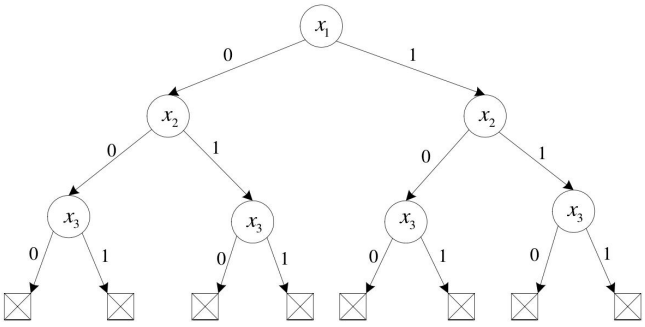
介绍 DPLL 的主要是 The Quest for Efficient Boolean Satisfiability Solvers 这篇文章，这是一篇文献综述，介绍了 DPLL 目前主流的算法框架以及各个组成部分的具体实现。

在介绍这篇文章之前，我们先来讲讲什么是 DPLL？它是解决 SAT 问题的一种算法。那什么是 SAT 问题？它是 Boolean Satisfiability Problem 的缩写：对于一个 CNF（合取范式），是否存在存在一组 variable assignment（真值指派），使得这个 CNF 的所有子句都可满足（所有子句都取值为真），如果存在的话就是 SAT（可满足），不存在的话就是 UNSAT（不可满足）。

具体来说，DPLL 的核心思想就是依次对 CNF 实例的每个变量进行赋值，其搜索空间可以用一个二叉树来表示，树中的每个节点对应一个变量，取值只能为 0 或 1，左右子树分别表示变量取 0 或 1 的情况，从二叉树中根节点到叶子节点的一条路径就表示 CNF 实例中的一组变量赋值序列，DPLL 算法就是对这棵二叉树从根节点开始进行 DFS( 深度优先搜索 ) 遍历所有的通路，以找到使问题可满足的解。

下图是一个 DPLL 搜索空间的示例图：




但是单纯的深度优先搜索的时间复杂度一定是指数级别的，不可能用来求解大规模的问题，所以人们对原始 DPLL 算法进行了很多改进，这篇文章主要就是介绍这些改进的地方。

接下来我们看一下这篇文章的各个章节的内容，第 1、4、5 节分别是引言、其他技术和总结，这些不重要可以不看，关键点在第 2、3 节，包括以下内容：

2：主要是 **Fig2 与 Fig1** 新旧两个架构的介绍

3.1：变量决策策略的介绍

3.2：启发式探索算法

3.2.1 :BCP( 布尔约束传播 )的介绍 ,包括 **counters-based、head/tail list、2-literal** 三种不同的 BCP 算法

