class name as key
set_data = set()                  # get the subclasses in a set
```

5. The “SubClassOf” tag contains the super classes of the current class.
   Idea is to list all the classes inside this tag by getting the “CONCEPT” tags inside this and add them to the set.

   Further, a “SubClassOf” tag can have “NOT”, “EXISTS”, “AND” tags which can be further traversed down to get the “CONCEPTS” tag within them and add these concepts with required keywords to the set

```
# handle  SubClassOf part
for newChild in child.findall('SubClassOf'):

    #Get the all sub class values
    if(newChild.find('CONCEPT') != None):
        set_data.add(newChild.find('CONCEPT').text)

    #handle the Not subclass case
    if(newChild.find('NOT') != None):
        for notChild in newChild.findall('NOT'):
            set_data.add("not "+notChild.find('CONCEPT').text)

    #handle the EXISTS case
    if(newChild.find('EXISTS') != None):
        set_data.add(newChild.find('ROLE').text +
                     " some " + newChild.find('CONCEPT').text)

    #       You, 50 minutes ago • added the comments …
    if(newChild.find('AND') != None):
        for newChild in child.findall('EquivalentTo'):
            if(newChild.find('CONCEPT') != None):
                set_data.add(newChild.find('CONCEPT').text)

            if(newChild.find('NOT') != None):
                for notChild in newChild.findall('NOT'):
                    set_data.add("not "+notChild.find('CONCEPT').text)

            if(newChild.find('EXISTS') != None):
                set_data.add(newChild.find('ROLE').text +
                             " some " + newChild.find('CONCEPT').text)
```

6. The “EquivalentTo” tag contains the equivalent classes of the current class.
   Idea is to list all the classes inside this tag by getting the “CONCEPT” tags inside this and add them to the set.
   Further, a “EquivalentTo” tag can have “NOT”, “EXISTS”, “AND” tags which can be further traversed down to get the “CONCEPTS” tag within them and add these concepts with required keywords to the set

```
# handle Equivalent to part
for newChild in child.findall('EquivalentTo'):
    if(newChild.find('CONCEPT') != None):
        set_data.add(newChild.find('CONCEPT').text)

    if(newChild.find('NOT') != None):
        for notChild in newChild.findall('NOT'):
            set_data.add("not "+notChild.find('CONCEPT').text)

    if(newChild.find('EXISTS') != None):
        set_data.add(newChild.find('ROLE').text +
                     " some " + newChild.find('CONCEPT').text)

    if(newChild.find('AND') != None):
        for newnewChild in newChild.findall('AND'):
            if(newnewChild.find('CONCEPT') != None):
                set_data.add(newnewChild.find('CONCEPT').text)

            if(newnewChild.find('NOT') != None):
                for notChild in newnewChild.findall('NOT'):
                    set_data.add("not "+notChild.find('CONCEPT').text)

            if(newnewChild.find('EXISTS') != None):
                set_data.add(newnewChild.find('EXISTS').find(
                    'ROLE').text+" some " + newnewChild.find('EXISTS').find('CONCEPT').text)
```

7. Assign the set used above to the respective key/“Class”.

8. Handle the transitivity : this part of the code handles the transitivity part of the subsumption hierarchy.
   This can be done iteratively by adding the superclass of the super class of the current class to the set in the dictionary where the key is the current class and the value is the set of the current class.

```
def handle_transitivity(dictionary):
    """
    handle the transitivity of the sub class
    """
    # for transitivity
    dic = {}
    while(dic != dictionary):
        dic = dictionary.copy()
        for key in dictionary:
            news = dictionary[key]
            for item in dictionary[key]:
                if item in dictionary:
                    for i in dictionary[item]:
                        if i in dictionary:
                            news = news.union({i})

            dictionary[key] = news
```

9. **write the output file :** write the data to the text output file.

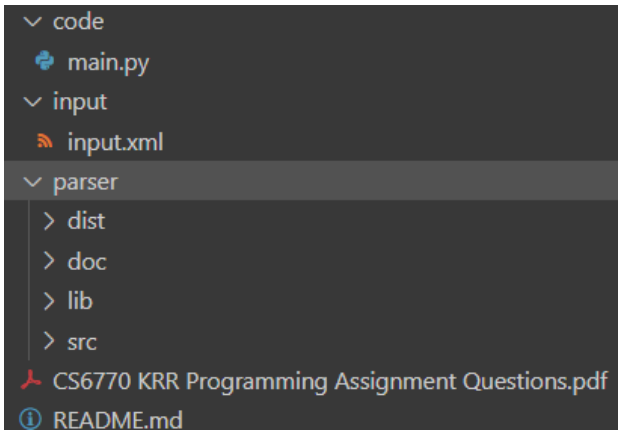
```
# open the text file with write permission
file = open(file_path, 'w')

# write the data of class and sub class data to the file
for key in data:
    file.write('Class: ' + key + '\n')
    set_data = data[key]
    list_data = list(set_data)

    if bool(set_data):
        file.write("\tSubClassOf: ")
        for item in list_data[:-1]:
            file.write(item+', ')

        file.write(list_data[-1])
        file.write("\n")
```

Code structure :



Code : This folder contains the main implementation of assignment. Code will take the ALC KB in Negation Normal Form (xml format) which generates the build the taxonomy (subsumption hierarchy) of concepts defined in the KB. Output file is exported to **parser\src\krr\test\output** which is used to create the output in xml file format.

Input : Contains the ALC KB in Negation Normal Form (xml files)

Parser : KRR software package which converts the text(output from assignment implementation) file to xml format.

Execution :

1. Read xml file and create the output file (.txt format)

Navigate to Code folder : code->main.py and run the below command.

Command : **python main.py**

2. Generate the output xml file (using KRR Package)

Generate the output file using the KRR package :

Navigate to the folder **parser\src\krr\test** folder and run the below command.

Command : **./run.sh**

Input and Output :

Input :

```
37      <ObjectProperty>
38          <ROLE>teaches</ROLE>
39          <Domain>
40              <CONCEPT>Person</CONCEPT>
41          </Domain>
42      </ObjectProperty>
43      <Class>
44          <CONCEPT>Student</CONCEPT>
45          <EquivalentTo>
46              <AND>
47                  <CONCEPT>Person</CONCEPT>
48                  <EXISTS>
49                      <ROLE>attends</ROLE>
50                      <CONCEPT>Course</CONCEPT>
51                  </EXISTS>
52              </AND>
53          </EquivalentTo>
54      </Class>
55      <ObjectProperty>
56          <ROLE>attends</ROLE>
57          <Domain>
58              <CONCEPT>Person</CONCEPT>
59          </Domain>
60      </ObjectProperty>
```

Output :

```
<Class>
    <CONCEPT>Teacher</CONCEPT>
    <SubClassOf>
        <EXISTS>
            <ROLE>teaches</ROLE>
            <CONCEPT>Course</CONCEPT>
        </EXISTS>
        <CONCEPT>Person</CONCEPT>
    </SubClassOf>
</Class>
<Class>
    <CONCEPT>Student</CONCEPT>
    <SubClassOf>
        <EXISTS>
            <ROLE>attends</ROLE>
            <CONCEPT>Course</CONCEPT>
        </EXISTS>
        <CONCEPT>Person</CONCEPT>
    </SubClassOf>
```