Project Four: Satisfiability (SAT)

Out: Nov. 5, 2020; Due: Nov. 22, 2020

I. Motivation

To give you experience in implementing abstract data types (ADTs), using interfaces (abstract base classes), and using interface/implementation inheritance.

II. Introduction

In this project, you will implement two simple methods for deciding the **satisfiability (SAT)** of a Boolean expression, which are by **enumeration** and **deduction**. We have provided a set of slides on introduction to SAT in the `Project-4-Related-Files.zip`. Please read it first. However, in terms of the detailed procedure, you should follow what is specified in this description.

Boolean algebra is a branch of algebra in which the values of variables are *true* and *false*, usually denoted as 1 and 0. The three main operations in Boolean algebra are the conjunction (*and*), the disjunction (*or*), and the negation (*not*). The conjunction (denoted as & or *) is a binary operation and it only produces *true* when both operands are *true* (otherwise it is *false*). The disjunction (denoted as | or +) is a binary operation and it produces *true* if at least one operand is *true* (otherwise it is *false*). The negation (denoted as ~, –, or a bar over the operand) is a unary operation that simply negates the value of the operand. For more information about Boolean algebra, please search online (e.g., https://en.wikipedia.org/wiki/Boolean_algebra).

With a given Boolean expression, we sometimes want to know whether there exists **at least one assignment** of all the variables that lets the expression evaluate to *true*. If such an assignment exists, this formula is called **satisfiable**, or **SAT** for short. Otherwise, if there is no such an assignment, the formula is called **unsatisfiable**, or **unSAT** for short. For convenience, we will write all our expressions in a standard form called **conjunctive normal form (CNF)**, or simply product-of-sums. The following is an example of CNF

$$\Phi = (a + c)(b + c)\left(\bar{a} + \bar{b} + \bar{c}\right),$$

where

- Each sum is called a **clause**. There are 3 clauses in this expression.

- Each symbol (regardless of its form) in the expression is called a **variable**, e.g., a, b, c.
- Each variable in true form is called a **positive literal**, e.g., a, b, c.
- Each variable in complement form is called a **negative literal**, e.g., $\bar{a}, \bar{b}, \bar{c}$.
- **Note**: An expression can have several literals of one variable.

The above example is SAT as variable assignment $(a, b, c) = (0,0,1)$ lets Φ evaluate to *true*.

The CNF is preferred in SAT problem, since an arbitrary Boolean expression can be transformed into this form, and this form simplifies the process of evaluation in the following ways:

- As long as one clause evaluates to *false* (0), the whole expression is *false*.
- In order for the whole expression to yield *true* (1), every clause has to be *true*. In other words, at least one literal in each clause has to be *true*.

The two methods used for deciding SAT will be introduced later.

III. Programming Assignment

You will provide one or more implementations of four separate abstractions for this project: a single clause, a single CNF expression, an SAT solver, and a driver. All files referred in this description are located in the `Project-4-Related-Files.zip`.

You may copy them to your own directory, but **do not modify them in any way**. This will help ensure that your submitted project compiles correctly.

