Reasoning about Equality

Consider the following KB. Although it is using the ‘=’, this is for true equality, not explicit unification.

\[
eveningstar = \text{venus} \quad \text{morningstar} = \text{venus} \quad \text{planet(venus)}
\]

**Question 1: Axioms**

List all of the equality axioms that should be added to the KB in order to reason about equality.

**Question 2: Proof with Axioms**

Using the axioms for equality, prove:

\[
?\text{planet(morningstar)}
\]

Do this as a top-down proof. For each derivation, show what rule from the KB you used and the substitution.

Warning: Do not use paramodulation. Review the lecture slides for an example of doing a proof with the axioms.

**Question 3: Paramodulation**

For paramodulation, what are the rewrite rules that the proof procedure can use, in addition to the resolution rule.

**Question 4: Proof with Paramodulation**

Prove \(\text{planet(morningstar)}\) using paramodulation.

**Prolog**

**Question 5: Cliques**

Consider the following prolog knowledge base.

\[
\begin{align*}
\text{node}(1). & \quad \text{node}(2). & \quad \text{node}(3). & \quad \text{node}(4). & \quad \text{node}(5). & \quad \text{node}(6). \\
\text{node}(7). & \quad \text{node}(8). & \quad \text{node}(9). & \quad \text{node}(10). & \quad \text{node}(11). & \quad \text{node}(12). \\
\text{node}(13). & \quad \text{node}(14). & \quad \text{node}(15). \\
\text{c}(1,2). & \quad \text{c}(1,3). & \quad \text{c}(1,5). & \quad \text{c}(1,7). & \quad \text{c}(1,8). & \quad \text{c}(1,11). \\
\text{c}(2,4). & \quad \text{c}(2,8). & \quad \text{c}(2,12). & \quad \text{c}(3,5). & \quad \text{c}(3,7). & \quad \text{c}(3,11). \\
\text{c}(4,6). & \quad \text{c}(5,7). & \quad \text{c}(5,11). & \quad \text{c}(6,8). & \quad \text{c}(7,11). & \quad \text{c}(8,11). \\
\text{c}(9,12). & \quad \text{c}(10,12). & \quad \text{c}(13,14). & \quad \text{c}(13,15). & \quad \text{c}(14,15).
\end{align*}
\]

This knowledge base is a graph, with each node presented with the predicate ‘node’, and each bi-directional arc represented with the predicate ‘c’.
Make a datalog program for finding a $k$-clique in the graph. A $k$-clique is a set of $k$ distinct nodes that are all connected to each other. You should make a recursive definition. You must define all predicates that you use, and cannot use any prolog capabilities that are outside of datalog.

We will be assuming the unique name assume, so you can use not(X=Y), if you are sure that the X and Y will be instantiated when that atom is evaluated by Prolog.

Use Prolog's lists [Top|Rest] and Prolog's numbers, and use the built-in ‘is’ and ‘>’. Again, you need to make sure that the appropriate parameters are bound.

You should make sure that your code gives you appropriate alternatives when you press the ‘;’ after it gives an answer. Your code should not go into an endless loop, or return incorrect solutions.

To avoid your code supplying many different variations of the same solution, impose an additional constraint that cliques need to be grown in ascending order of the nodes. In other words, when adding a new node to an existing clique, make sure the new node is greater in value than the nodes in the existing clique (this is why I used numbers to represent the nodes).

Hand in a copy of your Prolog code.

**Question 6: Prolog Part II**

Now, we will focus on efficiency. We will count the number of times that Prolog gets the name of a block. We will use the predicate cnt to hold the current number of times that node has been called. Use countclique to call your clique predicate, which will reset cnt to 0. Use the following getnode predicate instead of directly calling node.

```prolog
countclique(N,Result) :-
    retractall(cnt(_)),
    assert(cnt(0)),
    clique(N,Result).

countclique(_,_) :-
    cnt(N),
    print(N).

gidenavode(Node) :-
    node(Node),
    retract(cnt(Cnt)),
    Cnt1 is Cnt + 1,
    assert(cnt(Cnt1)).
```

Look up on the web what retract and assert do. Explain how they are affected when Prolog backtracks. Explain how they can keep track of the actual number of times that node is called, regardless of whether backtracks.

Also explain what the second case of countclique is doing.

**Question 7: Prolog Part III**

Modify your code so that you use the above two predicates (countclique and getnode). My code is able to fully search for all 1 cliques with executing getnode 15 times, 2-cliques with 240 times, 3-cliques with 585 times (and found 13 different ones), 4-cliques with 780 times (and found 5 different ones), and 5-cliques with 855 times (and found 1 one).

Your code should give a similar amount. If your code is reporting greater numbers, try to determine what it might be doing that is inefficient to improve your code.
Hand in your code, and report how many times getnode was called, and how many solutions you found, for finding 1, 2, 3, 4, and 5-cliques.