3.2.2：其他的探索算法，不重要，可以不看

3.3：冲突分析与学习的介绍，主要讲了按时序回溯与**不按时序回溯**的区别，以及 **rel_sat** 与 FirstUIP 两种冲突分析的方法

3.4：数据存储方式的介绍，包括早期的链表储存，**Chaff 采用的数列方式**以及 Tire 三叉树储存方式

3.5：预处理、随机重启的介绍

==========第 2 节==========

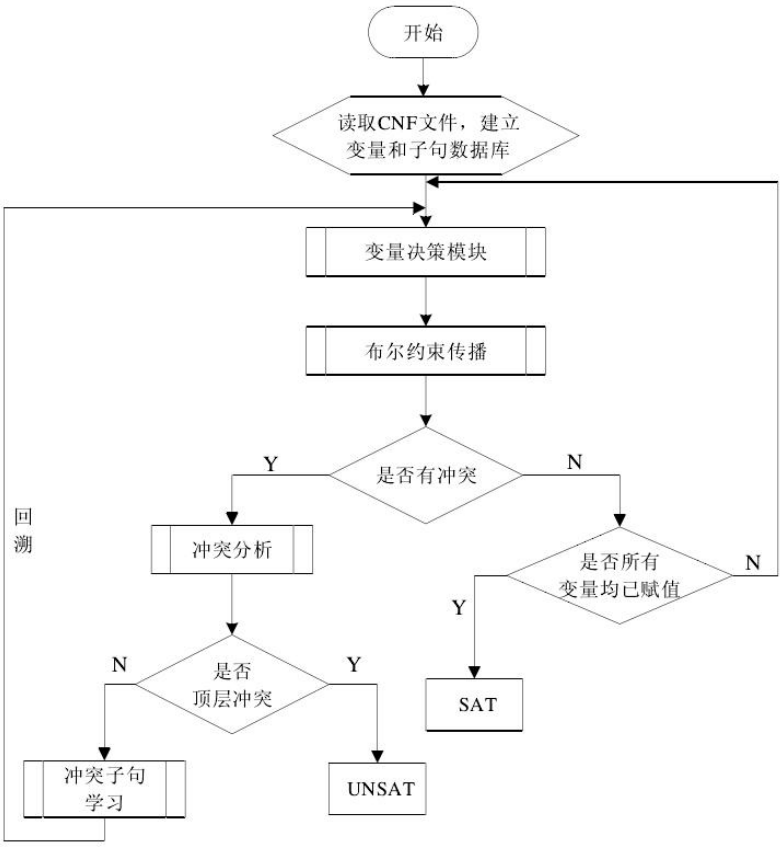
这里介绍第 2 节的内容，这一节主要从宏观上介绍了目前 DPLL 的总体架构，也就是 Fig2 的内容，如下所示：

```
status = preprocess(); //预操作，对应 3.5 节内容
if (status!=UNKNOWN) return status;
 while(1) {
    decide_next_branch();  //变量决策环节，对应 3.1 节内容
    while (true) {
       status = deduce();  //推理环节（BCP），对应 3.2 节内容
       if (status == CONFLICT) {
         blevel = analyze_conflict();  //冲突分析，对应 3.3 节内容
         if (blevel == 0)
         return UNSATISFIABLE;
         else backtrack(blevel); //智能回溯，对应 3.3 节内容
       }
       else if (status == SATISFIABLE)
       return SATISFIABLE;
```

```
            else break;
        }
}
```

简单分析一下 Fig2 的框架，首先执行的是 preprocess()这个预处理操作，其实就是对 CNF 实例进行各种化简减轻后续的求解工作量，预处理操作会在 3.5 节中介绍。如果预处理不能直接得出结果的话，就进行后面的 decide_next_branch()操作，这就是变量决策操作，它会分析从哪个变量开始赋值是最合适的，具体内容会在 3.1 节介绍。对变量赋值以后，会执行一个 deduce()操作，可以管它叫推理操作，目的是识别该赋值所导致的必要赋值，这些会在 3.2 节中具体介绍。当然不正确的赋值可能会产生错误，这就会产生冲突，我们需要 analyze_conflict()分析这个冲突得到冲突发生的根源位置，然后通过 backtrack()回溯到这个位置，为这个变量赋另外一个值，继续往下搜索，如此循环，直到找到满足 SAT 的一组真值指派。如果回溯到了最顶层还没有解决问题的话，那就表示这个 CNF 实例是不可满足的。

以下为整个 Fig2 的流程图：



除了目前主流的 Fig2 中介绍的框架之外，本文还介绍了一个老版本的框架，也就是 Fig1 中描述的框架，不是很重要我就不贴代码了，但我们需要知道老版本的框架是基于递归的，新版本是基于迭代的，递归速度慢，且容易发生溢出，相对迭代就有很多自身的劣势，另外，本文还提到新版本的另一个优势，*"The algorithm described in Fig. 2 is an improvement of algorithm in Fig. 1 as it allows the solver to backtrack non-chronologically"*，也就是非时间顺序回溯（智能回溯）的优势，这在 3.3 节中会具体介绍。

========== 第 3 节 ==========

这里介绍第 3 节的内容，这是本文重点章节，具体介绍了 Fig2 框架中的各个函数的具体实现方式，并比较了各种不同的实现方式各自的优劣所在。3.1 节讲变量决策的方式，3.2 节主要讲推理操作的方式，核心是 BCP（布尔约束传播），3.3 是冲突分析和学习，顺带讲了智能回溯，3.4 是数据存储方式，3.5 是预处理和随机重启。下面对每个章节进行介绍。