1. The Clause ADT

Your first task is to implement the following ADT representing a single clause:

```
/* Constants */
// Max number of literals in a clause
const unsigned int MAX_LITERALS = 50;


class Clause
/*
// Type: Clause
// ------------------
// The type Clause is used to represent a clause.
// Its attributes consist of:
// * literals: an array of literals.
// * numLiterals: number of literals in the clause,
//   e.g. numLiterals = 3 for a clause = (x0 | x1 | -x2).
// * value: used to represent 2 special states of a
//   clause: (0) and (1).
//   When clause = (0), numLiterals = 0 and value = 0.
//   When clause = (1), numLiterals = 0 and value = 1.
//   Otherwise, numLiterals > 0 and value = -1.
*/
{
    Literal literals[MAX_LITERALS];
    unsigned int numLiterals;
    int value;

public:
    Clause();
    // EFFECTS: constructs an empty clause.

    void addLiteral(Literal literal);
    // MODIFIES: this
    // EFFECTS: add a literal to the current clause.

    int evaluate(const int val[]) const;
    // REQUIRES: the i-th element in val[] represents the value of
    // the i-th variable: 1 (true), 0 (false), or -1 (unknown).
    // The length of val[] should be exactly equal to the number of
    // variables (represented by CNF::numVars).
    // EFFECTS: given an assignment of variables, evaluate and
    // return the value of the clause:
    // 1: the clause evaluates to true,
    // e.g. for clause = (x0 | -x1 | x2), if val = [0,0,0],
    // clause.evaluate(val) = 1
    // 0: the clause evaluates to false,
    // e.g. for clause = (x0 | -x1 | x2), if val = [0,1,0],
    // clause.evaluate(val) = 0
    // -1: the value of the clause is unknown,
    // e.g. for clause = (x0 | -x1 | x2), if val = [0,1,-1],
    // clause.evaluate(val) = -1

    void eliminate(Literal literal);
    // MODIFIES: this
    // EFFECTS: given a literal, simplify the current clause by
    // assuming that this literal is true. The order of the other
    // literals in the clause keeps unchanged.

    void print() const;
    // EFFECTS: print the clause in the following form:
    // If numLiterals = 0, simply print (0) or (1) according to the
    // "value" attribute.
    // Otherwise, print the CNF in form like: (-x0 | x1 | -x2).
    // Note that there should be a white space between every
    // literal and “|”, but there should be no space between
    // a literal and a parenthesis.
    // If there is only a single literal, print like: (x0).
    unsigned int getNumLiterals() const;
    // EFFECTS: return numLiterals of the Clause instance.

    Literal getLiteral(unsigned int i) const;
    // EFFECTS: return the i-th literal of the Clause instance, i.e.,
    // (this->literals[i]).

    ~Clause();
    // EFFECTS: destruct a Clause instance. If there's no extra
    // operations needed, just leave it empty.
};
```

The `Clause` ADT is specified in `clause.h`. The `Clause` ADT depends on the following `Literal` type:

```
struct Literal
/*
// Type: Literal
// ------------------
// The type Literal is used to represent a literal.
// Its attributes consist of:
// * ID: variable ID of the literal, e.g., 0 for literals x0 and -x0,
//   1 for literals x1 and -x1, etc.
// * negative: represent the form of the literal in the clause,
//   i.e., 'true' for negative form, e.g., -x0;
//         'false' for positive form, e.g., x0.
*/
{
    unsigned int ID;
    bool negative;
};
```

The type `Literal` is also declared in `clause.h`. It has an ID to represent its variable ID, and a Boolean value to indicate whether it is positive or negative.

You are asked to put your implementation of the `Clause` ADT in a file named `clause.cpp`.

