CSE560: Homework 6 Version 2

Version 1: Fixed example in Q24 for encoding the database from tcl to python.

Version 2: Q24: I am now giving you the KB encoded in python, and several hints. And I included the code that you should start with.

Horne Clauses

Let KB be the following.

\[
\begin{align*}
\text{false} & \leftarrow \text{b} \land \text{c} \\
\text{b} & \leftarrow \text{a} \\
\text{c} & \leftarrow \text{a}
\end{align*}
\]

**Question 1: Conjunctive Normal Form**

Write the above in conjunctive normal form.

**Question 2: Negation of Query**

In a refutation proof, to prove \(g\), you take the negation of \(g\) and add it to your knowledge base \(KB\) giving \(KB'\). If you can prove false from \(KB'\), than \(g\) follows from \(KB\).

Consider the query \(\neg a\).

What is its negation, written in conjunctive normal form?

**Question 3: Augmented KB**

Give \(KB'\), which is your original KB (in conjunctive normal form) along with the negation of your query (also in conjunctive normal form).

**Question 4: Form of Augmented KB**

Are all of the clauses in \(KB'\) horne? Explain why.

**Question 5: Unit Resolution**

Unit resolution can be viewed as a variation of the bottom up proof procedure where you only do resolutions where one of the resolvents is a unit clause. Also, you do not need to do any pruning with unit resolution.

If \(KB'\) is horne, unit resolution (ur) is guaranteed to find false if false follows from the \(KB'\).

In other words, \(KB' \vdash_{ur} \text{false}\) if \(KB' \models \text{false}\).

If \(KB'\) is not horne, unit resolution is not guaranteed to find false, but you can still use the bottom-up or top-down proof procedure.

Prove \(\neg a\) follows from \(KB\) using a refutation proof and unit resolution. Show all of your steps.
Disjunctive & Negative Knowledge

Consider the following KB:
a ∨ b ∨ c
¬a ∨ ¬b

Question 6: Interpretations and Models

Enumerate all interpretations, as a truth table, and indicate which ones are models.

Question 7: Resolvent

If you apply the resolution rule to two clauses from the knowledge base, getting g, should KB |= g? Justify your answer.

Use of Resolution Rule For each of the following, (a) indicate which interpretations it is true in, and (b) whether it is true in all models of the KB. Hence, conclude whether the resolution was sound or not.

\[
a ∨ b ∨ c
\]
\[
¬a ∨ ¬b
\]
Resolve a with ¬a

\[
b ∨ ¬b ∨ c
\]

\[
a ∨ b ∨ c
\]
\[
¬a ∨ ¬b
\]
Resolve b with ¬b

\[
a ∨ ¬a ∨ c
\]

\[
a ∨ b ∨ c
\]
\[
¬a ∨ ¬b
\]
Incorrectly resolve a with ¬a and b with ¬b

Question 8: Resolution Rule and Variables

What are all of the possible results from applying the resolution rule to the following.

\[
a(X,c) ∨ b(X,d)
\]
\[
¬b(e,Y)
\]

\[
a(X,c) ∨ b(X,d) ∨ f
\]
\[
¬a(e,Z) ∨ ¬b(e,Y)
\]

Question 9: Unit Resolution

Which of the above resolutions are examples of unit resolution?

Unicorns

*If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.*
Question 10: Logic

Write the above in logic. You don’t need to use any variables. For instance mythical can stand for that the unicorn is mythical. Also, note that immortal is just the same as not mortal.

Question 11: CNF

Convert clauses into conjunctive normal form.

Question 12: Type of KB

Is the set of clauses equivalent to horne? Is the set of clauses equivalent to datalog?

Mothers and Daughters

Consider the following clauses:

daughter(Daughter,Mother) ← ¬male(Daughter) ∧ mother(Mother,Daughter)
mother(mary,nancy)
¬male(nancy)

Question 13: CNF

Write each clause in conjunctive normal form.

Question 14: Type of KB

Is this datalog? Is this horne? Why or why not?

Question 15: Building the Consequent Set

Show, step-by-step, how the consequent set is built for a bottom-up proof. At each step, show what rules you applied, the new consequent set, and what you pruned out and why. From your consequent set, does it follow that daughter(nancy,mary).

Question 16: Top-Down Proof

Show the steps of a top-down proof that daughter(nancy,mary)

Disjunctive Answers

Consider the following KB:

poor(X) ← student(X)
student(john) ∨ student(tim)
Question 17: Top-Down Proof

What is the answer for \( \text{poor}(X) \).
Show the derivation using a top-down approach.

