CS551K – Software Agents and Multi-Agent Systems

Assessment Item 3 – Task 2

2. (Formal Argumentation) In this part of the assessment, we will create an end-to-end system that can parse files and reason with arguments. You will need to provide a report (maximum 4 pages) containing the answers to each of the items below. You will need to clearly indicate in your report what each part refers to. If we cannot work out which item that parts of your submission refer to then you may miss marks. The code files for each question should also be attached to your submission.

The ASPARTIX format allows one to represent Abstract Argumentation Frameworks using a syntax with only two predicates ("arg "and "att"). In the example below, we show an argumentation graph with the corresponding ASPARTIX input. The statement "arg(a)" declares the argument "a" whereas "att(a,b)" declares that argument "a" attacks argument "b". Each statement must end with a period "." and argument names should only be composed of alphanumeric characters "a" to "z", "A" to "Z", and "0" to "9".

a. Design a solution to parse ASPARTIX files and provide the corresponding pseudo-code/code-snippets as well as explanation on how your solution works in the report.
(2.5 marks)

Answer:
For this task I am using class Apx_parser, which takes as an argument filepath parameter. The 'read_file' method line by line converts the text from the file into list of arguments and list of attacks.The code snippet can be found in Appendix 1.

b. Implement a Python program **aspartix_parser.py** containing a function **read_file** that will take as input the path to a file in the ASPARTIX format and return a pair consisting of a set of strings (corresponding to arguments) and a set of pairs (corresponding to attacks) with the first argument assumed to attack the second. The **read_file** function should also print the number of arguments and attacks in your terminal. The box below shows the expected output of the **read_file** function on our example.
({"a", "b", "c", "d", "e", "f"}, {("a", "b"), ("b","a"), ("a","c"), ("b","c"), ("c", "d"),
("d","f"), …})
There are 6 arguments and 9 attacks.
You can use existing Python libraries such as the Regular Expression library or the Lark toolkit. In your report, show the output of your implementation on an input sample with 10 arguments and 5 attacks. **(2.5 marks)**

Answer:
The method and the function read_file described above. The output of the main file is:
['a', 'b', 'c', 'd', 'e', 'f'] [['a,b'], ['b,a'], ['a,c'], ['b,c'], ['c,d'], ['d,f'], ['e,d'], ['e,f'], ['f,e']]
There are 6 arguments and 9 attacks.
Code snippet for this task can be found in Appendix 1.

c. Design and implement solutions to compute the maximal (for set inclusion) conflict-free, admissible, preferred, and stable extensions. To answer this question, you will need to provide

the corresponding pseudo-code/code-snippets as well as explanation on how your solutions work in the report. **(5 marks)**

Answer:
The solution for each set implemented as the sequence of function calls. Functions correspond to each set required for the solution. Each subsequent function takes the output of the previous function as an input.
The "conflict_free" function takes as an input the data parsed from **aspartix_parser.py read_file** function, as two sets arguments and attacks. The function creates the set of all possible arguments from input 'arguments' set. Then it iterates the attacks set inside of the loop of different combinations. The aim is to find the combinations which will not contain any attacks between elements in the set. When the correct set found the function checking if this set is the subset of the set included into the list to derive only maximum sets. The combination sets are pre-sorted from biggest to smallest, which gives the possibility to include biggest sets first.
The 'admissible' function takes as an input the conflict_free set including all elements, not only maximum. This function checks if all elements are acceptable.
The preferred function takes as an input the set of admissible sets, which is not correct it has to be the set of complete extensions. The aim of preferred function is to derive the maxima sets from the input sets.
The idea of stable extension function is to derive output from preferred input sets. The output of stable extensions supposed to be the set of complete extensions where any argument which is not present in an extension is defeated by an argument in the extension.

d. Implement a Python program **semantics.py** that returns the maximal (for set inclusion) conflict-free, admissible, preferred, and stable semantics for a given argumentation graph represented as a file in the ASPARTIX format. Your implementation should be run from the command line as "**python3 semantics.py example.apx**" in the terminal. If, **example.apx** is the ASPARTIX file containing our argumentation graph.
Moreover, the arguments in the extensions and the extensions should be printed using the lexicographical order (**0-9** symbols appear before a-z symbols which in turn appear before A-Z symbols). For instance, the set {"**10**", "**9a**", "**a1**", "**b2**", "**A1**"} follows the lexicographical order whereas the set {"**9a**", "**10**", "**A1**", "**b2**", "**a1**"} does not. In the example of item 1 above, {"**e**", "**a**"} is a preferred extension and should be printed as {"**a**", "**e**"}. Please note that although you are not allowed to use existing code/libraries for this question, you can make use of the **read_file** function implemented in item 1. **(4.5 marks)**
Answer:
The implementation of the semantics.py can be found in appendix 2. The output is:
Conflict free extensions: [] [['a'], ['a', 'd'], ['a', 'e'], ['a', 'f'], ['b'], ['b', 'd'], ['b', 'e'], ['b', 'f'], ['c'], ['c', 'e'], ['c', 'f'], ['d'], ['e'], ['f']]
Admissible extensions: [] [['a'], ['a', 'e'], ['b'], ['b', 'e'], ['e']]
Preferred extensions: [['a', 'e'], ['b', 'e']]
Stable extensions: None
The arguments are sorted according to the conditions given.