2. The CNF ADT

Your second task is to implement the following ADT representing a CNF expression:

```
/* Constants */
// Max number of clauses in a CNF expression
const unsigned int MAX_CLAUSES = 100;

// Max number of variables in a CNF expression
const unsigned int MAX_VARS = 50;

class CNF
```

```
/*
// Type: CNF
// ------------------
// The type CNF is used to represent the conjunctive normal form of a
// Boolean expression.
// Its attributes consist of:
// * clauses: an array of clauses.
// * numClauses: number of clauses in the CNF expression,
//   e.g. numClauses = 2 for CNF expression (x0 | -x1) & (x0).
// * numVars: number of variables in the CNF expression,
//   e.g. numVars = 2 for CNF expression (x0 | -x1) & (-x0 | x1)
*/
{
    Clause clauses[MAX_CLAUSES];
    unsigned int numClauses;
    unsigned int numVars;

public:
    CNF();
    // EFFECTS: construct an empty CNF expression.

    void addClause(Clause cls);
    // MODIFIES: this
    // EFFECTS: add a clause to the current CNF expression.

    void eliminate(Literal literal);
    // MODIFIES: this
    // EFFECTS: given a literal, simplify the current CNF expression
    // by assuming that this literal is true.

    int evaluate(const int val[]) const;
    // REQUIRES: the i-th element in val[] represents the value of the
    // i-th variable: 1 (true), 0 (false), or -1 (unknown).
    // The length of val[] should be exactly equal to the number of
    // variables.
    // EFFECTS: given an assignment of variables, evaluate and
    // return the value of the CNF expression:
    // 1: the CNF expression evaluates to true,
    // e.g. for cnf.clauses = (x0 | -x1 | x2) & (x0 | x1 | -x2),
    // if val = [0,0,0], cnf.evaluate(val) = 1
    // 0: the CNF expression evaluates to false,
    // e.g. for cnf.clauses = (x0 | -x1 | x2) & (x0 | x1 | -x2),
    // if val = [0,0,1], cnf.evaluate(val) = 0
    // -1: the value of CNF expression is unknown,
    // e.g. for cnf.clauses = (x0 | -x1 | x2) & (x0 | x1 | -x2),
    // if val = [0,0,-1], cnf.evaluate(val) = -1


    bool hasUnit() const;
    // EFFECTS: check whether there exists a unit clause in this CNF
    // expression.
    Literal getUnit() const;
    // REQUIRES: the current CNF expression has at least one unit
    // clause.
    // EFFECTS: return the only literal of the leftmost unit clause.

    CNF unitPropagate(Literal unit, int val[]) const;
    // MODIFIES: val
    // EFFECTS: given a literal and assuming it to be true, infer the
    // value of its variable, and then return a simplified CNF
    // expression with that variable eliminated from "this" CNF
    // expression. Note that "this" CNF expression is not
    // changed.
    // E.g. cnf.clauses = (-x0) & (x1 | x2), val=[-1,-1,-1], you
    // should first set val=[0,-1,-1], and return a simplified CNF
    // expression with variable x0 eliminated from the current CNF
    // expression.

    void print() const;
    // EFFECTS: print the CNF in form like: (-x0) & (x1 | x2) & (1).
    // See Section Driver Program for details on the output format.

    unsigned int getNumVars() const;
    // EFFECTS: return the number of variables in the CNF expression.

    unsigned int getNumClauses() const;
    // EFFECTS: return the number of clauses in the CNF expression.

    void setNumVars(unsigned int n);
    // MODIFIES: this
    // EFFECTS: set the number of variables of the CNF expression.

    ~CNF();
    // EFFECTS: destruct a CNF instance. If there's no extra operation
    // needed, just leave it empty.
};
```

The `CNF` ADT is specified in `cnf.h` The `CNF` ADT depends on the `Clause` type, and includes `clause.h`. You are asked to put your implementation of this ADT in a file named `cnf.cpp`.

3. The Solver Interface

Your third task is to implement two different methods for solving the SAT problem. The interface for a `Solver` is:

```
class Solver
/*
// Type: Solver
// ------------------
// The type Solver is an abstract base class that provides the SAT
// solver interface.
// You need to implement its 2 derived classes in solver.cpp.
*/
{
public:
    virtual void solve(const CNF &cnf) = 0;
    // EFFECTS: handle the "SAT" command (defined in the
    // Driver Program section).
    // Solve if the CNF expression is satisfiable, or SAT;
    // and output corresponding solution info.

    virtual void evaluate(const CNF &cnf, const int val[]) = 0;
    // REQUIRES: value in array val[] is either 0, 1, or -1.
    // The length of val[] is equal to the number of variables.
    // EFFECTS: handle the "EVAL" command (defined in the
    // Driver Program section).
    // Evaluate the value of the CNF expression given an
    // assignment and output the solution info.

    virtual ~Solver() {};
    // Added to suppress compiler warning.
    // If you do not need to do anything when destructing,
    // just ignore it.
};
```

The `Solver` ADT is specified in `solver.h` The `Solver` ADT depends on the `CNF` type, and includes `cnf.h`. You are to implement two different derived classes from this interface.

