Implement the DPLL algorithm in C++. Start with the code provided in the “hw.tar” archive file. To unpack the archive, type “tar xvf hw.tar”. The code as provided includes a Makefile, a main program (in “sat.cpp”), and pure virtual classes for Clause and Formula. There is also a header file called “debug.h” that provides a macro called “Assert” that you can use for debugging purposes. The statement “Assert(x)” checks that condition x is true and gives an error message if it is not. There is an example of its use in “sat.cpp.” There is also a subdirectory called “data” that includes a familiar set of sample input files.

1. You will need to start by providing classes ClauseImpl and FormulaImpl that inherit from Clause and Formula respectively and implement their pure virtual functions. To do this, think about what data structure you could use to hold the clauses and literals (hint: use the STL), and implement the new classes in files named clause_impl.h, clause_impl.cpp, formula_impl.h, formula_impl.cpp. Don’t forget to add these to the Makefile. When you’ve finished this step, you should be able to compile and run on a sample input, and have the program print the clauses back:

Example:

```
> make
 g++ -g -c sat.cpp -o sat.o
 g++ -g -c formula.cpp -o formula.o
 g++ -g -c clause.cpp -o clause.o
 g++ -g -c debug.cpp -o debug.o
 g++ -g -c clause_impl.cpp -o clause_impl.o
 g++ -g -c formula_impl.cpp -o formula_impl.o
 g++ sat.o formula.o clause.o debug.o clause_impl.o formula_impl.o -o sat
> sat data/hole2
 (-1, -3, 0)
 (-1, -5, 0)
 (-3, -5, 0)
 (-2, -4, 0)
 (-2, -6, 0)
 (-4, -6, 0)
 (1, 2, 0)
 (3, 4, 0)
 (5, 6, 0)
```

2. Next, uncomment the lines in sat.cpp that refer to the Solver. Write a class called Solver with a method called solve (you will need to add files “solver.h” and “solver.cpp”). Aim for an implementation that is not too difficult and at the same time reasonably efficient. You may want to start with an inefficient implementation and incrementally improve it. In fact, in general, with C++, it is a good idea to make small, incremental changes, so you can easily fix and debug problems as soon as they are introduced.
3. One important design decision will be where to put different pieces of functionality, i.e. is “unit propagation” a ClauseImpl method or a FormulaImpl method? (Note: there is not necessarily a right answer here, but try to make your design logical and simple). If it is necessary to avoid downcasting, you may extend the interfaces of Clause and Formula in clause.h and formula.h. You should not add code to the clause.cpp or formula.cpp files. Implementation should be in the clause_impl.cpp and formula_impl.cpp files.

4. If you type “make OPT=1” the compiler will optimize your code. How big a difference does this make?