========== 3.1 ==========

3.1 节主要介绍了变量决策的几种方法，也就是 Fig2 中 decide_next_branch()函数的实现方法。我们知道这样一个事实，先对哪个变量进行赋值，会直接影响二叉搜索树的规模大小，也就会影响求解时间，所以如何快速有效地决策出下一个将要赋值的变量，这很重要。比如，某个 CNF 实例有 x1、x2、x3 三个变量，那么我们先从哪个变量开始赋值呢？文章 3.1 节就是在分析这个问题。

3.1 节一共介绍了 4 种变量决策方式，分别是 MOM（JW）方法、Literal Count Heuristics 方法、VSIDS 方法以及一个改进的方法，论文中的名字是 A Fast and Robust SAT Solver。下面我一一介绍一下这 4 种方法。

1.**MOM 法或者 JW 法**，MOM 的意思是 Maximum Occurrences on Minimum sized，也就是在最短字句中出现频率最高的变量优先，它基于这样的思路：在一个子句中，只要一个文字得到满足，那么这个子句就得到满足了。所以，如果一个子句的长度越长（含有字母数越多）,那么可以使得该子句满足的文字数目也就越多,这个子句也就越容易满足，所以它就不是所谓的“矛盾的主要方面”，我们不需要过于关注这个子句；然而，如果一个子句长度很小，那它就很不容易被满足，所以我们要优先考虑它们，给予它们更高的权重，这样做的目的就是让那些不容易被满足的子句优先得到满足。

具体的方法是求解出和变量 l 相对应的 J 值，哪个变量 J 值大就选哪个先赋值，变量对应的 J 值的计算公式如下：

$$J(l)=\sum_{l\in Ci}2^{-n_i}(i=1,2,3,.....m)$$

其中 l 表示变量，Ci 表示包含变量 l 的子句，共有 m 个子句，$n_i$ 表示这个子句的长度。它体现出了 MOM 算法的两个关键点：“最短的子句”和“出现频率最高的变量”，”最短子句”体现在长度越短（n 越小），$2^{-n_i}$ 的值就越大，它能给 J 值的贡献就越多；”出现频率最高的变量”体现在 l 出现次数越多的话，相加的项数就越多，J 值就更容易变的很大。

2.**Literal Count Heuristics**，它主要是统计已经给某些字母赋值但仍然没有得到满足的子句的数量，这个数量依赖于变量赋值，所以每次换变量的时候，所有自由变量都需要重新统计这个数量，效率较低。

3.**VSIDS**，Variable State Independent Decaying Sum，直译过来就是独立变量状态衰减和策略，它的具体操作步骤如下：

（1）为每一个变量设定一个 score，这个 score 的初始值就是该变量在所有子句集中出现的次数。

（2） 每当一个包含该字母的冲突子句被添加进数据库，该字母的 score 就会加 1（冲

突学习机制会在后面的 3.3 介绍）

（3）哪个变量的 socre 值最大，就从这个变量开始赋值

另外，为了防止一些变量长时间得不到赋值，经过一定时间的决策后，每一个变量的 score 值都会被 decay，具体方式是把 score 值除以一个常数（通常为 2-4 左右）。

*4.*文章里还介绍一种名为 **A Fast and Robust SAT Solver** 的方法，文中只说它和 VSIDS 非常类似，最大的区别是*"the decision heuristic will limit the decision variable to be among the literals that occur in the last added clause that is unresolved"*。

========== 3.2 ==========

3.2 节主要介绍 Fig2 中的 `deduce()` 函数的实现方法，这个阶段我们可以称之为"推理"，当一个变量被赋值后，可以通过推理来减少一些不必要的搜索，加快效率。推理过程主要依赖于 Unit clause rule（单元子句规则），所谓单元子句就是：在这个子句中，除了一个文字未赋值外，其他所有的文字都被赋值并体现为假，这样的子句就是 Unit clause（单元子句），剩下的这个文字就是 unit literal（单元文字）。很容知道，在单元子句中，这最后一个文字必须体现为真，整个子句才能被满足，把所有的单元文字都赋值并体现为真的过程就是 BCP（布尔约束传播）。