The first derived class is the **enumerative solver**, which simply enumerates all possible assignments and test whether at least one of them can make the expression yield 1. In this case, the final solution is SAT; otherwise, the final solution is unSAT.

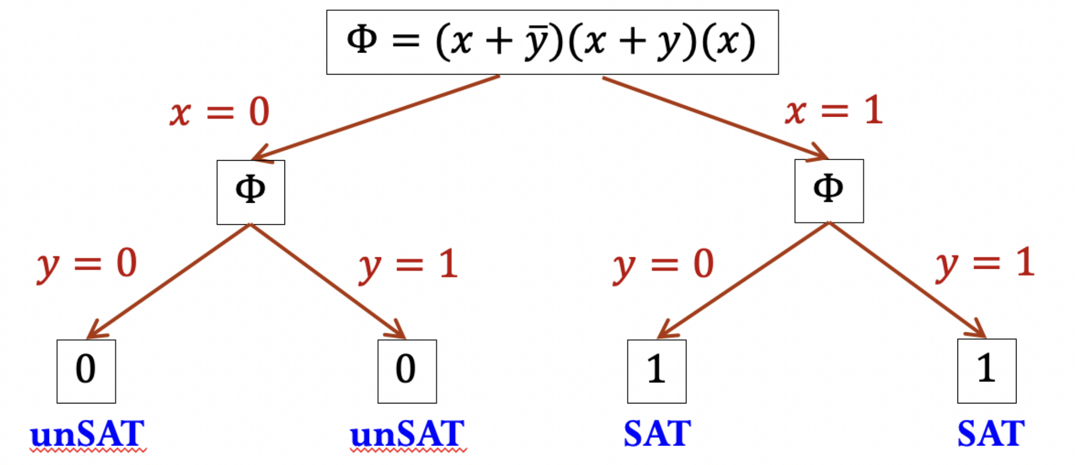

Figure 1. An example of enumerative method.

Fig. 1 shows an example of solving the SAT problem given a Boolean expression with the enumerative solver. The expression is not evaluated until values of all the variables are determined. Note that in some cases, you can evaluate the expression without knowing the values of all the variables. For example, in the example shown in Fig. 1, the expression is 0 when we know x is 0. However, the enumerative solver will only make evaluation after all the variables are determined.

In this project, the enumerative solver should be able to tell whether a CNF expression is SAT. If it is unSAT, the solver simply tells the user. If it is SAT, it needs to print the complete assignment with the **minimum lexicographic order**. Since we give all variables in the form of $x_0, x_1, x_2, \dots, x_{n-1}$, we can represent the assignment of variable values as a tuple: $(v_0, v_1, v_2, \dots, v_{n-1})$, where v_i can only be 0 (*true*) or 1 (*false*). For two assignments $(a_0, a_1, \dots, a_{n-1})$ and $(b_0, b_1, \dots, b_{n-1})$, we compare them from the first element. If the two elements from both assignments are equal, we proceed on to compare the next. When we find the smallest *i* such that $a_i < b_i$ ($a_i > b_i$), we say that the first assignment is smaller (larger) than the second one.

The second derived class is the <u>**deductive solver**</u>. The deductive method can be summarized as following steps:

1. First print: "`Start deductive solver for:`", and then on a new line, the CNF expression you are going to solve. Initialize a **decision variable set** as empty.
2. Check whether there are unit clauses in the CNF expression:
   a) If there is no unit clause, continue to step 3.
   b) If there are unit clauses, pick the leftmost one (i.e., the unit clause in the `CNF::clauses` array with the smallest index) and do unit propagation. Print the deduction you made (e.g. "`Unit propagate x0 = 1:`"), and then on a new line, the expression after this propagation is applied. Continue to the beginning of step 2.
3. Check the status of the current CNF expression:
   a) If you cannot determine SAT/unSAT, continue to step 4.
   b) If the expression is unSAT, continue to step 5.
   c) If the expression is SAT, tell the user and end the algorithm.
4. Pick the unassigned variable with the smallest index, assign it with 0, and simplify the expression. Add this variable to the **decision variable set**. Print your decision (e.g. "`Make decision x2 = 0:`") and then on a new line, the expression after simplification. Then continue to step 2.
5. Check if the decision variable set is empty. If yes, the entire CNF expression is unSAT. Tell the user so and end the program. Otherwise, pick the variable that is **most recently added into the decision variable set**, assign it with 1, and simplify the expression. This can be viewed as **reversing the most recent decision**. Remember to remove that variable from the

decision variable set. Print your new decision (e.g. "`Reverse previous decision x2 = 1:`") and then on a new line, the expression after simplification. Then continue to step 2.