e. In your report, show the output of your implementation on the five test files (test1.apx to test5.apx) provided in MyAberdeen. **(3 marks)**

Answer:
test1.apx output:

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'] [['c,a'], ['a,b'], ['b,c'], ['b,d'], ['d,e'], ['d,g'], ['h,e'], ['f,g'], ['g,f']]
There are 8 arguments and 9 attacks.
Conflict free extensions: [] [['a'], ['a', 'd'], ['a', 'd', 'f'], ['a', 'd', 'f', 'h'], ['a', 'd', 'h'], ['a', 'e'], ['a', 'e', 'f'], ['a', 'e', 'g'], ['a', 'f'], ['a', 'f', 'h'], ['a', 'g'], ['a', 'g', 'h'], ['a', 'h'], ['b'], ['b', 'e'], ['b', 'e', 'f'], ['b', 'e', 'g'], ['b', 'f'], ['b', 'f', 'h'], ['b', 'g'], ['b', 'g', 'h'], ['b', 'h'], ['c'], ['c', 'd'], ['c', 'd', 'f'], ['c', 'd', 'f', 'h'], ['c', 'd', 'h'], ['c', 'e'], ['c', 'e', 'f'], ['c', 'e', 'g'], ['c', 'f'], ['c', 'f', 'h'], ['c', 'g'], ['c', 'g', 'h'], ['c', 'h'], ['d'], ['d', 'f'], ['d', 'f', 'h'], ['d', 'h'], ['e'], ['e', 'f'], ['e', 'g'], ['f'], ['f', 'h'], ['g'], ['g', 'h'], ['h']]
Admissible extensions: [] [['a', 'd', 'f'], ['a', 'd', 'f', 'h'], ['d', 'f'], ['d', 'f', 'h'], ['f'], ['f', 'h'], ['h']]
Preferred extensions: [['a', 'd', 'f', 'h']]
Stable extensions: None

Test2.apx output:
['a', 'b', 'c', 'd', 'e'] [['d,a'], ['b,a'], ['e,d'], ['a,e'], ['b,c'], ['c,e']]
There are 5 arguments and 6 attacks.
Conflict free extensions: [] [['a'], ['a', 'c'], ['b'], ['b', 'd'], ['b', 'e'], ['c'], ['c', 'd'], ['d'], ['e']]
Admissible extensions: [] [['b'], ['b', 'e']]
Preferred extensions: [['b', 'e']]
Stable extensions: None

Test3.apx output:
['a', 'b', 'c', 'd'] [['a,b'], ['b,c'], ['c,d'], ['d,c']]
There are 4 arguments and 4 attacks.
Conflict free extensions: [] [['a'], ['a', 'c'], ['a', 'd'], ['b'], ['b', 'd'], ['c'], ['d']]
Admissible extensions: [] [['a'], ['a', 'c'], ['a', 'd'], ['b', 'd'], ['d']]
Preferred extensions: [['a', 'c'], ['a', 'd'], ['b', 'd']]
Stable extensions: None

Test4.apx output:
['a', 'b', 'c', 'd', 'e'] [['a,b'], ['b,a'], ['b,c'], ['c,d'], ['d,e'], ['e,c']]
There are 5 arguments and 6 attacks.
Conflict free extensions: [] [['a'], ['a', 'c'], ['a', 'd'], ['a', 'e'], ['b'], ['b', 'd'], ['b', 'e'], ['c'], ['d'], ['e']]
Admissible extensions: [] [['a'], ['b'], ['b', 'd']]
Preferred extensions: [['a'], ['b', 'd']]
Stable extensions: None

Test5.apx output:
['a', 'b', 'c', 'd'] [['a,b'], ['b,a'], ['b,c'], ['a,c'], ['c,d']]
There are 4 arguments and 5 attacks.
Conflict free extensions: [] [['a'], ['a', 'd'], ['b'], ['b', 'd'], ['c'], ['d']]
Admissible extensions: [] [['a'], ['a', 'd'], ['b'], ['b', 'd']]
Preferred extensions: [['a', 'd'], ['b', 'd']]
Stable extensions: None

f. Given an argumentation graph, a modification consists of removing an existing attack or adding a new attack between arguments. Design a solution that will take as input an

argumentation graph and a set of arguments X and output a new graph with the least number of modifications such that X is a preferred extension for the new graph. To answer this question, you will need to provide the corresponding pseudo-code/code-snippets as well as explanations in the report. **(3.5 marks)**

