I use DPLL with BCP and PLP as the SAT solver in bonus.

1. DPLL

My DPLL is based on the pseudo-code from the class as following:

```
bool DPLL(CNF φ, AssignMap A)
{
  1. φ' = BCP(φ,A)
  2. φ'' = PLP(φ')
  3. if(φ'' = ⊤) then return SAT;
  4. else if(φ'' = ⊥) then return UNSAT;
  5. p = choose_var(φ'');
  6. if(DPLL(φ'',A[p ↦ ⊤])) then return SAT;
  7. else return (DPLL(φ'',A[p ↦ ⊥]));
}
```

Like line (1), first call BCP. In order to improve the performance of DPLL, I judge whether cnf is sat after BCP. If it is, there is no need to call the PLP.

```
std::vector<std::vector<int>> modifiedCnf = BCP(cnf, &: assignMap);
if (modifiedCnf.empty()) {
  return true;
} else if (fomularIsFalse(modifiedCnf)) {
  return false;
}
```

Like line (2)-(4), call PLP and then judge whether cnf is sat. If cnf is empty, then it is sat. If cnf has a clause like [0], then it is unsat.

```
modifiedCnf = PLP(numVar, cnf: modifiedCnf, &: assignMap);
if (modifiedCnf.empty()) {
  return true;
} else if (fomularIsFalse(modifiedCnf)) {
  return false;
}
```

Like line (5), choose a variable that is not assigned a value.

```
int p = chooseVar(numVar, assignMap);
```

Like line (6)-(7), Call the DPLL recursively with *p* is true and *p* is false.

```
std::map<int, bool> assignMapPTrue = assignMap;
assignMapPTrue.insert( v: { &: p,  u2: true});
std::map<int, bool> assignMapPFalse = assignMap;
assignMapPFalse.insert( v: { &: p,  u2: false});
if (DPLL(numVar,  cnf: modifiedCnf,  &: assignMapPTrue)) {
  return true;
} else {
  return DPLL(numVar,  cnf: modifiedCnf,  &: assignMapPFalse);
}
```

2. BCP

My BCP is based on the pseudo-code from Piazza as following:

BCP maintains a queue or map A of (variable, value) pairs. This is a list of unit clauses and decisions.

1: for each pair (v, u) in A
2: for each clause C in ϕ
3: if (v occurs positively in C and u is the value true) OR
v occurs negatively in C and u is the value false)) then
mark the clause C as satisfied and remove from ϕ
4: if (v occurs positively in C and u is the value false) OR
v occurs negatively and u is the value true)) then
mark the literal as false, and shrink the clause C by dropping v
5: if C becomes unit, add it to the map A with appropriate value and remove it from ϕ
6: Return the modified formula //Note that the formula ϕ is being modified by either dropping clauses (step 3 and step 5) or dropping literals (step 4)

First use a list to record the order of keys in *assignMap*. Because line (5) will add pair to *assignMap* and the map in C++ will be automatically sorted based on the key. So, the adding will affect the for loop for the pair in *assignMap*. I use a list to avoid that.

```
std::list<int> assignMapOrder;
for (mapIter = assignMap.rbegin(); mapIter != assignMap.rend(); ++mapIter) {
  assignMapOrder.push_back( x: mapIter->first);
}
```

Like line (1)-(2), for loop the pair in *assignMap*, the clause in cnf and the literal in clause.

```
for (orderIter = assignMapOrder.begin(); orderIter != assignMapOrder.end(); ++orderIter) {
  for (int i = 0; i < modifiedCnf.size(); i++) {
    for (int j = 0; j < modifiedCnf[i].size(); j++) {
```

Like line (3), if the clause has a literal that is true, remove the clause from cnf.

```
if ((*orderIter == modifiedCnf[i][j] && assignMap[*orderIter]) ||
    (*orderIter == 0 - modifiedCnf[i][j] && !assignMap[*orderIter])) {
  modifiedCnf.erase( position: modifiedCnf.begin() + i);
```

Like line (4), if the clause has a literal that is false, remove the literal from the clause. And if the literal is the only literal in the clause, make the clause be [0], which means the clause is false.

```
else if ((*orderIter == modifiedCnf[i][j] && !assignMap[*orderIter]) ||
         (*orderIter == 0 - modifiedCnf[i][j] && assignMap[*orderIter])) {
if (modifiedCnf[i].size() == 1) {
  modifiedCnf[i][j] = 0;
} else {
  modifiedCnf[i].erase( position: modifiedCnf[i].begin() + j);
  j--;
}
```

