1. Question 1: Short answer questions

1.1 **Soundness of a proof system:** In mathematical logic, a proof system is said to be sound if and only if any formula that can be proven in the system is logically valid with respect to the system's semantics. In most cases, this is due to its rules that have the property of preserving truth.

From the above definition in short, we can define that a proof system P is sound for logic L, if and only if whenever P has a proof of an Logic formula F, then F is valid.

The converse of soundness is known as completeness.

Completeness of a proof system: In mathematical logic, a proof system is called complete with respect to a particular property,if every formula having the property can be derived using that system, i.e. is one of its theorems; otherwise the system is said to be incomplete.
Intuitively, a system is called complete in this particular sense, if it can derive every formula that is true.

From the above definition in short we can define proof system P is complete for logic L, if for any valid L-formula F, the system P has a proof of F.

The proof system mentioned in the question is Unsound but complete. The reason is that for all formulas, including invalid ones, P produces proofs and thus violates the soundness definition for the proof systems. Nonetheless, P is complete, since P generates a proof for logic formula if it is a valid formula.

1.2 **Semantic argument method:** As discussed in class, semantic argument method have 5 basic proof rules that can be used to prove the validity of boolean formulas.

1. Proof rule 1: **I |= ¬F === I ⊭ F**
2. Proof rule 2: **I |= F ⋀ G === I |= F and I |= G**
3. Proof rule 3: **I |= F ⋁ G === I |= F | or I |= G**
4. Proof rule 4: **I |= F → G === I ⊭ F or I |= G**
5. Proof rule 5: **I |= F ↔ G === I |= F ⋀ G or I |= ¬F ⋀ ¬G**

From the above proof rules we can observe that some of the proof likes 1 and 2 deal with negations and can lead to use of negations recursively in deduction procedure.

Bob has accidentally commented on the part of the algorithm that deals with this recursive case for negation, let;s see how it will affect the soundness and completeness of the system.

Soundness of the Bob's method: We have already defined that a sound system whenever produces a proof , the input is guaranteed to be valid. This Bob's method might miss out producing proof for the formulas which use rules for negations. But Bob's method is sound as whenever it produces a proof, the input is valid,because it uses Alice's sound proof system except on missing the proofs for formulas that use rules for negation in deducing the validity of the formula.

Completeness of Bob's method: We have defined in 1.1 that for a system to be complete for logic L, if for any valid L-formula F, the system P has a proof of F. The problem with Bob's method is that it will not be able to deduce the valid formulas that use rules involving negation for the reasoning with proof rules. Thus this method will not be able to provide proofs for such formulas, which does not support definition of a complete system.So Bob's method/system is incomplete.

Bob's system is sound but incomplete.

1.3 Are context-free grammars closed under infinite union?

A context-free grammar is a set of recursive rules which are used to create string patterns. All standard languages and more can be represented in context-free grammar, but they can not represent all possible languages.A context-free grammar (CFG) consisting of a finite set of grammar rules.

Let
$L_1 = \{ab\}$
$L_2 = \{aabb\}$
$L_3 = \{aaabbb\}$
…….
$L_n = \{a^n b^n\}$
Be the context free grammars, the the infinite union of these context free grammars become:

$L = L_1 \cup L_2 \cup L_3 \cup L_4$……..

$L = \{a^n b^n,$ where n>=1) which indicates that it is not a finite set of context free grammar.

So we can conclude that infinite union of context free grammars may not be closed.

2. Question 2: Semantics of Boolean Formulas

Let's assume that the DAG is represented using the node class as shown below (referred from A1 solutions):

```
class node {
int type ; / / AND, OR , NEG, IMPLIES , IFF , TRUE , FALSE , VAR
int value ; / / Variable name
node left ;
node right ;
} ;
```

Pseudo code to Evaluate a Boolean formula for a given assignment is as below.

Terminology: Let Boolean formula in DAG = F and given assignments is a dictionary of values for all variables. Assignments dictionary = A.

As mentioned in above code snippet the possible values for the type variable of a node class a AND, OR , NEG, IMPLIES , IFF , TRUE , FALSE , VAR these are nothing but the literals of the boolean formula. The value holds the variable name when type = VAR.

