AUCSC 460 Lab 6 (3pages)

First Name: Nathan Last Name: Pelletier ID: 1480408

1. **Goal of Lab 5**

   1.1 To understand logic and forward chaining algorithm.

2. **Getting Credit for the Lab** This lab handout has questions.

   2.1 As part of the lab, you are to write the answers to these questions.

   2.2 Before the class ends, instructor will check your (unfinished) answers to record effort. Labs are graded on effort and correctness. The instructor may ask you a few questions to make sure you understand the material and then record your success.

   2.3 You then submit **lab6.pdf** after answering the questions and **lab6.py** onto the eclass lab assignment page.

   In case that you do not finish it during lab, you can submit your lab assignment on the eclass by due date (Students who missed lab cannot upload lab document.): by 11:59 pm Tuesday next week. The full marks are 100 pts.
   If your submission date is Wednesday, you will have 10 pts deduction.
   If your submission date is Thursday, you will have 20 pts deduction.
   If your submission date is after Thursday, your submission will not be marked. You will get zero mark.

You can discuss all the following problems with classmates, but do not look at others' solutions or copy them.

Problem 1. Prove, or find a counterexample to, each of the following assertions (a) to (e):

a. α is valid if and only if $True \models \alpha$.
b. If $\alpha \models (\beta \wedge \gamma)$ then $\alpha \models \beta$ and $\alpha \models \gamma$.
c. If $\alpha \models (\beta \vee \gamma)$ then $\alpha \models \beta$ or $\alpha \models \gamma$ (or both).
d. $\alpha \equiv \beta$ if and only if the sentence $(\alpha \Leftrightarrow \beta)$ is valid.
e. $\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

a. Assertations are of the form A and B and C |= D or E where the left-hand side must be true for the one part of the right-hand side to be true therefore True |= a means that a is always true.
   If a is always true we call it valid.
   With these definitions we can conclude that:
      if a is valid then we can assume True |= a and
      if True |= a then we can assume a is valid.
b. We know that (B and C) must be true as a |= D means that D is true if a is true. The only way that (B and C) can be true is if B and C are true so we can conclude that a |= B and a|= C hold true as both variables are tied to a.
c. We know that a |= B, C is of the form if a is true then B or C are true. Putting (B or C) is unnecessary here but regardless, we know that (B or C) must be true if a is true. By the

definition of or B may be true, C may be true, or B and C may be true. Therefore, we can conclude that

a |= (B or C) implies either that a |= B or a |= C or (a |= B and a |= C)

d. Triple equals means that two objects are equivalent.
This means that a is B.
If a is B then we can re-write the problem to "is the sentence (a ←→ a) valid"
We know that a → a is true and that a ← a is true as True → True = True and False → False = False, therefore we can conclude that (a ←→ a) is always true and therefore valid. If (a ←→ a) is valid then (a ←→ B) is valid.