**BCP 的目标就是识别出单元子句并对单元文字赋值，能够减少搜索空间或提前逼出冲突。**

这里介绍了 3 种 BCP 的实现方法，分别是 counters 方法，head/tail list 方法，2-literal watching 方法，下面一一介绍一下。

1·**counters** 方法，具体做法是是为每个变量设置两个 lists，分别用来保存所有包含这个变量的正负字母的子句，并且每个子句都设置两个计数器，分别统计体现为真的字母数和体现为假的字母数。如果体现为假的字母等于总字母数，那就产生了冲突；如果体现为假的字母数比总字母数少 1，那就出现了单元子句，就需要对单元文字进行自动赋值。

这种方法效率并不高，如果一个 CNF 实例有 m 个子句， n 个变量，每个子句平均有 l 个文字，则有一个变量被赋值的时候，会有平均 ml/n 个计数器需要更新，在回溯的时候，每取消一个变量赋值，也会平均有 ml/n 个计数器的更新，所以当 l 很大的时候，这种方式效率并不高。

举一个例子：

1:$(\neg m \vee n \vee p)$
2:$(m \vee p \vee q)$
3:$(m \vee p \vee \neg q)$
4:$(m \vee \neg p \vee q)$
5:$(m \vee \neg p \vee \neg q)$
6:$(\neg n \vee \neg p \vee q)$
7:$(\neg m \vee n \vee \neg p)$
8:$(\neg m \vee \neg n \vee p)$

初始的时候，1-8 号子句各有两个计数器（分别记录赋值为 0 和 1 的文字数量），一开始所有计数器的值都是 0。变量 m 有链表分别用来保存所有包含 $m$ 和 $\neg m$ 的子句，包含 $m$ 的链表中有子句 2，3，4，5，而包含 $\neg m$ 的链表中有子句 1，7，8，其余变量也均有这样的两个链表。当给 m 赋 0 时，包含 $m$ 的链表中的子句 2，3，4，5 的 0 计数器就会更新，数量加 1，包含 $\neg m$ 的链表中的子句 1，7，8 的 1 计数器也会更新，数量加 1；如果再给 p 赋 0 的话，包含 $p$ 的链表中的子句 1，2，3，8 的 0 计数器就也会更新，数量加 1，此时 2，3 两个子句的 0 计数器数量变为 2，比总文字数 3 少 1，均成为了单元子句，2 可以推出 q 必须赋 1，而 3 可以推出 q 必须赋 0，产生冲突，BCP 完成任务。

2.**head/tail list** 方法，它为每个子句设置两个引用指针，分别是头指针和尾指针，初始时，头指针指向子句第一个文字，尾指针指向最后一个文字，每个子句的文字存放在一个数组中。对于一个变量 v，设置有四个链表，分别装有句头是 v 的子句，句头是非 v 子句，句尾是 v 的子句，句尾是非 v 的子句，分别标记为 `clause_of_pos_head(v)`，`clause_of_pos_tail(v)`，`clause_of_neg_head(v)`，`clause_of_neg_tail(v)`。**假设有 m 个子句的话，所有变量的四个链表中就共存放着 2m 个子句，无论后面这些子句怎么调换位置，子句总数是不变的。**

假设某变量 v 被赋值为 1，那么所有句头和句尾为 v 的子句就都可以忽略了，因为他们已经被满足了。而对于句头或句尾是非 v 的子句，就需要移动头尾指针寻找下一个未被赋值的字母，头指针往后移，尾指针往前移，移动时可能会发生以下四种情况：

