This solver is based on DPLL, in which BCP and PLP are used.

In BCP, we traverse the clauses to see whether there exists a unit clause. If so, we assign appropriate value to make this clause satisfied (*Fig. 1*). Then, we remove every clause containing the unit clause's literal and discard the complement of the unit clause's literal from every clause containing that complement (*Fig. 2*). Repeat these steps until there is no clause or no unit clause.

```
int single_var;
for (unsigned int i = 0; i < bcp_cnf.size(); i ++){
    if (bcp_cnf[i].size() == 1){
        single_var = bcp_cnf[i][0];
        if (single_var > 0)
            var_assignment[single_var] = true;
        else if (single_var < 0)
            var_assignment[-single_var] = false;
        break;
    }
}
```

Fig. 1

```
for (int i = 0; i < bcp_cnf.size(); i ++){
    for (int j = 0; j < bcp_cnf[i].size(); j ++){
        if (single_var==bcp_cnf[i][j]){
            bcp_cnf.erase(bcp_cnf.begin()+i);
            i--;
            break;
        }
        else if (single_var==-bcp_cnf[i][j]){
            if(bcp_cnf[i].size()==1){
                tmp_result = false;
                return bcp_cnf;
            }
            bcp_cnf[i].erase(bcp_cnf[i].begin()+j);
            j--;
        }
    }
}
```

Fig. 2

As for PLP, if a variable occurs only positively in the formula, we assign this variable true. If a variable occurs only negatively in the formula, we assign complement of this variable false (*Fig. 3*). Then we traverse the formula to remove every clause containing this variable (*Fig. 4*).

```
std::map<int, bool> pure_var;
for(std::map<int, bool>::iterator iter=all_var.begin(); iter!=all_var.end(); iter++)
{
    int tmp = iter->first;
    if (all_var.find(-tmp)==all_var.end()){
        pure_var[tmp] = true;
        if (pure_var[tmp] > 0){
            var_assignment[pure_var[tmp]] = true;
        }
        else var_assignment[-pure_var[tmp]] = false;
    }
}
```

Fig. 3

```
for (int i = 0; i < plp_cnf.size(); i ++){
    for (int j = 0; j < plp_cnf[i].size(); j ++){
        if (pure_var.find(plp_cnf[i][j])!=pure_var.end()){
            plp_cnf.erase(plp_cnf.begin()+i);
            i--;
            break;
        }
    }
}
return plp_cnf;
```

Fig. 4

If the length of formula returned by BCP function is 0, we directly return true. If the BCP detects unit conflict, we return false. If the length of formula returned by PLP function is 0, we directly return true. After using BCP and PLP, the input becomes simpler. Then, we choose a literal and assign a truth value to it, simplify the formula, and then recursively call the DPLL function to check if the simplified formula is satisfied (Fig. 5)

```
int var = choose_var(plp_cnf);
var_assignment[var]=true;
std::vector<std::vector<int>> tmp(plp_cnf);
for (int i = 0; i < plp_cnf.size(); i ++){
    std::vector<int>::iterator it = find(plp_cnf[i].begin(), plp_cnf[i].end(), var);
    std::vector<int>::iterator it2 = find(plp_cnf[i].begin(), plp_cnf[i].end(), -var);
    if (it != plp_cnf[i].end()){
        plp_cnf.erase(plp_cnf.begin()+i);
        i--;
    }
    else if (it2 != plp_cnf[i].end())
        plp_cnf[i].erase(it2);
}

if (dpll_rec(plp_cnf)){
    return true;
}
```

Fig. 5

If we cannot find a solution, we backtrack and assign this variable false, simplify the formula, and then recursively call the DPLL function to check if the simplified formula is satisfied (*Fig. 6*).

```
else{
    var_assignment[var] = false;
    for (int i = 0; i < tmp.size(); i ++){
        std::vector<int>::iterator it = find(tmp[i].begin(), tmp[i].end(), var);
        std::vector<int>::iterator it2 = find(tmp[i].begin(), tmp[i].end(), -var);
        if (it2 != tmp[i].end()){
            tmp.erase(tmp.begin()+i);
            i--;
        }
        else if (it != tmp[i].end())
            tmp[i].erase(it);
    }
    return dpll_rec(tmp);
}
```

Fig. 6