What below recursive function is doing:

- If the assignment dict is empty, it returns an error. It also returns an error if the assignment for a variable which is in formula is not available in error.
- If the formula itself is just TURE , then the function returns True, this is one of the recursive base cases.
- If the formula itself is just FALSE, then the function returns False, this is one of the recursive base cases.
- If the DAG/sub DAG node object type is any of the binary operator like AND,OR,IMPLIES or IFF, then a recursive function to calculate both left and right sub dag's is called using F.left and F.right as Eval function argument along with assignments dictionary.
- If the graph traversal has reached the leaf node, then this function returns the assignment of the variable at that leaf node. If the assignment of that variable is not available in the given dictionary then a dictionary key error is thrown.
- If the graph has reached leaf nodes at a binary operator as the root node then based on the operator the resultant value of the sub dag is returned to the recursively called function.
- If the root node type is NEG, then the function returns the negation of the recursive call of the left node of that root node. As negation has only one operand, in this case it is a left node.
- If the root node is none of the predefined types, then the function returns an error.
- After recursively evaluating the the nodes, then the function reaches one of the two bases available and returns either TRUE or FALSE as output, if the formula is well defined as per the requirements.

```
Eval(formula F, dict A){ // starting evaluate recursive function
    if(dict A == empty){
        // return an error if the assignments dictionary is empty
        return empty_dict_error;}
    if(F == Null) {
        // returning nothing if the formula is empty
        return ; }
    else if (F == TRUE) {
        // returning True if the formula itself is True
        return TRUE;
    }
    else if (F == FALSE) {
        // returning False if the formula itself is False
        return FALSE;
    }
    // condition to decide what to do when the operator is binary
    else if (F.type == AND or F.type == OR or F.type == IMPLIES or F.type == IFF){
        // repulsively calculating the left  sub tree, if leaf node variable value is
returned
        val  L = Eval(F.left,A)
        // repulsively calculating the right subtree, if leaf node variable value is
returned
        val  R = Eval(F.right,A)
        // declaring result variable
        result = ""
        //calculating the resultant operation based on operator
        If (F.type == OR)
           { result = L || R }
        else if (F.type == AND)
           { result = L && R }
        else if (F.type == IMPLIES)
           { result = !L || R }
        else if (F.type == IFF)
           { result = L == R}
        return result;
    }
    else if (F.type == VAR){
          // returning the value assigned for the variable by finding the assigned variable
from the assignment dictionary A
           if (A(F.value) != Null){ // checking if the variable value is in dictionary
                return  A(F.value); }
           else {
                // return a dictionary key error if the variable assignment is not available
in the given assignments dictionary
                Return dict_key_error: }
    }
    else if (F.type == NEG){
        // returning the negation of the formula to be recursively evaluated
        // as negation is unary operator only left branch will be available
        return ! EVAL(F.left,A);
    }
    else {
       return "error";
    }
} // endo of function EVAL
```

3. Question 3: Negation Normal Form

A normal form of a formula F is another formula F' such that F is equivalent to F' , but F' obeys certain syntactic restrictions. For a formula to be in **Negation Normal Form (NNF)** requires two syntactic restrictions. The only logical connectives in the formula should be ¬, ∧, ∨ (i.e., no →, ↔) and the second is that Negations appear only in literals and can only appear in front of variables. So to convert general Boolean formulas into NNF we have follow below steps:

- **First step:** recursively translate the implies(→) and iff(↔) in DAG to simplified ¬, ∧, ∨ using equivalence preserving laws.
  - **For IMPLIES :-** p → q is translated to equivalent ¬p V q . So the type of the node is to be changed to V from → and the left sub DAG has to be negated.
  - **For IFF :-** p ↔q is translated to (¬p V q) ^ (p V ¬q). So the type of root node is changed to AND, and new nodes are introduced at left and right to store (¬p V q) and (p V ¬q)
- **Second step:** push negations recursively such that appear only in literals or in front of variables.

At first we can write a function to recursively translate implies and iff to ¬, ∧, ∨ and returns the modified DAG. Then we call another function to recursively push the negations down to literals. Let's first look at the function for translation.