（1）第一个遇到的文字 l 已经体现为真了，那么子句就已经满足了，什么都不用做，忽略这个子句就行了。

（2）第一个遇到的文字 l 是未赋值的，且不是句尾，那么就把这个子句 c 从 v 的链表中移除，放入到 l 的对应的链表中。

（3）如果头尾之间只剩一个变量未赋值，其他文字都体现为假了，就出现了单元子句，直接推断出单元文字的取值。

（4）如果头尾之间所有文字都体现为假，那产生冲突，需要回溯。

当一个变量被赋值的时候，平均有 m/n 个子句需要更新（m 为子句数，n 为变量数）。

还是这个例子：

1:$(\neg m \vee n \vee p)$
2:$(m \vee p \vee q)$
3:$(m \vee p \vee \neg q)$
4:$(m \vee \neg p \vee q)$
5:$(m \vee \neg p \vee \neg q)$
6:$(\neg n \vee \neg p \vee q)$
7:$(\neg m \vee n \vee \neg p)$
8:$(\neg m \vee \neg n \vee p)$

初始时，我们可以将各个子句填入变量的链表中，如下表所示：

| | clause_of_pos_head() | clause_of_neg_head() | clause_of_pos_tail() | clause_of_neg_tail() |
|---|---|---|---|---|
| m | 2，3，4，5 | 1，7，8 | | |
| n | | 6 | | |
| p | | | 1，8 | 7 |
| q | | | 2，4，6 | 3，5 |

**前两列可以拼成一个完整的 1-8 子句集，后两列也可以拼成一个 1-8 完整的子句集。**

此时，头尾指针分别指向每个子句的头部和尾部。

当我们给 m 赋值为 0 的时候，m 的 `clause_of_neg_head(m)` 链表中的子句 1，7，8 就可以忽略不看了，因为已经被满足，把它们标注为绿色，而 `clause_of_pos_head(m)` 中的子句 2，3，4，5 还没有被满足，这些子句头指针需要后移一位，转移后这些子句的头指针指向的文字就不是 m 了，所以需要将表格中 2，3，4，5 的位置换一下，更换位置的子句用红色标注，这次赋值后结果如下：

| | clause_of_pos_head() | clause_of_neg_head() | clause_of_pos_tail() | clause_of_neg_tail() |
|---|---|---|---|---|
| m | | 1，7，8 | | |
| n | | 6 | | |
| p | 2，3 | 4，5 | 1，8 | 7 |
| q | | | 2，4，6 | 3，5 |

以此类推，当我们再给 p 赋 0 的时候，4，5 就可以不用考虑了，2，3 的头指针需再后移一位，2，3 在表格中的位置也需要更换，如下表所示：

| | clause_of_pos_head() | clause_of_neg_head() | clause_of_pos_tail() | clause_of_neg_tail() |
|---|---|---|---|---|
| m | | 1，7，8 | | |
| n | | 6 | | |
| p | | 4，5 | 1，8 | 7 |
| q | 2 | 3 | 2，4，6 | 3，5 |

此时已经满足条件（3）了，形成了两个单元子句，2 可以推出 q 必须赋 1，而 3 可以推出 q 必须赋 0，产生冲突，BCP 完成任务。

3.**2-literal watching** 方法，这个方法与 H/T 类似，也要为每个子句关联两个指针，与 H/T 不同之处是，这两个指针没有先后次序，也就没有所谓的头和尾的概念，这样设置会带来很多好处，比如初始时这两个指针的位置可以是任意的（W/T 必须放在头和尾的位置），移动时也可以向前后两个方向移动（W/T 中头指针只能向后移，尾指针只能向前移），回溯时无需改动指针的位置（W/T 回溯时需要把指针变回原来的位置）。但这种设置也有弊端，即只有遍历完所有子句的文字后，才能识别出单元子句。想对应的，每个变量 v 也设置了两个 list 来分别存放以 watching 指针分为 v 以及非 v 的子句。接下来的操作都与 W/T 类似，当某个变量 v 赋值为 1 的话，watching 指针为 v 的子句可以忽略，watching 指针为非 v 的子句开始移动指针。