Like line (5), if the clause is unit, judge whether the literal is in *assignMap*. If it is in, like the red square, if the literal is false, then make the clause be [0]. Otherwise, remove the clause from cnf. If the literal is not in *assignMap*, like the blue square, if the literal > 0, add it into *assignMap* with true; if the literal < 0, add it into *assignMap* with false. And then remove the clause from cnf.

```
if (modifiedCnf[i].size() == 1 && modifiedCnf[i][0] != 0) {
  if (assignMap.count( k: modifiedCnf[i][0]) == 1) {
    if (!assignMap[modifiedCnf[i][0]]) {
      modifiedCnf[i][0] = 0;
    } else {
      modifiedCnf.erase( position: modifiedCnf.begin() + i);
      i--;
      break;
    }
  } else if (assignMap.count( k: 0 - modifiedCnf[i][0]) == 1) {
    if (assignMap[0 - modifiedCnf[i][0]]) {
      modifiedCnf[i][0] = 0;
    } else {
      modifiedCnf.erase( position: modifiedCnf.begin() + i);
      i--;
      break;
    }
  } else {
    if (modifiedCnf[i][0] > 0) {
      assignMap.insert( v: { &: modifiedCnf[i][0],  u2: true});
      assignMapOrder.push_back( x: modifiedCnf[i][0]);
    } else {
      assignMap.insert( v: { u1: 0 - modifiedCnf[i][0],  u2: false});
      assignMapOrder.push_back( x: 0 - modifiedCnf[i][0]);
    }
    modifiedCnf.erase( position: modifiedCnf.begin() + i);
    i--;
    break;
  }
}
```

3. PLP

My PLP is based on the pseudo-code from Piazza as following:

```
PLP ( Formula F , Assignment & A ) {
   L = [ ]
   for every variable v in the formula F {
      if ( v occurs only positively )
         A[v] = True
         L.append( v )
      else if ( v negatively in the entire formula F )
         A[v] = False
         L.append( v )
   }
   for each variable u in L  {
      for each clause C in F
         if ( u occurs in C )
            drop C from F
   }
}
```

First, find and use *reverseList* to store all the non-pure literal in cnf. Perform a for loop on cnf, when *literal* > 0, *pureLiteralCount*++. When *literal* < 0, *pureLiteralCount*++. If *literal* > 0 and *pureLiteralCount* < 0 or *literal* < 0 and *pureLiteralCount* > 0, the literal is not pure literal, add it to reverseList.

```
for (int i = 0; i < modifiedCnf.size(); i++) {
  for (int j = 0; j < modifiedCnf[i].size(); j++) {
    if (modifiedCnf[i][j] > 0 && pureLiteralCount[modifiedCnf[i][j]] >= 0) {
      pureLiteralCount[modifiedCnf[i][j]]++;
    } else if (modifiedCnf[i][j] < 0 && pureLiteralCount[0 - modifiedCnf[i][j]] <= 0) {
      pureLiteralCount[0 - modifiedCnf[i][j]]--;
    } else {
      reverseList.push_back( x: abs(modifiedCnf[i][j]));
    }
  }
}
```

Then, all the variables that is not in reverseList is pure literal, add it to list and *assignMap*.

```
for (int i = 1; i <= numVar; i++) {
  if (std::find( first: reverseList.begin(),  last: reverseList.end(),  value_: i) == reverseList.end() && pureLiteralCount[i] > 0) {
    list.push_back( x: i);
    assignMap.insert( v: { &: i,  u2: true});
  } else if (std::find( first: reverseList.begin(),  last: reverseList.end(),  value_: i) == reverseList.end() && pureLiteralCount[i] < 0) {
    list.push_back( x: 0 - i);
    assignMap.insert( v: { u1: 0 - i,  u2: false});
  }
}
```

Finally, for all the clauses that has pure literal, remove it from cnf.

```
std::list<int>::iterator iter;
for (iter = list.begin(); iter != list.end(); ++iter) {
  for (int i = 0; i < modifiedCnf.size(); i++) {
    for (int j = 0; j < modifiedCnf[i].size(); j++) {
      if (*iter == modifiedCnf[i][j]) {
        modifiedCnf.erase( position: modifiedCnf.begin() + i);
        i--;
        break;
      }
    }
  }
}
```