As previously used we use the same DAG structure , by using node class defined in Q2.

```
translate(node F) {
    if(F == Null){
        return Null;
    }
    // if the root node of the sub DAG is AND or OR we can directly call translate recursively
to simplify the sub DAG's
    else if(F.type == AND or F.type == OR){
        translate(F.left)
        translate(F.right)
    }
    // if the type of root node is NEG, we can directly call translate on F.left as no right
sub DAG exists fro NEG nodes
    else if(F.type == NEG){
        translate(F.left)
    }
    // when the root node type is IMPLES we have to translate it using Equivalence preserving
laws.
    else if(F.type == IMPLIES){
        F.type = OR //modifying root node type as mentioned above
        F.left = ! F.left // negating left sub DAG
        translate(F.left)
        translate(F.right)
    }
    else if(F.type == IFF){
        F.type = AND
        // changing the type of left node to OR and its nodes respectively storing  (¬p V q)
        F.left.type = OR
        F.left.left = ! F.left
        F.left.right = F.right
```

```
        // changing the type of right node to OR and its nodes respectively for storing  (p V
¬q)
        F.right.type = OR
        F.right.left = F.left
        F.right.right = ! F.right

        // recursively translate right and left nodes
        translate(F.left)
        translate(F.right)
    }
    // writing recursive Base cases
    else if(F.type == TRUE or F.type == FALSE or F.type == VAR)
        return node;
    }
    else {
        return error;
    }
}// end of function
```

What above function is doing:

- For the translation recursive function aboe, the base cases are TURE, FALSE, VAR, then the functions return the node value.
- If the type of node is AND,OR,NEG no transformations are needed to be performed, so just left and right sub DAG(if present) are recursively called for simplification.
- If the type of Node is Either IMPLIES or IFF, then the type of the node is changed and the left and right nodes are modified as mentioned in the beginning.

The above function returns the modified DAG after translating all IMPLIES and IFF's with equivalent boolean formulas and modifying the DAG. Now we can push the negations according to NNF definitions to fall into literals and before the variables. The below function takes input from the modified DAG from the translate function and pushes the negations down to variables.

```
push_negation(node F){
    if(F == Null){
        return Null;
    }
    else if(F.type == NEG){
        //checking if the left node is again a negation, which indicates double negation
which has to be removed.
        if(F.left.type == NEG){
             return push_negation(F.left.left);
        }
    else if(F.type ==OR){
        // then we have to modify this node to push negation to literals, OR will become AND
and the literals are negated
        F.type = AND
        F.left = ! F.left
        F.right = ! F.right
        // recursively neg pushing left and right sub dags
```

```
        push_negation(F.left)
        push_negation(F.right)
    }
    else if(F.type == AND){
         // then we have to modify this node to push negation to literals, AND will become OR
and the literals are negated
        F.type = OR
        F.left = ! F.left
        F.right = ! F.right
        // recursively neg pushing left and right sub dags
        push_negation(F.left)
        push_negation(F.right)
    }
}// end of function
```

What above function is doing:

- If the type of node is NEG , then the function checks if the left node is also negation to avoid double negations, which is not allowed in NNF form. If there are double negations then it is sent for push_negations recursively.
- If the type of node is OR then the function modifies the type to AND and negates both left and right nodes and then calls push_negations on those nodes.
- If the type of node is AND then the function modifies the type to OR and negates both left and right nodes and then calls push_negations on those nodes.
- ANd outputs the final NNF translated Boolean formula G, which is obtained from F.

Explain why your algorithm is equivalence-preserving:

- In the transformations of Implies and IFF , we used equivalent boolean formulas with NEG,AND,OR to replace the.
- In the push negations part we used De Morgan's laws for pushing negation down to variables.
- In both the steps, we have used equivalent perseverance formulas for replacing the original ones.
- So the approach above mentioned is an equivalence preserving method.

4. Question 4: DPLL SAT Solvers

Assumptions and terminology:

- Formulas in CNF are given as follows:
  - Every clause is a series of space-separate numbers. Each positive number represents a positive literal and each negative number represents a negative literal.
  - A formula is a set of such clauses, one clause per line.
- So the parsed format of the input formula is 2D array with each row as a clause and each element in the array is a variable.

  Function to carry out BCP in DPLL algorithm. The inputs of the function are CNF_formula and the unit_variable to be considered for BCP. BCP gets rid of all the negated variables from all the clauses of the formula. If any clause becomes empty after removing it, then -1 is returned to indicate contradiction and thus UNSAT.

```
// function to carry out BCP part of DPLL algorithm
def boolean_constraint_propagation(CNF_formula, unit_variable):
    modified_CNF_formula = []
    for clause in CNF_formula:
```

```
      // if the unit variable is in clause retaining it
        if unit_variable in clause: continue
       // if the negation of unit variable is in clause removing them
        if -unit_variable in clause:
            c = [x for x in clause if x != unit_variable]
            // after removing neg unit variable if we get a empty clause thats a
contradiction indicating UNSAT, so returning -1
            if len(c) == 0: return -1
            // else the modified clause is appended to new 2D array
            modified_CNF_formula.append(c)
        else:
            // if neg of unit variable is not available in clause, appending the un
modified clause to the return list
            modified_CNF_formula.append(clause)
    return modified_CNF_formula
```