Answer:

g. Implement a Python program **dynamic_AF.py** containing a function **enforce_extension** that will take as input both the path to a file in the ASPARTIX format and a path to a file with a set of arguments X and output a new graph with the least number of modifications such that X is a preferred extension. In your report, show the output of your implementation on the inputs below. **(4 marks)**

File	*X*
test1.apx	{"*a*","*b*","*g*","*h*"}
test1.apx	{"*a*","*b*","*d*","*f*","*h*"}

Answer:

At this question I implemented the code that takes example.apx as the input file if command line argument is empty.

Appendix 1

```
#!/usr/bin/env python
import re

class Apx_parser:
    def __init__(self, filepath):
        self.file = open(filepath, 'r')

    def read_file(self):
        arguments = []
        attacks = []
        #print("parser")
        lines = self.file.readlines()
        for line in lines:
            if line.startswith('arg'):
                arguments.append(re.search(r"\(([A-Za-z0-9]+)\)",
line).group(1))
            elif line.startswith('att'):
                attacks.append(re.findall(r"\(([A-Za-z0-9,]+)\)", line))
            else:
                pass
        return arguments, attacks

    def close(self):
        if self.file:
            self.file.close()
            self.file = None
```

Appendix 2

```
import sys
from aspartix_parser import Apx_parser
import itertools
```

```
def conflict_free(arguments, attacks):
    confl_free_sets = []
    combs = []

    for i in range(1, len(arguments) + 1):
        els = [list(x) for x in itertools.combinations(arguments, i)]
        combs.extend(els)
    combs_sorted = [list(combs_sorted) for combs_sorted in combs][::-1]
    # print("Combs: ", combs_sorted)
    for i in combs_sorted:
        att_count = 0
        # print(i)
        for att in attacks:
            # print(att)
            if set([str(att)[2], str(att)[4]]).issubset(set(i)) or any([i
in item for item in

confl_free_sets]):  # any(x in item for item in confl_free_sets for x in
i):#(True if list(filter(lambda x:i in x,confl_free_sets)) else
False):#(any([set(i).issubset(set(item)) in item for item in
confl_free_sets])):
                break
            else:
                att_count += 1
            #    print(att_count)
            # if ((str(att)[2] and str(att)[4]) not in i) and (not any([i
in item for item in confl_free_sets])):
            #     att_count += 1
            #     print(att_count)
            if att_count == len(attacks):
                confl_free_sets.append(i)
    return confl_free_sets


def admissible(confl_free, attacks):
    admissible_sets = []

    for ext in confl_free:
        count = 0
        # print(ext)
        for att in attacks:
            if str(att)[4] not in ext:
                count += 1
                # print(att, count)
            else:
                # print(att, count)
                for atr in attacks:
                    # print(att, atr, count, str(atr)[4], str(att)[2])
                    if (str(att)[2] == str(atr)[4]) and (str(atr)[2] in
ext):
                        count += 1
                        # print(count)
        if count == len(attacks):
            admissible_sets.append(ext)

    return admissible_sets


# def complete(admissible_sets, attacks):
#     complete_ext = []
```

```
#     for adm in admissible_sets:
#         for att in attacks:
#             if (str(att)[2] in adm) and
#
#     return complete_ext

def preferred(admissible_sets):
    preferred_exts = []

    for adm in admissible_sets:
        count = 0
        # print(adm, count)
        for adm_t in admissible_sets:
            if set(adm).issubset(set(adm_t)) and adm_t != adm:
                pass
            else:
                count += 1
                # print(adm, adm_t, count)
            if count == len(admissible_sets):
                preferred_exts.append(adm)

    return preferred_exts


def stable_extensions(stable_exts):
    pass


if __name__ == '__main__':
    filepath = 'example.apx'
    if sys.argv[1:]:
        filepath = sys.argv[1]

    arguments = []
    attacks = []
    parser = Apx_parser(filepath)
    arguments, attacks = parser.read_file()
    parser.close()
    print(arguments, attacks)
    print("There are ", len(arguments), " arguments and ", len(attacks), "
attacks.")
    # print(str(attacks[0])[4])
    confl_free = conflict_free(arguments, attacks)
    print("Conflict free extensions: ", "[]", sorted(confl_free))
    admissible_sets = admissible(confl_free, attacks)
    print("Admissible extensions: ", "[]", sorted(admissible_sets))
    # complete_ext = complete(admissible_sets, attacks)
    preferred_exts = preferred(admissible_sets)
    print("Preferred extensions: ", sorted(preferred_exts))
    stable_exts = stable_extensions(preferred_exts)
    print("Stable extensions: ", stable_exts)
```