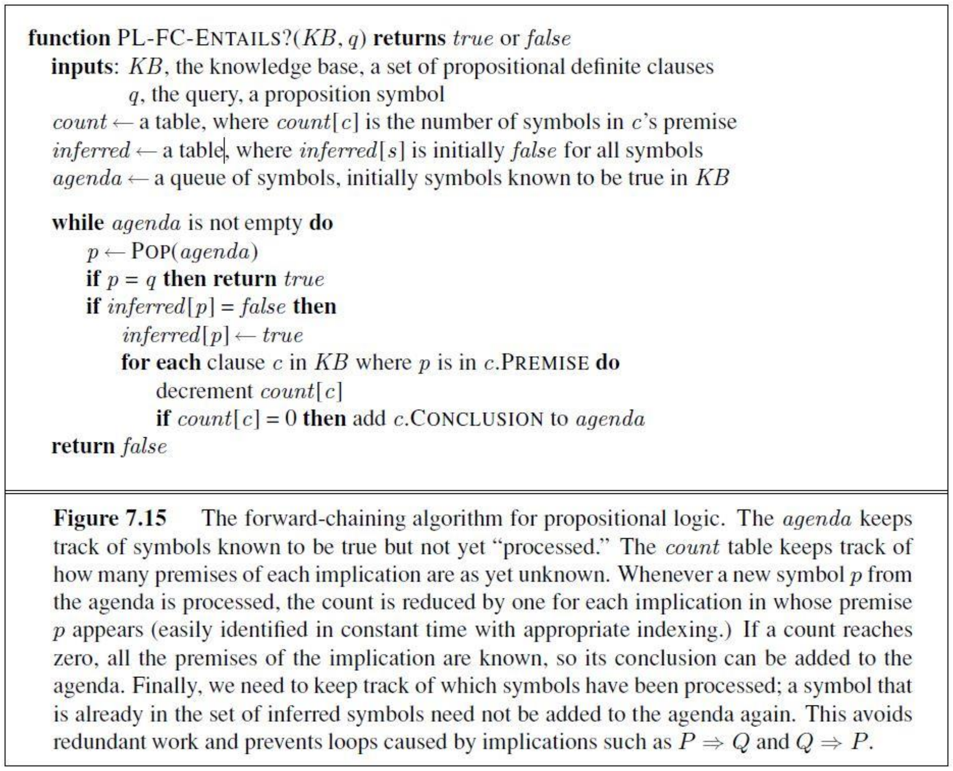
Going the other way (a←→ B) means that a and B is true if a and B are True or if a and B are false. For (a←→ B) to be valid a and B must give the same outputs. But that's what a≡B means, that a is B and B is a

e. a |= B means that a asserts B so if a is true then B is true.
(a and not B) is only true if a is true and b is False but this breaks the above definition therefore (a and not B) can never be true.

Going the other way (a and not B) requires a = True and B = False.
True |= False can not exist as a asserts the value of B therefore (a and not B) is always false with the inclusion of the rule a |= B.

Problem 2.

You will implement a Forward Chaining algorithm of Figure 7.15 in the textbook. You can do minor changes of the algorithm in textbook, but no major changes.
Below is the result by running the python code. (Black is what your code says, and red is what user typed.)



```
function PL-FC-ENTAILS?($KB$, $q$) returns $true$ or $false$
  inputs: $KB$, the knowledge base, a set of propositional definite clauses
          $q$, the query, a proposition symbol
  $count$ ← a table, where $count[c]$ is the number of symbols in $c$'s premise
  $inferred$ ← a table, where $inferred[s]$ is initially $false$ for all symbols
  $agenda$ ← a queue of symbols, initially symbols known to be true in $KB$

  while $agenda$ is not empty do
    $p$ ← POP($agenda$)
    if $p = q$ then return $true$
    if $inferred[p] = false$ then
      $inferred[p]$ ← $true$
      for each clause $c$ in $KB$ where $p$ is in $c$.PREMISE do
        decrement $count[c]$
        if $count[c] = 0$ then add $c$.CONCLUSION to $agenda$
  return $false$
```

Figure 7.15 The forward-chaining algorithm for propositional logic. The *agenda* keeps track of symbols known to be true but not yet "processed." The *count* table keeps track of how many premises of each implication are as yet unknown. Whenever a new symbol p from the agenda is processed, the count is reduced by one for each implication in whose premise p appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

```
Please input knowledge base:(type 'exit' if you want to stop)
P=>Q
KB[0].premise[0]:P, KB[0].conclusion:Q, count:1
Please input knowledge base:(type 'exit' if you want to stop)
L^M=>Q
KB[1].premise[0]:L, KB[1].premise[1]:M, KB[1].conclusion:Q,
count:2
Please input knowledge base:(type 'exit' if you want to stop)
B^L=>M
KB[2].premise[0]:B, KB[2].premise[1]:L, KB[2].conclusion:M,
count:2
Please input knowledge base:(type 'exit' if you want to stop)
A^P=>L
KB[3].premise[0]:A, KB[3].premise[1]:P, KB[3].conclusion:L,
count:2
Please input knowledge base:(type 'exit' if you want to stop)
A^B=>L
KB[4].premise[0]:A, KB[4].premise[1]:B, KB[4].conclusion:L,
count:2
Please input knowledge base:(type 'exit' if you want to stop)
A
Agenda[0]:A
Please input knowledge base:(type 'exit' if you want to stop)
B
Agenda[1]:B
Please input knowledge base:(type 'exit' if you want to stop) exit
stopped
```

