

```

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()
7       // 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
8       // Conflicts 返回
9       if not conflict then
10        if all variables assigned then
11          sat_files ++
12          break
13        else
14          Decide() // 选择一个新的 var 并对它进行赋值
15        else
16          Analyze()
17          // 对 Propagate 中返回的 confl 或者 concls 进行分析，返回 learnt
18          // clause 并分析得到 backtrack 到的 Decision Level
19          // learnts 结构会添加多个 clause
20          if top-level conflict found then
21            return UNSATISFIABLE
22          else
23            Backtrack()
24            sat_files = 0 // 撤销所有做过的 decide 直到得到的 conflict clause 是
25            unit 的
26            // 该操作可以起到对一个变量值取逆的效果
27
28 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<sup>[1]</sup>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 Clause 之间的包含关系<sup>[2][3]</sup>，此外，还可对原 Clause 集进行简化<sup>[4]</sup>，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  $C1 \subset C2$ ，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

```

Input: CNF_File file
Global Vars: PartResults, CNF-File
Output: MODEL or UNSAT

1 algorithm P-BIGSAT
2   Partition()
   // 将 CNF-File 分成多个 Partitions
3   forall thread i do
4     Load()
     // thread i 加载 Partitions[i]
     // thread i 加载 PartResults[i] 中的 assign 结果以及某种方式保存的
     // implicatin graph
     // thread i 加载 PartResults[i] 中的已满足 Partition 的数量 nSPartition
5     while TRUE do
6       Propagate()
7       if not conflict then
8         if all variables assigned then
9           nSPartition ++
10          if nSPartition == sizeof(Partitions) then
11            return MODEL and kill other thread
            // 将当前的 assign 作为 MODEL 输出, 并且结束其余线程
12          else
13            (assign, nSPartitions) -> Partitions[i+1]
            // 将当前的 assign 和 implication graph 作为自己的 PartResult
            // 进行输出, 并累加当前的 assign 已满足的 NSPartition 数量
14          else
15            Decide() // 选择一个新的 var 并对它进行赋值
16        else
17          Analyze()
18          if top-level conflict found then
19            return UNSATISFIABLE and kill other thread
20          else
21            Backtrack()
22            nSPartition = 0
            // 将 nSPartition 设置成 0 重新来计算

```

图 2: Parallel Overview

**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     CRef cr = Propagate()// Vec <CRef > crs=newPropagate() 基本实现
5     if cr == CRef_Undef then
6       // 如果是 newPropagate() 则改成 crs.size()==0
7       if decisionLevel() == 0 then
8         simplifyDB()
9       if learnts_out.size() - nAssigns() ≥ nof_learnts then
10        reduceDB()
11        // 把 reduce 掉的一些 learnt clauses 需要写回文件中
12        // 这里的 reduceDB() 沿用了 MINSAT 的做法是否会对性能或者结果造成影
13        // 响还需要进一步的大规模实验
14      if assigns.size() = nVars() then
15        if nSatFiles = nFiles then
16          // 所有的文件均已满足，找到了 Model
17          return SATISFIABLE
18        else
19          nSatFiles ++
20          Load()
21          // [新] 删除原本的 watches 结构，删除不在 vardata 中的 clauses，
22          // 重设 propQ，并用新加载文件的 clause 更新 watches 结构，这里的
23          // 更新有两种做法一个是删除全部上次 watches 结构，另一种是保留在
24          // vardata 中的出现的 clauses
25      else
26        lit p = lit(order.select())// 挑选下一个 var 来进行赋值
27    else
28      Vec <lit > learnt_clause
29      // 产生多个 learnt 需要初始化 Vec <Vec <lit > > learnt_clauses
30      int backtrack_level
31      if decisionLevel() == root_level then
32        return UNSATISFIABLE
33      analyze(cr, learnt_clause, &backtrack_level)
34      // 目前沿用了 MINSAT 的 analyze(), 每产生冲突就返回
35      // newAnalyze(crs, learnt_clauses, &backtrack_level) 正在 bug 调试中
36      cancelUntil(backtrack_level, root_level)
37      store(learnt_clauses)
38      // 【新】将所有学习到的 clause 写进文件和开辟的内存空间，这里还需要具体
39      // 实验确定参数
40      decayActivities()