2-literal watching 与 H/T 的流程几乎是相同的，这里就不再把流程写一遍了，可以参考 H/T 中给的例子的流程。

========== 3.3 ==========

3.3 节主要介绍 Fig2 中的 `analyze_conflict()`函数的实现方法，这个阶段我们可以称之为“冲突分析和学习”，**它的目的是找到冲突产生的原因，这就是分析的过程；并告诉 SAT 处理器这些搜索空间是会产生冲突的，以后不要再踩这些坑了，这就是学习的过程。**

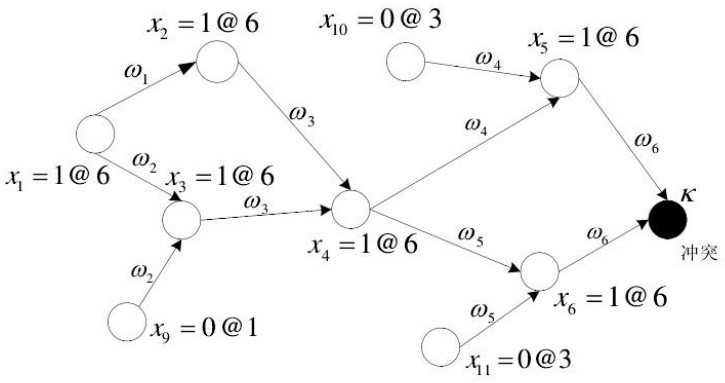
冲突与回溯是放在一起介绍的，因为产生了冲突，就需要回溯来解决，低级的冲突处理方式对应低级的回溯方式，智能的冲突处理方式就对应智能的回溯方式。

早期解决冲突的方法就是回到上一层，将变量取值翻转，继续往下进行搜索，这也叫时序回溯。但是这样做的结果很可能是冲突依旧存在，因为上一层的赋值也许并不是冲突产生的根本原因，从而白白浪费一次计算的时间，效率非常低。

目前主流的 SAT 处理器都采用基于冲突分析和学习的非时序回溯算法，它可以智能地分析出冲突产生的根本原因，并回跳多个决策层，并把会导致冲突的子句加入到子句集中。

以下为一次冲突分析和学习的例子：

$$\{x_9=0@1, x_{10}=0@3, x_{11}=0@3, x_{12}=1@2, x_{13}=1@2,\ldots\}$$
$$\omega_1=(\neg x_1\vee x_2);\qquad \omega_2=(\neg x_1\vee x_3\vee x_9)$$
$$\omega_3=(\neg x_2\vee\neg x_3\vee x_4);\quad \omega_4=(\neg x_4\vee x_5\vee x_{10})$$
$$\omega_5=(\neg x_4\vee x_6\vee x_{11});\quad \omega_6=(\neg x_5\vee\neg x_6)$$
$$\omega_7=(x_1\vee x_7\vee\neg x_{12});\ \omega_8=(x_1\vee x_8);\ \omega_9=(\neg x_7\vee\neg x_8\vee\neg x_{13})\ldots$$




从图中我们可以看出，导致冲突产生的根本原因是第 1 层中将 x9 赋为 0，在第 3 层中将 x10 赋为 0，在第 3 层中将 x11 赋为 0，在第 6 层中将 x1 赋为 1（x1=1@6 表示在第 6 层将 x1 赋值为 1）。由此我们可以直接回溯到第 3 层进行重新赋值，而不是仅仅回溯到上一层，并且我们知道$(x_9=0)\cap(x_{10}=0)\cap(x_{11}=0)\cap(x_1=1)$会导致冲突，我们就可以添加子句$\omega_{10}=(x_9\cup x_{10}\cup x_{11}\cup\neg x_1)$添加进子句库中，在接下里的搜索过程中就能预防相同的变量赋值再次出现。

具体的实现方式见下面的代码：

```
analyze_conflict(){
   cl = find_conflicting_clause();//找到冲突子句 cl
   while (!stop_criterion_met(cl)) {
     lit = choose_literal(cl);//选择一个文字
     var = variable_of_literal( lit );//这个文字所对应的变量命 var
     ante = antecedent( var );//ante 为一个单元子句
     cl = resolve(cl, ante, var);
  //resolve()返回一个子句，除了 var 所对应的文字，这子句需要包含 cl 和 ante 中的所有文字，其中 cl 是一个冲突子句，ante 是一个单元子句，所以返回的 cl 也是一个冲突子句
   }
```