The main strategy we used here to do deduction is the **unit clause rule**. A clause is called a **unit clause** if it has **exactly one** **<u>unassigned literal</u>**. According to the property of CNF, this unassigned literal must be *true* to ensure SAT. Thus, we can determine the value of its corresponding variable (i.e., *false* if the literal is negative and *true* if the literal is positive), and try to eliminate this variable in the entire expression. This process is called **unit propagation**.

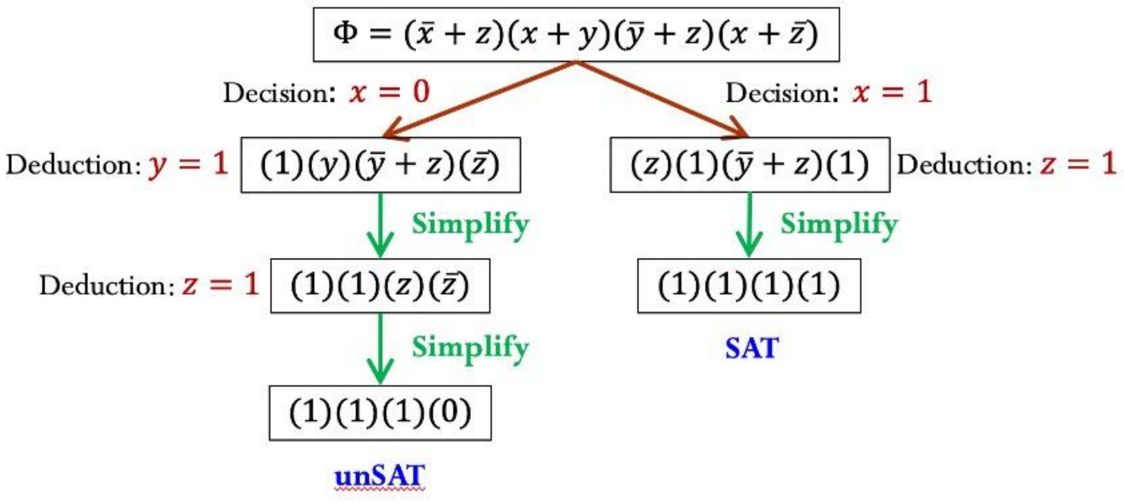

Figure 2. An example of deduction.

In Fig. 2, we show the process of a deductive solver. At first, since there is no unit clause, therefore we need to make a decision. We choose the decision variable as x and set its value to 0. Then, we simplify the CNF expression based on this decision. After that, we find that our CNF expression has a unit clause (y). In order to make the whole CNF expression *true*, y has to be 1. After we deduce and eliminate y, we find that there are two unit clauses: (z) and $(\bar{z})$. We deduce based on the first one (z). In order to make the whole CNF expression *true*, z has to be 1. After deduction and simplification, the expression becomes 0. This means that under the decision that $x = 0$, the CNF expression cannot be SAT. Therefore, we should reverse the most recent decision, that is, further explore the SAT possibility when $x = 1$. This leads to the right branch shown in the figure. After simplification, we find that the new CNF expression has a unit clause (z). After deduction and simplification, we find the result is SAT. Thus, the original CNF expression is SAT.