```

**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, 这里会产生‘多余的’*Conflict clause*, 需要通过 *newAnalyze()* 的 *seen* 结构避免重复学习, 目前已经基本实现

```

1 function newPROPAGATE()
2   Vec <CRef > crs
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
7       // WL 需要满足假设 clause 没有被满足, 那么两个 watcher 需要是 undef
8       // minisat 中用位置标记 watcher, clause 中 0,1 位置为 watcher
9       if tmp[i][0] == ¬p then
10        tmp[i][0] = tmp[i][1], lits[1] = ¬p
11        // 已知 watcher[1] 为 false
12        if assigns[var(tmp[i][0])] == TRUE then
13          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
14          continue
15        // 寻找下一个 watcher
16        for int k = 2; k < tmp[i].size(); k++ do
17          if tmp[i][k] != FALSE then
18            tmp[i][k] ↔ tmp[i][1]
19            watches[tmp[i][1]].push(tmp[i])
20            goto NextClause// 这里 j 没有增加, 所以这里的 tmp[i] 的将会删除
21        j++// 没有找到 watcher, 所以是 unit clause 或者是 conflict, watches
22        结构不用发生变化
23        if value(tmp[i][0]) == FALSE then
24          // 【新】发现了一个 conflict
25          crs.push(tmp) j++
26          continue
27        else
28          newUncheckedEnqueue(tmp[i][0], tmp)// unit propagation
29      NextClause:// 扫描下一个 clause
30   return crs

```

图 4: 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 *vardata* 结构

```

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
5     // 确定 p 的 decision level
6     if a != p && (&imG[var(a)].decisionLevel()) > max then
7       max = imG[var(a)].decisionLevel()
8   vardata[var(p)].reason = from
9   vardata[var(p)].level = max
10  trails.push(p, max) // 将 p 放到准确的 decision level 的 trails 路径上, 确保能够
    做正确的 BSF 这里还需要重新设计

```

图 5: newUncheckedEnqueue Algorithm

**Global Vars:** *Slover* & *S*

**Input:** 所有发现的冲突节点 **Vec** <**CRef**> *crs*

**Result:** 得到回溯的 level &  $\beta_L$  以及学习到的 clauses *out\_learnts*

```

1 function newANALYZE(Vec < CRef > crs, Vec < CRef > out_learnts, int&  $\beta_L$ )
2    $\beta_L = +\infty$ 
3   for int i = 0; i < crs.size(); i++ do
4     Vec <bool> seen(nVars(), false)
5     out_learnts.push(new Vec <Lit>())
6     out_learnts[i].push()
7     int pathC = 0
8     int max = maxLevel(crs[i])
9     int index = trails.size() repeat
10      for int j = 0; j < crs[i].size(); j++ do
11        if level(var(crs[i][j]))  $\geq$  max && seen[var(crs[i][j])] != 1 then
12          pathC ++ seen[var(crs[i][j])] != 1
13          // 这里应用 seen 和 trails 结构来对 vardata 构建的 implication
14          // graph 进行 BFS
15          // 这里的逻辑需要协同 newUncheckedEnqueue() 完成, debug 中...
16        else
17          seen[var(crs[i][j])] != 1
18          out_learnts[i].push(crs[i][j])
19        while seen[var(trail[index-])] do
20          nothing
21      crs[i] = reason(var(trails[index + 1])) // BFS 下一个节点为 trails[index + 1]
22      pathC -
23    until pathC  $\leq$  0;
24    out_learnts[i][0] =  $\neg$ p
25    redundant() // simplify conflict clause 应该可以保持原样?
26    if out_learnt.size() == 1 then
27       $\beta_L = 0$ 
28    else
29      // 保证 out_learnts[0] 中的 decision level 最大的 Lit 是所有
30      // out_learnts 中最小的
31      if max > maxLevel(out_learnts[i]) then
32        out_learnts[0]  $\leftrightarrow$  out_learnts[i]
33     $\beta_L = \max$ 
34  seen.clear() // 重设 seen 结构保证下一次的 BFS 正确

```

图 6: 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  $\Phi_i$ ，循环结束后 clause 集合为  $\Phi_f$ ，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  $\exists c^* \in \{c \mid c \text{ remains in } \Phi_i\}$ ，其中  $c^*$  的文字是  $c_d$  的文字的自己的自己，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.** 任何时刻，由算法 *Algorithm* 生成的 imG 实际上计算的是  $\Phi$  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  $\beta_L$ ，那么算法不可能在回溯到  $(\beta_L, c)$  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  $\Phi'$  的解，也就不会存在  $\Phi$  上的解，所以可得算法是 complete 的。