You will need a class KB that has premise, conclusion and count. The following is an example that you can refer to:

```
class KB:          def
__init__(self):
self.premise=[""]
self.conclusion=""
self.count = 0

inferred  = { } agenda
= [""]
```

```
What is your query?
Q
=============
Forward chaining algorithm starts
=============
***** Current agenda:A *****
A^P=>L, count:2
Premise A matched agenda
Count is reduced to 1
A^B=>L, count:2
Premise A matched agenda
Count is reduced to 1
=============
***** Current agenda:B *****
B^L=>M, count:2
Premise B matched agenda
Count is reduced to 1
A^B=>L, count:1
Premise B matched agenda
Count is reduced to 0 ==> Agenda L is created
=============
***** Current agenda:L *****
L^M=>P, count:2
Premise L matched agenda
Count is reduced to 1
B^L=>M, count:1
Premise L matched agenda
Count is reduced to 0 ==> Agenda M is created
=============
***** Current agenda:M *****
P^M=>Q, count:1
Premise M matched agenda
Count is reduced to 0 ==> Agenda Q is created
L^M=>P, count:1
Premise M matched agenda
Count is reduced to 0 ==> Agenda P is created
=====
***** Current agenda:Q *****
Goal Achieved
The query Q is true based on the knowledge.
---- The End -----
```

My version:

```
Please input knowledge base: (type 'exit' if you want to stop)
: P=>Q
KB[0].premise[0]:P, KB[0].conclusion:Q, count:1
Please input knowledge base: (type 'exit' if you want to stop)
: L^M=>Q
KB[1].premise[0]:L, KB[1].premise[1]:M, KB[1].conclusion:Q, count:2
Please input knowledge base: (type 'exit' if you want to stop)
: B^L=>M
KB[2].premise[0]:B, KB[2].premise[1]:L, KB[2].conclusion:M, count:2
Please input knowledge base: (type 'exit' if you want to stop)
: A^P=>L
KB[3].premise[0]:A, KB[3].premise[1]:P, KB[3].conclusion:L, count:2
Please input knowledge base: (type 'exit' if you want to stop)
: A^B=>L
KB[4].premise[0]:A, KB[4].premise[1]:B, KB[4].conclusion:L, count:2
Please input knowledge base: (type 'exit' if you want to stop)
: A
Agenda[0]:A
Please input knowledge base: (type 'exit' if you want to stop)
: B
Agenda[1]:B
Please input knowledge base: (type 'exit' if you want to stop)
: exit
stopped

What is your query?
Q
=============
Forward chaining algorithm starts
=============
***** Current agenda:A ******
P=>Q, count: 1
L^M=>Q, count: 2
B^L=>M, count: 2
A^P=>L, count: 2
Premise A matched agenda
Count is reduced to 1
A^B=>L, count: 2
Premise A matched agenda
Count is reduced to 1
=============
***** Current agenda:B ******
P=>Q, count: 1
L^M=>Q, count: 2
B^L=>M, count: 2
Premise B matched agenda
Count is reduced to 1
A^P=>L, count: 1
A^B=>L, count: 1
Premise B matched agenda
Count is reduced to 0
==> Agenda L is created
=============
***** Current agenda:L ******
```

```
P=>Q, count: 1
L^M=>Q, count: 2
Premise L matched agenda
Count is reduced to 1
B^L=>M, count: 1
Premise L matched agenda
Count is reduced to 0
==> Agenda M is created
=============
***** Current agenda:M ******
P=>Q, count: 1
L^M=>Q, count: 1
Premise M matched agenda
Count is reduced to 0
==> Agenda Q is created
=============
***** Current agenda:Q ******
Goal Achieved
The query Q is true based on the knowledge.
---- The End -----
```