This SAT solver is based on DPLL algorithm. It fills the blanks in the skeleton provided in the lecture slides.

```
DPLL(Θcnf, assign) {

Propagate unit clauses;

if "conflict": return FALSE;

if "complete assign": return TRUE;

"pick decision variable x";

return
    DPLL(Θcnf | x=0, assign[x=0]) ||
    DPLL(Θcnf | x=1, assign[x=1]);
}
```

The SAT solver (satSolver.cc) mainly consists of 4 parts:

- DPLL function
- PLP algorithm
- BCP algorithm
- Helper functions
    - isPureLiteral
    - chooseVar

The input of the DPLL algorithm is a CNF and an assignment map. In the assignment map,

- 1 stands for a variable is set to true
- -1 stands for a variable is set to false
- 0 stands for a variable is not set with any value yet

At the beginning, all variables in the assignment map will be assign with 0.

The DPLL algorithm first performs PLP and BCP iteratively, as shown in the blue square blow. Then, the DPLL algorithm will try different assignments by using backtracking technique, as shown in the red square below.

```cpp
bool DPLL(std::vector<std::vector<int>> cnf, std::map<int, int> assignment) {
    while (true) {
        if(cnf.empty()) {
            return true;
        }

        if(hasEmptyClause(cnf)) {
            return false;
        }

        bool ifBCP = BCP(cnf, assignment);

        bool ifPLP = PLP(cnf, assignment);

        if (!(ifBCP || ifPLP)) {
            break;
        }
    }

    int lit = chooseVar(cnf, assignment);

    assignment[abs(lit)] = -2;// just remove it from unassigned status

    std::vector<std::vector<int>> newCNF;
    std::map<int, int> newAssignment = assignment;

    newCNF = propagate(cnf, lit);
    newAssignment[lit] = 1;
    if (DPLL(newCNF, newAssignment)) {
        return true;
    }

    newCNF = propagate(cnf, -lit);
    newAssignment[lit] = -1;
    if (DPLL(newCNF, newAssignment)) {
        return true;
    }

    return false;
}
```

Since PLP and BCP both use propagation in the input CNF, so we designed a common function called propagate. It intakes the CNF and the literal we want to propagate. It will decide which variables in a clause shall be removed and which clauses in the input CNF shall be dropped. This function returns a simplified CNF.

```cpp
std::vector<std::vector<int>> propagate(std::vector<std::vector<int>> cnf, int lit) {
    std::vector<int> clausesToDrop;

    for (int i = 0; i < cnf.size(); i++) {

        std::vector<int> litToRemove;

        for (int j = 0; j < cnf.at(i).size(); j++) {
            if (abs(lit) == abs(cnf.at(i).at(j))) {
                // should remove that literal in current clause
                if (cnf.at(i).at(j) * lit < 0) {
                    litToRemove.push_back(j);
                    break;
                }
                // should drop current clause from CNF
                if (cnf.at(i).at(j) * lit > 0) {
                    clausesToDrop.push_back(i);
                    break;
                }
            }
        }

        // delete literals in current clause
        for (int k = 0; k < litToRemove.size(); k++) {
            cnf[i].erase(cnf.at(i - k).begin() + litToRemove[k]);
        }
    }

    // drop clauses
    for (int i = 0; i < clausesToDrop.size(); i++) {
        cnf.erase(cnf.begin() + clausesToDrop.at(i) - i);
    }

    return cnf;
}
```

To improve the efficiency of the SAT solver, we introduce PLP algorithm. PLP firstly finds if there are pure literals in CNF. If there are, PLP will call the propagate function to drop literals and remove clauses. After that, the assignment map will be updated. Overall, by doing this, the original CNF is simplified.

```cpp
bool PLP(std::vector<std::vector<int>> &cnf, std::map<int, int> &assignment) {
    bool foundPureLiteral = false;

    std::vector<int> pureLiterals;
    std::map<int, int>::iterator it;
    std::map<int, int>::iterator itEnd;
    it = assignment.begin();
    itEnd = assignment.end();

    while (it != itEnd) {
        // Only choose unassigned vars
        if (it->second == 0) {
            if (isPureLiteral(cnf, it->first)) {
                foundPureLiteral = true;
                pureLiterals.push_back(it->first);
                cnf = propagate(cnf, it->first);
            }
        }
        it++;
    }

    // Update assignment map
    for(int i = 0; i < pureLiterals.size(); i++) {
        int cur = pureLiterals.at(i);
        if (cur > 0) {
            assignment[cur] = 1;
        }
        if (cur < 0) {
            assignment[-cur] = -1;
        }
    }

    return foundPureLiteral;
}
```

In BCP, the algorithm will continuously find unit clauses in the input CNF. After finding an unit clause, the algorithm will assign it with a proper value (true/false) and then call propagate to simplify the input CNF.

```cpp
bool BCP(std::vector<std::vector<int>> &cnf, std::map<int, int> &assignment) {
    bool foundUnitClause = false;

    while (true) {
        int unitClause = 0;

        for (int i = 0; i < cnf.size(); i++) {
            if (cnf.at(i).size() == 1) {
                unitClause = cnf.at(i).at(0);
            }
        }

        if (unitClause == 0) {
            return foundUnitClause;
        } else {
            foundUnitClause = true;
        }

        if (unitClause > 0) {
            assignment[unitClause] = 1;
        }

        if (unitClause < 0){
            assignment[-unitClause] = -1;
        }

        cnf = propagate(cnf, unitClause);
    }

}
```

Also, we have several tool functions like chooseVar. It will choose the variable appears in the CNF with highest frequency. Because we want to eliminate as more variables as we can. This technique will also improve the efficiency of the SAT solver.

```cpp
int chooseVar(const std::vector<std::vector<int>> &cnf, std::map<int, int> assignment) {
    int var = 0;
    int max = 0;
    std::map<int, int>::iterator it;
    std::map<int, int>::iterator itEnd;
    it = assignment.begin();
    itEnd = assignment.end();

    while (it != itEnd) {
        // Only choose unassigned vars
        if (it->second == 0) {
            int count = 0;
            for (int i = 0; i < cnf.size(); i++) {
                for (int j = 0; j < cnf.at(i).size(); j++) {
                    if(abs(cnf.at(i).at(j)) == it->first) {
                        count++;
                    }
                }
            }
            // Choose variables that appears most often in CNF
            if(count > max) {
                max = count;
                var = it->first;
            }
        }
        it++;
    }
    return var;
}
```

Another tool function would be isPureLiteral. It determines if a literal is a pure literal (appears only positively or only negatively) in a CNF.

```
bool isPureLiteral(const std::vector<std::vector<int>> &cnf, int lit) {
    bool positive = false;
    bool negative = false;
    for (int i = 0; i < cnf.size(); i++) {
        for (int j = 0; j < cnf.at(i).size(); j++) {
            int cur = cnf.at(i).at(j);
            if (lit == cur) {
                positive = true;
                if (negative) {
                    return false;
                }
            }
            if (lit == -cur) {
                negative = true;
                if (positive) {
                    return false;
                }
            }
        }
    }
    return false;
}
```