The below count_variables function is used to count the number of times each literal is occurring in the formula. This count dictionary can be used to find out the pure literals that are available in the formula, which we will see in pure_literals function.

```
# function to count all different number of variables in the formula
# can be used to decide on the pure literals as in pure literals function
def count_literals(CNF_formula):
    counter = {}
    for clause in CNF_formula:
        for literal in clause:
            if literal in counter:
           // incrementing the count by 1 if the literal is available in dictionary
                counter[literal] += 1
            else:
            // if the literal is not available adding the literal to dictionary and
setting the count to 1
                counter[literal] = 1
    return counter
```

The below pure_literal function deals with the pure literals found in the input formula and gets possible assignments for them. The count dictionary generated using the previous function is used in this function. For every literal in the dictionary if the negation of the literal is not available in the dictionary then that is considered as a pure literal.

```
# function to find pure literals so that the pure literals can be assigned values
def pure_literals(CNF_formula):
    counter = count_literals(CNF_formula)
    assignment = []
    pures = []
    // checking if the negation of a literal is available in dictionary , if not
then its a pure literal ,
    for literal, times in counter.items():
        if -literal not in counter: pures.append(literal)
```

```
    for pure in pures:
        formula = boolean_constraint_propagation(CNF_formula, pure)
    assignment += pures
    // returning the modified formula and the assignments achieved due to pure
literals
    return formula, assignment
```

The below unit_propagation function is used to carry out the unit propagation part of DPLL. This function derives assignments for all the unit clauses in the formula by selecting one after and another.

```
// Function to carry out unit propagation part of DPLL algorithm
def unit_propagation(CNF_formula):
    assignments = []
    // loop to find unit clauses in the formula
    unit_clauses = [c for c in CNF_formula if len(c) == 1]
    // recursively selecting first unit clause to proceed further until all unit
clause are resolved
    while len(unit_clauses) > 0:
        unit = unit_clauses[0]
        // using BCP to get rid of negative literals of the unit_clause variables
        CNF_formula = boolean_constraint_propagation(formula, unit[0])
        assignment += [unit[0]]
        // if a contradiction appeared in BCP, then return -1 to get UNSAT
        if CNF_formula == -1:
            return -1, []
        // if all the clauses are solved in formula and an empty formula is
received the assignments are returned.
        if not CNF_formula:
            return CNF_formula, assignments
        unit_clauses = [c for c in formula if len(c) == 1]
    // returning the modified formula and the achieved assignments due to unit
clause
    return CNF_formula, assignments
```

The below function is used to select a literal randomly from the available literals to proceed further in the DPLL algorithm.

```
// selecting a random literal to proceed further when no further decisions are to
be made in DPLL
def random_variable_selection(CNF_formula):
    counter = get_counter(CNF_formula)
    return random.choice(counter.keys())
```

Function to achieve backtracking part of the DPLL algorithm. At first in this function teh formula is first reduced using pure literals and unit propagation steps. Once these steps are done then a random literal is selected and assigned true value to it, and the back_tracking function is called on the formula using modified assignments dictionary. If the recursive back_tracking fails to provide a solution, then the random literal is assigned the false value and recursively called using back_tracking function. If both the assignments fail to generate any solution assignment then the DPLL function outputs UNSAT.

```
// function to backtrack the selected variable, when a contradiction is achieved
for selected random literal
def back_tracking(CNF_formula, assignment):
    // as the first step identifying pure literals and getting assignments using
them
    CNF_formula, pure_assignments = pure_literal(CNF_formula)
    if formula == - 1:
        return []
    // as the second step  we performed unit propagation and getting assignments
using them
    CNF_formula, unit_assignments = unit_propagation(CNF_formula)
    if formula == - 1:
        return []

    // adding the obtained assignments to the original assignments list
    assignments = assignments + pure_assignments + unit_assignments
    // return the assignment when the formula is completely resolved
    if not CNF_formula:
        return assignments
    // randomly selecting a variable to proceed further when all resolvable
literals are resolved
    random_variable = random_variable_selection(CNF_formula)


   // recursively calling backtracking function on the selected variable
    solution = back_tracking(boolean_constraint_propagation(CNF_formula,
random_variable), assignments + [random_variable])
   // if the solution for the selected random variable is not obtained , then the
negation of the variable is selected and back_tracking is called to explore the
other side of the graph
    if not solution:
        solution = backtracking(boolean_constraint_propagation(formula, -variable),
assignment + [random_variable])
    return solution
```