The code for both solvers must be implemented in a file named `solver.cpp`. You must also declare a static global instance of each of the two solvers you implement in your `solver.cpp` file. Finally, you should implement the following “access” functions that return pointers to each of these two global instances in your `solver.cpp` file.

```
extern Solver *getEnum();
// EFFECTS: returns a pointer to a "enumerative solver"
// You should only declare one static global instance of EnumSolver

extern Solver *getDeduct();
// EFFECTS: returns a pointer to a "deductive solver"
// You should only declare one static global instance of DeductSolver
```

Hint: You can define your own data members and methods in derived classes, as long as you do not modify `solver.h` and implement all methods declared in the `Solver` interface. For both enumerative and deductive solver, one recommended way is to implement them recursively. If you want to write a recursive function, you can define a helper function in your derived class.

4. The Driver Program

Finally, you are to implement a driver program that can be used to deal with user input and invoke corresponding functions to solve the problem.

You are asked to put your implementation of this driver program in a file named `main.cpp`.

The driver program, when run, takes only one argument that indicates which solver the user wants to use. The argument is one of the two strings, “ENUM” and “DEDUCT”, denoting the enumerative solver and the deductive solver, respectively. Suppose you compile your program and the output executable is called `p4`. The command you use to run it should be like:

```
    ./p4 ENUM
```

You can assume that the argument input by the user is always valid (either “ENUM” or “DEDUCT”).

Then your program starts to read input from the **standard input**. The user will first input information about the CNF expression. The first line of the input would be two integers: n, representing the number of variables, and m, representing the number of clauses you are going to read next. **You can assume the number of variables and the number of clauses will not exceed the maximum constants we set for you (in the source code).**

After the first line, there are m lines of clauses. All these clauses compose the final CNF expression. A sample input would be like:

```
5 6
x0 | -x4 | x3
```

```
x0 | -x3 | x2 | x4
x2 | x4 | x1
x3 | -x1
-x3
x0
```

There are 5 variables and 6 clauses. They form the following CNF expression:

$$\Phi = (x_0 + \overline{x_4} + x_3)(x_0 + \overline{x_3} + x_2 + x_4)(x_2 + x_4 + x_1)(x_3 + \overline{x_1})(\overline{x_3})(x_0)\,.$$

Each literal is written in the form of `"x"` followed by a non-negative integer, and there is an extra `"-"` at the beginning if it is in the negative form. **You can assume the symbol is always "x", and the indices are 0, 1, …, $n-1$, if there are n variables.** Expressions like $(x_0 + x_3)(x_0 + \overline{x_3} + x_2)$ will not be provided (but can occur in your intermediate calculation), since the indices do not follow the above rule.

Then, after the CNF expression is loaded into the program, your program should start waiting for user commands to conduct various kind of operations. There are four types of user commands: `EXIT`, `PRINT`, `EVAL`, and `SAT`. They are described below (we will continue to use the above example for illustration):

- **`EXIT`**
  Exit the solver.

  Input:
  ```
  EXIT
  ```

  Simply exit the program and print "Bye!" with newline:
  ```
  cout << "Bye!" << endl;
  ```

- **`PRINT`**
  Print the CNF expression you have loaded on a single line in a complete form. There should be a white space between every literal and `"|"`, also between every parenthesis and "`&`". But there should be no space between a literal and a parenthesis.

  Input:
  ```
  PRINT
  ```
  Sample output:
  ```
  (x0 | -x4 | x3) & (x0 | -x3 | x2 | x4) & (x2 | x4 | x1) & (x3 | -x1)
  & (-x3) & (x0)
  ```

- **`EVAL`**

The command `EVAL` takes an assignment of all the variables as input and evaluates the CNF expression for that assignment. The values are given in the order of variable index, and is either 0, 1, or -1. **You can assume the input is always valid.**
Note: The variable assignment is provided in a new line after the command EVAL. There is a white space after the colon in the output, and there is no punctuation at the end. Also, "True", "False" and "Unknown" are capitalized.

