```
   Input: CNF_File file
   Output: Model or Unsat
 1 algorithm BigSat
 2     Partition() // 需要完成的部分
 3     while sat_files < all_files do
 4         Load() // 加载新文件，更新 Solver 中的一些数据结构
 5         while True do
 6             Propagate()
               // 和原算法的作用基本类似，但会新加载的文件后将 Clause 中存在的所有
               // Conflicts 返回
 7             if not conflict then
 8                 if all variables assigned then
 9                     sat_files ++
10                     break
11                 else
12                     Decide() // 选择一个新的 var 并对它进行赋值
13             else
14                 Analyze()
                   // 对 Propagate 中返回的 confl 或者 confls 进行分析，返回 learnt
                   // clause 并分析得到 backtrack 到的 Decision Level
                   // learnts 结构会添加多个，但是返回仅返回一个
15                 if top-level conflict found then
16                     return Unsatisfiable
17                 else
18                     Backtrack()
                       // 撤销所有做过的 decide 直到得到的 conflict clause 是 unit 的
                       // 该操作可以起到对一个变量值取逆的效果
19     return Model // current assign
```

图 1: OverView

任务 1. 完成 *Partition*(int nParts, double range)

Partition 的作用是将指定文件 Clause set 分割成多个 subset 并写入不同文件中，每个 subset 内的 Clause 互不相交，并且这些子集的并与原集合相同或是其等价 [1] 子集，此外每个 subset 需要满足所有 Clause 的 Lit 个数之和基本相同，在进行分割的同时需要对原集合的包含关系作出一定程度的分析，使得生成的子集中尽可能少的有 Cluase 之间的包含关系 [2][3]，此外，还可对原 Clause 集进行简化 [4]，更早的发现 Unsat 情况。

Partition 除了可以直接获取到全局的原 CNF 文件信息之外，需要接受两个参数 nParts 代表输入文件需要分割成几块，以及 range 代表分割时候允许的误差范围，这里指的是含有 Lit 数最大的集合和最小的集合之间的差占平均 Lit 数的比例。

[1] 这里的等价性指的是，如果 A 等价于 B，那么对于任何赋值序列 M，$\phi_A|_M = \phi_B|_M$。
其中 $\phi_X|_M \in \{1, 0\}$ 为 Clause 集合 X 所有 Clause 的并下在赋值序列 M 下取值。

[2] Clause 之间的包含关系指的是 Clause 的 Lit 集合之间的包含关系
假设某个 Clause C1 为 1,2,3，另一个 Clause C2 为 1,2,3,4，则 C1⊂ C2, 此时留下 C1
假设某个 Clause C1 为 1,-2,3，另一个 Clause C2 为 1,2,3,4, 则两者没有包含关系

[3] 这里尽可能少的最低限度指的是在同一个文件里面不应该出现有包含关系的两个 Clause

[4] 可以参考 MINISAT 的 $simplify()$ 方法

```
   Global Vars: Slover &S
   Output: SATISFIABLE or UNSATISFIABLE
1  function SEARCH()
2      Partition() // 【新】分割大文件为小文件
3      while true do
4          Vec <imGnode > κ = newPropagate()
5          if κ.size() == NULL then
               // 没有发现 CONFLICT
6              if decisionLevel() == 0 then
7                  simplifyDB()
8              if learnts_out.size() − nAssigns() ≥nof_learnts then
9                  reduceDB() // 把 reduce 掉的一些 learnt clauses 需要写回文件中
10             if assigns.size() = nVars() then
11                 if nSatFiles = nFiles then
                       // 所有的文件均已经满足，找到了 Model
12                     return SATISFIABLE
13                 else
14                     nSatFiles ++
15                     Load()// 【新】删除原本的 watches 结构，重设 propQ，并用新加载文件
                           的 clause 更新 watches 结构
16             else
17                 lit p= lit(order.select())// 挑选下一个 var 来进行赋值
18         else
               // 找到了 CONFLICT
19             Vec <Vec <lit > > learnt_clauses
20             int backtrack_level
21             if decisionLevel() == root_level then
22                 return UNSATISFIABLE
23             analyze(κ, learnt_clause, &backtrack_level)
24             cancelUntil(max(backtrack_level, root_level))
25             store(learnt_clauses)// 【新】将所有学习到的 clause 写进文件和开辟的内存空
                   间
26             decayActivities()
```

