How it works:

The two basic structures I used in implementing this code are a solver and a clause.  Clause has 3 elements:

**Size** - how many literals are in the clause

**Level\_sat (stands for level satisfied)** - if the clause has a 'true' literal in it, at what level was this literal assigned to be 'true'

**lits[]** - an array of literals that are in the clause.  These are data type 'lit' which is actually just an integer but is used in a different way.  I'll explain what a lit is below, because it is important to understand what it is to understand the algorithms.

A 'lit' is a mapping of the literals in the clause to a positive integer (or zero).  So every literal in the input file is translated into a 'lit' using this formula:     **lit = (2 \* (absolute\_value(number\_in\_input\_file) - 1))**and then **add 1 if the number\_in\_input\_file was negative.**  To illustrate this, if the input file had a clause with the number -4 in it, the lit that would translate to is 7.  I'll show this in steps below:

number\_in\_input\_file = -4

lit = (2 \* (absolute\_value(-4) -1)) + 1     (the  +1 is because the original # was negative)

lit = (2 \* (4-1)) +1

lit = (6) + 1

lit = 7

And here is an input clause, and the lits translated from this clause:

**Input Clause:** 1 -2 -4 -3 2 -1 3 -4

**Lits:** 0 3 7 5 2 1 4 7

So lits are always positive or zero, and every odd lit represents the negation of the even lit before it (e.g. 7 represents -4, and 6 represents 4)

Hopefully this is clear to you, because that is how the clause information is stored.

The other structure is the Solver, which is described below:

Solver:

**Size** - number of variables in the solver (same as the number of variables in the input file)

**cap** - a number used to resize the solver as the number of variables in the decision making process grows.

**tail** - This is a number that indicates which clause in the vecp clauses is the last unsatisfied clause.  All the clauses beyond this one are satisfied.

**cur\_level (current level)** - the current level down the binary tree that the solver is.  The root of the tree is 0.

**satisfied** - this helps me to find out if the solver was satisfied by some variable assignment. It is always false until a satisfying solution is found.

**clauses -** This is a vecp (vector of pointers) of all of the clauses.  vecp is described in vec.h, and was a part of the minisat parser.  It is essentially just a vector of pointers to the memory locations where each "clause" data structure is stored.

**decisions -** This is an array of boolean values.  The index into the array is the lit that the value is representing (so decisions[5] is the decision for lit 5.  See the explanation of lits above).  The value for a lit goes to 'true' only if it is chosen as the next decision down the binary tree.  This is used for backtracking - if you backtrack one level and see that both the negative and positive literals have been decided at that level, you know you have to backtrack again.

**level\_choice** - this array of lits just keeps track of which lit was assigned to 'true' at each level of the tree.  It only stores the first choice for each level, not the assignments made due to unit clauses (see the algorithms description).  So level\_choice[4] is the first literal assignment to 'true' at level 4 of the decision tree.

**assigns -** This is an array of lbools (this is a datatype that has three values, l\_True, l\_False, and l\_Undefined).  Like 'decisions', the index into this array is representative of the lit.  This array keeps track of **all** the assignments to literals so far, including assignments made during unit clause propagation.  If the solver is solved, this array has the solution.  So assigns[31] = l\_True means that lit 31 has been assigned the value of 'true' currently.

**levels -** this array of integers just keeps track of the level that each lit was assigned (so levels[22] = 4 means that lit 22 was assigned to true).  It is -1 if the lit hasn't been assigned yet.

**counts -** This keeps track of the number of occurrences of each literal in the current list of clauses that have not yet been satisfied.  It doesn't count literals in satisfied clauses, or literals in unsatisfied clauses that are already assigned to be FALSE.  This information is used by the algorithm to make a decision (see algorithms description).

I used the very basic DPLL algorithm.  The algorithm goes like this:

**repeat forever** {

**pick a literal to assign to 'true'** ( I used the heuristic of picking the clause that is most frequent in the set of clauses)

**assign it** (figure out what clauses are satisfied because of this assignment. Keep track of this.)

**if there is a conflict because of this assignment, go undo what you've done** till you reach a point in the decision tree where the other side of the tree hasn't been explored yet. Start over and make that assignment.

**if not, propagate unit clauses** (find clauses with only one unassigned literal and all others assigned to false.  You have to assign this literal to true because if it is false the whole clause will become false, which is a conflict.  Keep searching for unit clauses and assigning them till you can't find any more).

**if there is a conflict because of any unit clause assignments, go undo what you've done** till you reach a point in the decision tree where the other side of the tree hasn't been explored yet. Start over and make that assignment.

**If there were no conflicts, repeat from the top**

**}**

Along the way, you just check if the solution is found or if there is no solution.  This can happen like so:

**Solution is found:** If after propagating a literal assignment, there are no more unsatisfied clauses, then you solved it!

**Solution doesn't exist:** If while undoing what you've done because of a conflict (AKA backtracking) you can't find a branch of the tree that both sides haven't been explored on, then you've explored the whole binary tree - no solution exists!

**vec.h** - defines the veci (vector of integers) and vecp (vector of pointers) structures.  These were a part of minisat.

**solver.h** - gives function declarations from solver.c that will be used in other files (main.c), and also defines the structure of the solver.

**main.c** - parses the input file, then just runs solver\_solve (the main loop of code) and prints the result to the output file.

**solver.c -** this is where most of the work happens.  The core function is solver\_solve, which just implements the basic pseudocode I've written above.  The clause structure is defined here, and all of the functions used to solve the solver are defined here as well. This is the file that the code spends 99.99% of its time in, because all of the solver code is here.

The basic pseudocode is included above.  The while(true) loop in solver\_solve is the loop forever part.  Just follow the major branches of the function solver\_solve to see how I implemented each part of the pseudocode.