Complete Knowledge Assumption

Consider the following KB.

\[
\text{bird(tweety)} \\
\text{fly}(X) \leftarrow \text{bird}(X) \land \neg \text{abnormal}(X) \\
\text{abnormal}(X) \leftarrow \text{toy}(X) \\
\text{abnormal}(X) \leftarrow \text{dead}(X) \\
\text{toy(gun)} \\
\text{dead(elvis)}
\]

Question 18: Need for CKA

Without assuming the CKA (just using normal disjunctive and negative knowledge), can you prove that tweety can fly? If yes, give such a proof, if not, show, using either top-down or bottom-up why the proof fails.

Question 19: Negation as Failure

Can we prove that tweety can fly if we use negation as failure? If so, give the proof? Make sure you show all of the details of the proof.

Question 20: Clark Normal Form

Clark completion is sometimes employed on just some of the predicates. We will be doing that with the KB of the previous question for \text{abnormal}, \text{toy} and \text{dead}.

Write the Clark normal form of each of the 3 predicates.

Question 21: Clark Completion

Write the clark completion of each.

Question 22: CNF

For each rule that needs to be added for the Clark Completion (just the one sided implication that needs to be added), write it in conjunctive normal form.

Question 23: Augmented KB

Add the completion assumptions to the database and prove using the top-down proof procedure for disjunctive knowledge that tweety flies. Note that you can also assume the Unique Name assumption (that equality is the same as unification).
Programming

Question 24

In this question, you will implement a theorem prover for Horn clauses that uses refutation and unit resolution. The knowledge base will not use function symbols, thus guaranteeing that it always halts.

You will be using the following KB.

ostrich(sam)
canary(tweety)
bird(X) ← ostrich(X)
bird(X) ← canary(X)
fly(X) ← bird(X) ^ normal(X)
not normal(X) ← ostrich(X)
normal(X) ← canary(X)

You will need to encode the KB in CNF. We will be using Python lists to store the disjunction, and we will use not for negative literals. So the third line will be coded as:

['bird', 'X'], ['not', ['ostrich', 'X']]

Since previous questions in this assignment have tested your ability to convert clauses into CNF, here is the converted KB.

kb = [['ostrich', 'sam'],
      ['canary', 'tweety'],
      ['bird', 'X'], ['not', ['ostrich', 'X']],
      ['bird', 'X'], ['not', ['canary', 'X']],
      ['fly', 'X'], ['not', ['bird', 'X']], ['not', ['normal', 'X']],
      ['not', ['normal', 'X']], ['not', ['ostrich', 'X']],
      ['normal', 'X'], ['not', ['canary', 'X']]]

Make a procedure called prove(query, kb) that takes as input a query (a positive or negative literal) and a KB, and determines if query follows from the KB.

Here is how prove should work (if you want to do it a different way, please explain how it works in a paragraph). Note that for proving each one, you need to add its negation to the knowledge base and see if you can derive false (which will be an empty list).

Let your consequent set be the initial KB. Add the negation of the query to the knowledge base. Your KB is a list, which is mutable; so make sure you make a copy of the KB before altering it.

For each pair of clauses in the KB, if one of them is a unit clause and they can be resolved and the result is not already in the consequent set, then add it in. Keep doing this until you are not able to add anything to your consequent set or you derive an empty disjunction (which is how false is represented).

Hint 1: Make sure you apply fresh variables!

Hint 2: The control structure is a bit difficult. You will be iterating over the clauses in the KB. When you find a new resolvent, do not add it immediately into the KB. Instead, add it to newClauses, and when you have finished iterating over the KB, then add it in. Once you have added in the new clauses, you will need to iterate again, as you might be able to now derive additional clauses. In fact, you need to keep cycling until no new clauses are found.

Please use hw4standard. You should use fresh variables, unify and substitute from it. You should also use the following three procedures as well. SameClause determines if two clauses are the same (minus variable renaming). Use this to make sure you do not add a clause that you already have that just happens to use different variables. Negate takes a literal and negates it. You should use this to convert your initial query and as part of your resolution rule. PrettyCNF prints a clause that is a disjunction.
def PrettyCNF(expr):
    s = ""
    for literal in expr:
        if s != "": s += " v "
        if type(literal) is list:
            if literal[0] == "not":
                s += "!"
            atom = literal[1]
        else:
            atom = literal
        s += prettyexpr(atom)
    return s

def sameclause(a,b):
    subs = {}
    ret = unify(a,b,subs)
    if ret == False or subs == {}: return ret
    # same clause if bottom just has variables and each is different
    seen = []
    for bottom in ret.values:
        if not isVar(bottom):
            return False
        if bottom in seen:
            return False
        seen.append(bottom)
    return True

def negate(literal):
    if type(literal) is list and literal[0] == "not":
        return literal[1]
    return ["not",literal]

Make sure your code is easy to read. And add in output statements that show what unit clause and other clause it is considering, and whether it finds a new resolvent to add.

Test your code on the KB above, with the queries fly(sam) and fly(tweety). You should get false for the first and true for the second.