图 2: Search Algorithm

Global Vars: *Slover* &S

Result: 更新 S.imG，构造所有当前赋值产生的冲突节点，并且在算法的运行过程中维持 watches 结构性质正确

```
1  function newPropagate()
2      Vec <imGnode > κ
3      while PropQ.size()>0 do
4          lit p = propQ.dequeue()
5          Vec <Cref > tmp = watchesp[p]
6          for int i, j = 0; i < tmp.size(); i + + do
               // WL 需要满足假设 clause 没有被满足，那么两个 watcher 需要是 undef
               // minisat 中用位置标记 watcher，clause 中 0,1 位置为 watcher
7              if tmp[i][0] == ¬p then
8                  tmp[i][0] = tmp[i][1], lits[1] = ¬p
                   // 已知 watcher[1] 为 false
9                  if assigns.[var(tmp[i][0])] == True then
10                     j++// 不需要对这里进行修改，即使发生了回溯也不会破坏 WL 性质
11                     continue
                   // 寻找下一个 watcher
12                 int max = -1
13                 for int k = 2; k < tmp[i].size(); k++ do
14                     if tmp[i][k] ! = False then
15                         tmp[i][k] ↔ tmp[i][1]
16                         watches[tmp[i][1]].push(tmp[i])
17                         goto NextClause// 这里 j 没有增加，所以这里的 tmp[i] 的将会删除
18                     if imG[var(tmp[i][k])].decisionLevel() >max then
19                         max = imG[var(tmp[i][k])].decisionLevel()
20                 j++// 没有找到 watcher，所以是 unit clause 或者是 conflict,watches 结构不用发生变化
21                 if value(tmp[i][0])== False then
                       // 【新】发现了一个 conflict，为该 confl 构造冲突节点
22                     if imG[var(tmp[i][0]).decisionLevel() >max] then
23                         max = var(tmp[i][0]).decisionLevel()
24                     imGnode & node = new imGnode (max)
25                     foreach lit p in temp[i] do
26                         imG[node.getindex()].push(imG[var(p)])// 孩子指向父母
27                     conflict_vars.push(var(p)), κ.push(node)
28                     j++
29                     continue
30                 else
31                     newUncheckedEnqueue(tmp[i][0], tmp[i])// 【新】这里更新 imG
32             NextClause:// 扫描下一个 clause
33     return κ
```

图 3: newPropagate Algorithm

```
Global Vars: Slover &S
Input: 需要做 BCP 的 lit p、使得它发生 BCP 的 Cref from
Result: 更新 implication Graph 结构:S.imG
1  function newUncheckedEnqueue(lit p, Cref from)
2      assigns[var(p)] = lbool(!sign(p))
3      int max = -1
4      foreach lit a in from do
           // 确定 p 的 decision level
5          if a != p && (&imG[var(a)]).decisionLevel()>max then
6              max = imG[var(a)].decisionLevel()
7      imG[var(p)].setDL (max)// 更新 p 的 decision level
8      foreach lit a in from do
           // 更新 imG
9          if a != p then
10             imG[var(p)].push(& imG[var(a)])// 反向的图，更快找 1-UIP
11     trail.push(p)
```