Sample input 1:
```
EVAL
1 0 1 0 0
```
Sample output 1:
```
The value of the Boolean expression is: True
```

Sample input 2:
```
EVAL
1 1 0 1 1
```
Sample output 2:
```
The value of the Boolean expression is: False
```

Sample input 3:
```
EVAL
-1 1 0 -1 1
```
Sample output 3:
```
The value of the Boolean expression is: Unknown
```

- **`SAT`**

  Decide whether the current CNF expression is SAT or unSAT using the type of solver assigned by the program argument. For the enumerative solver, you need to print the solution with the **minimum lexicographic order** after you say it is SAT. For the deductive solver, you need to output the detailed process of deduction.

  Input:
  ```
  SAT
  ```
  Sample output (enumerative SAT):
  ```
  The expression is SAT with one solution:
  1 0 0 0 1
  ```

  Sample output (deductive SAT):
  ```
  Start deductive solver for:
  (x0 | -x4 | x3) & (x0 | -x3 | x2 | x4) & (x2 | x4 | x1) & (x3 | -x1)
  & (-x3) & (x0)
  Unit propagate x3 = 0:
  ```

```
(x0 | -x4) & (1) & (x2 | x4 | x1) & (-x1) & (1) & (x0)
Unit propagate x1 = 0:
(x0 | -x4) & (1) & (x2 | x4) & (1) & (1) & (x0)
Unit propagate x0 = 1:
(1) & (1) & (x2 | x4) & (1) & (1) & (1)
Make decision x2 = 0:
(1) & (1) & (x4) & (1) & (1) & (1)
Unit propagate x4 = 1:
(1) & (1) & (1) & (1) & (1) & (1)
The expression is SAT!
```

We then give another unSAT sample CNF expression:

$$\Phi = (x_0 + x_1)(x_0 + \overline{x_1})(\overline{x_0} + x_1)(\overline{x_0} + \overline{x_1})$$

to see the sample output when the expression is unSAT.

Sample output (enumerative unSAT):

```
The expression is unSAT!
```

Sample output (deductive unSAT):

```
Start deductive solver for:
(x0 | x1) & (x0 | -x1) & (-x0 | x1) & (-x0 | -x1)
Make decision x0 = 0:
(x1) & (-x1) & (1) & (1)
Unit propagate x1 = 1:
(1) & (0) & (1) & (1)
Reverse previous decision x0 = 1:
(1) & (1) & (x1) & (-x1)
Unit propagate x1 = 1:
(1) & (1) & (1) & (0)
The expression is unSAT!
```

IV. Implementation Requirements and Restrictions

- You may `#include <iostream>`, `<sstream>` `<iomanip>`, `<string>`, `<cstring>`, `<cstdlib>`, and `<cassert>`. No other system header files may be included, and you may not make any call to any function in any other library.
- You may not use the `goto` command.
- You may not have any global variables in the driver. You may use global state in the class implementations, but it must be static and (except for the two solvers) const.

- You may assume that functions are called consistently with their advertised specifications. This means you need not perform error checking. However, when testing your code in concert, you may use the `assert()` macro to program defensively.
- The input is read from standard input. When you are testing, you can write your test case in a file and use input file redirection to save time.
- We may compile part of your source codes with our driver program to test whether you have implemented the inheritance correctly.

V. Source Code Files and Compiling

There are three header files (`clause.h`, `cnf.h`, and `solver.h`) and six sample input/output files (`sat.in`, `sat_enum.out`, `sat_deduct.out`, `unsat.in`, `unsat_enum.out`, and `unsat_deduct.out`) in `Project-4-Related-Files.zip` from our Canvas Resources:

You should copy the header files into your working directory. **DO NOT modify them!**

You need to write four other C++ source files: `clause.cpp`, `cnf.cpp`, `solver.cpp`, and `main.cpp`. They are discussed above and summarized below:

`clause.cpp`:	your `Clause` ADT implementation
`cnf.cpp`:	your `CNF` ADT implementation
`solver.cpp`:	your two `Solver` ADT implementations
`main.cpp`:	your driver program to deal with input

After you have written these files, you can type the following command in the terminal to compile the program:

```
g++ -std=c++11 -Wall -o p4 main.cpp clause.cpp cnf.cpp
solver.cpp
```

This will generate a program called `p4` in your working directory. In order to guarantee that the TAs compile your program successfully, you should name you source code files exactly like how they are specified above.