Combining all above functions to produce the final DPLL function.

```
def DPLL_algorithm(CNF_formula):
    // calling back_tracking function with empty assignments, which return
assignments if SAT and an empty assignments if its UNSAT
    solution = back_tracking(clauses, [])
    // if the solution is not empty printing the assignments
    if solution:
        // changing the solution to avoid print dont care literals, whose negative
and positive literals are available in the solution.
        solution += [x for x in range(1, nvars + 1) if x not in solution and -x not
in solution]
        solution.sort(key=lambda x: abs(x))
        print('SATISFIABLE')
        print('v ' + ' '.join([str(x) for x in solution]) + ' 0')
    else:
        print('UNSATISFIABLE')
```

*Note : All the code is commented properly for each step to understand it properly. And the above code is a rough draft of how a DPLL algorithm works.

Why is DPLL SAT solver a decision procedure for the Boolean satisfiability problem?

In short, for an algorithm to be a decision procedure, it has to make a decision and terminate gracefully with the accepted values of output. In the case of DPLL SAT solver, we can assume that the solver is a decision procedure if it is able to find an assignment for SAT input and show the formula as UNSAT for UNSAT input formulas. We can observe how the DPLL algorithm works and derive that it will eventually produce results as expected above.

- **If input is a SAT formula:** DPLL at first performs unit resolution and then gets rid of pure literals for the input formula and the input formula is modified to a more concrete one at this stage. Then a literal from the available literals in the modified formula is selected, assigned a value and then again above two steps are performed, if a contradiction occurs then the assigned value for the literal is reversed.
- In each recursive step because of BCP and pure literals reduction, the literals in the modified formula get reduced one by one.
- The resusive steps reach a point where all the variables are assigned and value and there is no where to go and modify the variables, which is the end step of the algorithm.
- By the last recursive step end, DPLL in the worst case has tried all the possible combinations i.e. 2^n assignments for the available n variables. As we said the input is a SAT, DPLL will definitely find at least one assignment for showing that the input is SAT.
- **If input is a UNSAT formula:** We can intuitively notice that if the boolean formula is UNSAT, there must be a contradiction in one of the resolving steps , probably in the BCP step.
- Let's say if x is the unit literal selected to perform BCP, so we try to get rid of all !x(negation of x) values form the formula, in this procedure if any of the resulting clauses is empty, intuitively indicating that there is something like a resultant (x and !x) that can never lead to UNSAT.
- As we perform assignments to one variable after another, there would be a stage where DPLL SAT solver would reach a contradicting resolving state, that will be decided as UNSAT. At worst, DPLL might take O(2^n) to decide that input is UNSAT.

So with DPLL SAT solver, if the input is a SAT formula, the solver will definitely find a satisfying assignment for the input formula and terminate. On the other side given an unsatisfiable input, the solver will eventually prove that it is unsatisfiable and terminate. Thus we can decide that DPLL is a decision procedure for Boolean satisfiability problems.

5. Question 5: Tseitin Transformation and Conjunctive Normal Form

Tseitin's transformation converts formula F to equisatisfiable formula F' in CNF with only a linear increase in size. I.e F is satisfiable if and only if F' is satisfiable.

The following are the steps that are followed in Tseitin transformation:

- **Step 1:** Introduce a new variable p_G for every subformula G of F, if G is not an atom i.e has only one positive literal.
- **Step 2:** Consider each subformula G : G 1 ◦ G 2 (◦ arbitrary boolean connective)
- **Step 3:** Convert p G ↔ p G 1 ◦ p G 2 to equivalent CNF (by converting to NNF and distributing ∨'s over ∧'s).

Tseitin transformation is a polynomial time algorithm, But how?

- We can notice that from the above mentioned steps, in the first step Tseitin transformation introduces new variables for every sub formal which is always polynomial is time, as it cannot assign more number of new variables than the existing number of variables.
- In the third step, the formula is converted to NNF and then CNF.
- Assume that our formula is represented as a well formed DAG. In such cases when the formula is converted to NNF, we traverse through each node only a few number of times along with introducing only a few new nodes. The same happens when traversing DAG for converting NNF to CNF, which can also finish the conversion in polynomial time.
- In total the time taken to complete the 3 steps will be a polynomial time.