Also, try it on the following KB with just a single clause in it: boy(goo(X,Y)) v boy(foo(X,Y)) and make sure that you can prove boy(X).

Hand in your code.

**Prolog: Boolean Expressions**

Unification is a powerful construct in Prolog. It allows variable values to be partially and incrementally specified. To illustrate this, we are going to look at boolean expressions in conjunctive normal form. Say you are given an equation like \((a\lor b\lor \neg c)\land (a\lor c)\). This equation is satisfiable as it has a solution, for example where \(a\), \(b\) and \(c\) are each true. In fact, as long as \(a\) is true, it does not matter what \(b\) and \(c\) are assigned to.

As a tie into the class, \(a\), \(b\) and \(c\) can be viewed as 0-ary predicates; \((a\lor b\lor \neg c)\land (a\lor c)\) can be viewed as a knowledge-base with 2 disjunctions, and we are asking if there is model. For the rest of this section, we will refer to \(a\), \(b\) and \(c\) as boolean variables, as they can be true or false.

You will write a prolog program that will determine if an equation has a solution. The actual code to this problem is very small, as it will strongly make use of Prolog’s variables and backtracking. You only need to use pure prolog. You do not need to use ‘not’, or ‘findall’, nor any command that we did not discuss in
class. The program is very small, but you might find it difficult to conceptualize. Hence, I have broken it into a number of steps.

**Question 25: Disjunctions**

Consider a boolean expression written in conjunctive normal form: conjunction of disjunctions, where each of the disjuncts is either a positive or negative literal.

Let’s just focus on a disjunction. For a disjunction, you just need to make sure that at least one of the disjuncts is true. Give all of the minimal different ways to make the following disjunction true. For each boolean variable, specify if it should be true, false, or does not matter (you should have 4).

\((a \lor b \lor \neg c \lor d)\)

**Question 26: Prolog and Disjunctions**

To make best use of prolog, we will use a Prolog variable for each boolean variable. This way, we can use Prolog’s built-in unification to update their values across the entire expression if we change it in one disjunction.

Also, we will use \(p(X)\) for a positive literal and \(n(X)\) for a negative literal. This way, we can use unification to see if an element in a disjunctive is a positive or negative literal.

And we will use prolog lists to group the disjuncts in a disjunction and to group all of the conjuncts. So the above expression can be written as: 
\[[[p(A),p(B),n(C)],[p(A),n(B)],[n(A)]]\]

Now, build a predicate called disjunction that takes a disjunction and makes one of the disjuncts true (thus making the entire disjunction true). On backtracking, it should go through all of the choices, just like you did in the previous question. Here is what it should return, where you press ‘;’ after each result.

?– disjunction([p(A),p(B),n(C)]).
A = true ;
B = true ;
C = false ;
false.

If some boolean variables have values already, your routine should be able to take this into account. The second response is because Prolog did not have to make a variable assignment to make the second disjunct true.

?– disjunction([p(A),p(true),n(C)]).
A = true ;
true ;
C = false ;
false.

Lastly, consider the following in which the second variable was assigned false. The second disjunct cannot be made true, so Prolog skips over it.

?– disjunction([p(A),p(false),n(C)]).
A = true ;
C = false ;
false.

Hand in your Prolog code for disjunction.
**Question 27: Conjunction of Disjunctions**

Write a predicate called `conjunction` that will make one disjunct in each disjunction true, and so, make the entire conjunction true.

As the boolean variables are Prolog variables, setting a variable to make one disjunct true will affect its value every else it is used.

If it is not possible to make a disjunction true (due to which disjunct was chosen to make prior disjunctions true), your code should allow Prolog to backtrack to consider making other disjuncts true instead.

Our use of Prolog’s variables and backtracking will make this code extremely small!

Consider this example:

\[(a \lor b \lor \neg c) \land (a \lor \neg b) \land (\neg a)\]

If we first try making \(a\) true to make the first disjunct true, and which makes the second disjunct true, we will have to abandon this, as that will make the 3rd disjunct false. We can then try to make \(b\) true to satisfy the first disjunct, and make \(a\) true to make the second disjunct false, which also leaves it at a deadend with the third disjunct. We can then try to make \(c\) false to satisfy the first disjunction, which will lead to a solution. All of the backtracking to determine how to make each disjunction true can be automatically done with Prolog.

Hand in your code.

**Question 28: Program Runs**

Show the results from executing it on the following example, showing all of the different values it returns:

```
conjunction([[p(A),p(B),n(C)],[p(A),n(B)],[n(A)]])
```

Also, run it on the following example, also showing all of the different values it returns.

```
conjunction([[p(A),p(B),n(C),p(D)],[p(A),n(B)],[n(A)],[n(D),n(A)]])
```