VI. Hints & Tips

- The SAT problem is a famous NP-complete problem, which means people cannot find an efficient way to solve it in polynomial time so far. You can search online to learn more about this problem.

- Since it is an NP-complete problem, we can only find different ways to improve the process of finding solutions. You may have already realized that the deductive solver is simply an improvement of the enumerative solver. The deductive solver is actually a simplified version of the DPLL algorithm for solving SAT problems.
- Theoretically, the deductive approach can take as much time as the enumerative one in some cases. However, for most cases, the deductive approach does avoid unnecessary checking. For example, if there are some conflicting clauses in the expression, the deductive solver can find it at the very beginning, while the enumerative solver needs to go through all possibilities.
- The functions provided in `clause.cpp` and `cnf.cpp` are to help you implement the final functions more easily. You may not use all of them in your implementation of the solvers, but you still need to implement them correctly in case that we will test them individually.
- The interfaces are designed to be layered (lower layers cannot call functions from higher layers). Therefore, you may want to implement them from lower to higher.

VII. Testing

For this project, you should write individual, focused test cases for all the ADT implementations. For these ADTs, determine the required behaviors of the implementation. Then, for each of these behaviors:

- Determine the **specific** behavior that the implementation must exhibit.
- Write a program that, when linked against the implementation of the ADT, tests for the presence/absence of that behavior.

You may want to test your program part by part. For example, you may first write some extra code to test the functions in the `Clause` ADT individually. Then after confirming that the `Clause` ADT is correct and can deal with all kinds of input, you can proceed on to write other code to test you `CNF` ADT, and then the `Solver` implementations, etc.

Below is an example of code that tests a hypothetical "integer add" function (declared in `addInts.h`) with an "expected" test case:

```
// Tests the addInts function
#include "addInts.h"
int main() {
    int x = 3;
    int y = 4;
    int answer = 7;
    int candidate = addInts(x, y);
    if (candidate == answer) {
        return 0;
    } else {
        return -1;
    }
}
```

You may write code like this to test individual functions before you complete the entire project. You can let your main function return 0 if the behavior is as expected, and return other value if it is not. Often, -1 is usually used to denote failure and 0 is used to denote success.

In Linux you can check the return value of your program by typing

```
echo $?
```

immediately after running your program. You also may find it helpful to add error messages to your output.

You should write a collection of `Clause`/`CNF`/`Solver` implementations with different, specific bugs, and make tests to identify the incorrect code.

Also, you may want to test your entire program after you completed all parts. We have supplied two correct sets of input and output, which are simply pure text files. In `sat.in` (`unsat.in`), there is a sample CNF expression and a sequence of instructions. The correct output of two solvers are in `sat_enum.out` (`unsat_enum.out`) and `sat_deduct.out` (`unsat_deduct.out`), respectively. To test your program, type the following into the Linux terminal once it has been compiled (we use `sat.in` as an example):

```
./p4 ENUM < sat.in > test_enum.out
diff test_enum.out sat_enum.out
./p4 DEDUCT < sat.in > test_deduct.out
diff test_deduct.out sat_deduct.out
```

If the `diff` program reports any differences at all, your code has a bug.

VIII. Submitting and Due Date

You should submit four source code files `clause.cpp`, `cnf.cpp`, `solver.cpp`, and `main.cpp`. (You do not need to submit a `Makefile` for this project.) These files should be submitted as a tar file via the online judgment system. See the announcement from the TAs for details about submission. The due time is 11:59 pm on Nov. 22, 2020.

IX. Grading

Your program will be graded along three criteria:

1. Functional Correctness
2. Implementation Constraints
3. General Style

Functional Correctness is determined by running a variety of test cases against your program, checking against our reference solution. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. For example, significant code duplication will lead to General Style deductions.