Bonus question description:

How to build and run: Using the same commands from the pa2 description

```
$ cd bonus && mkdir build && cd build && cmake ../
```

After running make command in the /bonus/build folder, an executable is created in the build folder with name “bonus”.

Algorithm description:

The flow of control is similar to pa2 assignment skeleton code, where we call a function “satCallingMiniSat” in the “satSolver.cc” file. That is the input formula is first parsed to a tree and then a Tseitin Transformation is performed on it, which is then passed as input to “satCallingMiniSat”.

In “satCallingMiniSat” we take the clauses in vector<vector<int>> data type and make it in the form vector<vector<Lit>> data type.

An instance of Solver class is created and the “solve” member function is called to give the desired output.

Now there are 2 new files added for the implemented SAT solver.

“SolverTypes.h” – It contains a class “Lit” which holds a literal number and the sign it holds in a particular clause.

For eg.: if the clause is as: (a) * (-a), then the two objects of Lit that will be created will hold literal number for ‘a’, which stays the same for the formula, and the sign is given a Boolean value true for the first clause (a) and Boolean value of false for the second clause (-a).

“Solver.h” – It contains “Solver” class which holds the tseitin transformed formula in CNF form and the assignment map which is initially empty.

Class variables (private):

- vector<vector<Lit>> clauses; - tseitin transformed formula in CNF form
- map<long long int, bool> assignment; - variable assignment map

Member functions of Solver class (private):

- chooseVar(vector<vector<Lit>> clauses): this function is called just once to initially when we need to choose the first unit clause which is the new variable created for the formula itself in the Tseitin transformation, this will return the first literal which appears as unit clause in the “clauses” input.

Member functions of Solver class (public):

- mkLit(…): helper function to create a new Literal
- addClause(…): adding clause to the formula
- print(): debug code to print current formula state
- solve(): this function returns true if the formula in the Solver instance is satisfiable or returns false if the formula in the Solver instance is unsatisfiable.

Other functions:

- print(…): debug code to print current formula state, takes as input the current formula
- chooseVar(vector<vector<Lit>> clauses, map<long long int, bool> assignment): This function takes as input the current modified formula and the assignment map. It will choose a literal such that it appears maximum number of times and is still not assigned a value in the assignment map, and return the literal number.
- bcp(vector<vector<Lit>> clauses, map<long long int, bool> assignment): This function is implements the following pseudocode for bcp (Boolean Constraint Propogation):

```
1: for each pair (v, u) in A
2:       for each clause C in ϕ
3:             if ((v occurs positively in C and u is the value true) OR
                    v occurs negatively in C and u is the value false)) then
                    mark the clause C as satisfied and remove from ϕ
4:             if ((v occurs positively in C and u is the value false) OR
                    v occurs negatively and u is the value true)) then
                    mark the literal as false, and shrink the clause C by dropping v
5:             if C becomes unit, add it to the map A with appropriate value and remove it from ϕ
6: Return the modified formula //Note that the formula ϕ is being modified by either dropping clauses (step 3 and step 5) or dropping literals (step 4)
```

- plp(vector<vector<Lit>> clauses, map<long long int, bool> assignment): This function is implements the following pseudocode for plp (Pure Literal Propogation):

```
PLP ( Formula F , Assignment & A ) {
   L = [ ]
   for every variable v in the formula F {
      if ( v occurs only positively )
         A[v] = True
         L.append( v )
      else if ( v negatively in the entire formula F )
         A[v] = False
         L.append( v )
   }
   for each variable u in L  {
      for each clause C in F
         if ( u occurs in C )
            drop C from F
   }
}
```

- dpll(vector<vector<Lit>> clauses, map<long long int, bool> assignment): This function is implements the following pseudocode for dpll:

```
bool DPLL(CNF φ, AssignMap A)
{
  1. φ' = BCP(φ,A)
  2. φ'' = PLP(φ')
  3. if(φ'' = ⊤) then return SAT;
  4. else if(φ'' = ⊥) then return UNSAT;
  5. p = choose_var(φ'');
  6. if(DPLL(φ'',A[p ↦ ⊤])) then return SAT;
  7. else return (DPLL(φ'',A[p ↦ ⊥]));
}
```

To check if any of the BCP or PLP returned a true or false, a new Lit object is used with literal number 0 (which is not a literal present in the CNF formula), and the sign value of the object is set to true if the formula is satisfied, or false if the formula is UNSAT.
The return formula is checked both after BCP and the PLP step and the corresponding Boolean value is returned to “satCallingMiniSat” function from where we call the “solve” member function. If the Boolean output from dpll is true, then the formula is SAT, otherwise its UNSAT.