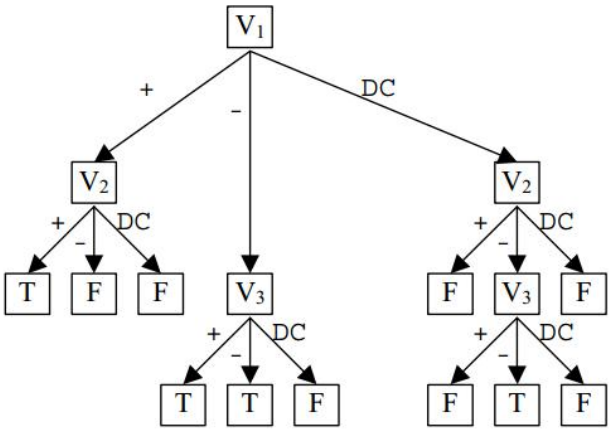
```
    add_clause_to_database(cl);
    back_dl = clause_asserting_level(cl);
    return back_dl;
}
```

========== 3.4 ==========

接下来介绍 3.4 节，主要讲了数据的存储方式，包括 3 种：早期的链表和指针数组的存储方式（GRASP 与 rel_sat 采用），Chaff 采用的数组方式，以及 SATO 采用的 trie 存储方式。

早期的链表存储方式，尽管比较方便对子句进行操作，例如增加和删除子句，但是存储效率很低，因为在处理的过程中因为缺少 access locality（局部访问）的能力，往往会造成很多的 cache misses（缓存丢失）。而 Chaff 采用的数组的方式，尽管没有链表灵活，但是存储效率大大提高，access locality（局部访问）的能力也有大幅提升。

Tire 是以三叉树的方式存储，它的每个内部节点都是都是一个变量的索引，从根节点到叶节点路径上的所有变量组成一个子句。每个节点的三条孩子边分别被标记为 Pos(positive)、Neg(negative)和 DC(dont care)，例如节点 v 的 Pos 边表示文字 v,Neg 边表示文字 $\neg v$，DC 边表示没有这个变量的文字。Tire 的叶节点是 T 或者 F，T 表示 CNF 实例中有这个子句，F 表示没有这个子句。下面是文中给出的一个例子：



从图中我们可以清楚地看出这个 CNF 实例中有哪些子句，以 V1 的左孩子的三条路径举例，第一条路径从根节点到叶节点为（V1，+），（V2，+），T，表示（V1+V2）这个子句是存在的；第二条路径从根节点到叶节点为（V1，+），（V2，-），F，表示（V1+$\neg v2$）这个子句是不存在的；第三条路径从根节点到叶节点为（V1，+），（V2，DC），T，表示（V1）这个子句是不存在的，以此类推，可以得出所有的子句，(V1+V2) (V1’+V3) (V1’+V3’)(V2’+V3’)。

这种存储方法的好处是可以很容易的检索出前半部分相同的子句，这是由三叉树的层级结构所决定的。

========== 3.5 ==========

第 3.5 节介绍了预处理和随机重启。

预处理对应了 Fig2 中的 `preprocess()` 操作，所谓预处理就是根据逻辑蕴含推理以及冗余子句删除和添加等值子句等相关技术，对输入的 CNF 实例进行一系列的化简，然后将化简后的 CNF 公式送至原来的 SAT 求解器进行求解，目标是使得化简后的 CNF 实例能减轻后续的求解工作量。

随机重启机制就是清除现在所有的变量赋值状态，重新选择一组决策变量进行赋值，然后进行正常的搜索过程。主要原因是，由于最初的变量决策顺序可能不是最优的，这就会导致搜索陷入某些子空间中而白白浪费时间。随机重启包括暂停求解和利用之前学到的信息重新启动决策分析这两个阶段。由于重启之前添加的一些学习子句在重启之后依旧存在，因此以前的搜索过程不会白费，反而有利于全局搜索顺序的调整。

**注：红色标注的部分为以前考过的内容**