图 4: newUncheckedEnqueue Algorithm

```
Global Vars: Slover &S
Input: 所有发现的冲突节点 κ
Result: 得到回溯的 level &β_L 以及学习到的 clauses out_learnts
1  function newAnalyze(Vec < imGnode > κ, Vec < Cref > out_learnts, int&β_L)
2      int counter = 0
3      Vec <lit > p_resason
4      β_L = +∞
5      for int i = 0; i< κ.size(); i++ do
6          Vec <bool > seen(nVars(), false)
7          out_learnts.push(new Vec <Lit>())
8          out_learnts[i].push()
9          int p = -1
10         int index = κ[i].getindex()
11         int pathC = 0
           // imG 上如何标识该边所组成的 clause 为 learnt, 需要重新设计
           // 下面利用 BFS 在 imG 上寻找 1-UIP 得到 learnt_clause
12         Queue BFS
13         BFS.push(imG[index].getReason())
14         BFS.push(imG[conflict_vars[i]].getReason())
15         repeat
16             int q = BFS.pop().getindex()
17             varBumpActivity(q)
18             if imG[q].decisionLevel() >= imG[inde].decisionLevel() && ! seen[q] then
19                 p = q // maybe UIP pathC++
20             else
21                 out_learnts[i].push(mkLit(q, !sign(assigns[q])))
22             BFS.push(imG[q].getReason()) pathC–
23         until pathC≤0;
24         out_learnts[i][0] = p]
           // smplify conflict clause 应该可以保持原样?
25         if out_learnt.size() == 1 then
26             β_L = 0
27         else
28             int max_i = 1
29             for int lc_index = 2; lc_index<out_learnts[i].size(); lc_index++ do
30                 int l_1 =imG[out_learnts[i][lc_index]].decisionLevel()
31                 int l_2 =imG[out_learnts[i][max_i]].decisionLevel()
32                 if l_1 >l_2 then
33                     max_i = lc_index
34             out_learnts[i][1]↔ out_learnts[i][max_i] β_L = imG[var(out_learnts[i][1])]
35         seen.clear()
```

图 5: newAnalyze Algorithm

Soundness

Algorithm 中产生的解是所求 CNF 的一个解。

Proof.

在 *Algorithm* 最外层的循环过程中，所有的行为都只会增加 clause，或者删除增加的 clause。假设 $Partition()$ 后的 clause 集合为 Φ_i，循环结束后 clause 集合为 Φ_f，由于循环过程中 clauses 集合变化的性质，则我们有 $\Phi_i \subseteq \Phi_f$，所以如果我们有在返回结果 assignment A 使得 $\Phi_f|_A = 1$，则必有 $\Phi_i|A = 1$，是 $Partiton()$ 后所有 clause 的一个解。

在 $Partition()$ 过程中，$\forall c_d \in \{c \mid c \text{ is deleted in } \Phi_i\}$，那么必然 $\exists c^* \in \{c \mid \text{remains in } \Phi_i\}$，其中 c^* 的文字是 c_d 的文字的自己，所以有 $c^*|_A = 1 \rightarrow c_d|_A = 1$，所以 A 必然是原文件所有 clauses 的一个解。

Completeness

证明大纲：

由于 $\mathbf{A}(x)$ 以及 $\alpha(x)$ 的定义，我们可以得到：

Lemma 2.1. 任何时时刻，由算法 Algorithm 生成的 imG 实际上计算的是 Φ 的一个子集的真实 imG

待说明

由 GRASP 算法可得，假设算法回溯的 decision level 为 β_L，那么算法不可能在回溯到 (β_L, c) 之间找到解 $\rightarrow$ 则可以知道在发生冲突后，不可能在回溯到 $(\beta_L, min(\kappa\_level))$ 之间找到解,(这里需要说清楚 current decision level c 和 $min(\kappa\_level)$ 之间的关系)。同时由 Lemma 2.1 可知，在回溯到 $(\beta_L, min(\kappa\_level))$ 之间不存在任何一个与当前 imG 相对应的 Φ' 的解，也就不会存在 Φ 上的解，所以可得算法是 complete 的。