上海交通大学

硕士学位论文

# 一个高效、健壮的SAT程序的研究与开发

姓名：焦加麟

申请学位级别：硕士

专业：计算机软件与理论

指导教师：徐良贤

2003.2.1

# 一个高效、健壮的 SAT 程序的研究与开发

## 摘 要

尽管在数学上命题逻辑的理论研究已经相当成熟了，但是寻找命题逻辑公式可满足性判定的高效并且健壮的算法（SAT 算法）仍然是一个令人兴奋的研究方向，特别是在 SAT 问题及其算法被越来越多地应用到现实世界的生产生活中的今天。

本文在探讨了 SAT 算法及其实现技术的基础上，描述了作者为了实现一个高效、健壮的 SAT 程序而进行的研究和实验工作，主要内容包括：

第一，为了能够高效地实现 SAT 程序并能够在同等条件下比较各种不同的算法和技术，作者在总结前人成果和经验的基础上，结合自己的想法，设计了一个通用的基于 DPLL 的 SAT 程序框架：3D-DPLL 框架。作者利用 3D-DPLL 框架来实现了自己的 SAT 程序，并且借助 3D-DPLL 作为测试台比较了几种分支决策策略和 SAT 算法；

第二，作者实现了一个新的高效、健壮的 SAT 程序 QuickSAT。在高效地集成了几种已经被证明了能有效提高 SAT 程序运行性能的算法策略后，为了提高程序的效率（Efficiency）和健壮性（Robustness），作者在设计和实现 QuickSAT 时主要做了以下 3 个方面的工作：

（1）为了实现一个更加快速的 BCP 过程，仿照 SATO 的 Head/Tail Lists 模式，作者设计了一种称为“2-文字监视模式”的监控子句状态变化的新方案；

（2）QuickSAT 引进了适用于更加广泛的 SAT 问题的选择分支变量的决策策略和选定分支变量后选择分支的决策策略，其中在分支变量的选择策略中对变量导致冲突的活跃性的评估是基于一个更加广泛的子句集上的。另外，为提高决策过程的动态性，QuickSAT 设置了双重决策过程，可以根据不同的情况选择适合的过程。

（3）为了协调好重启动后原来添加的子句的保留和删除的关系，使得既不能因为保留过多的信息而导致内存空间的紧张，也不能浪费了重启动之前所进行的计算，QuickSAT 使用了一种新的子句数据库的管理方案。这种方案的创新在于决定一个子句是否要从子句数据库中删掉的依据不仅仅是它的长度，同时也考虑这个子句导致冲突的“活跃性”和它存在于子句数据库中时间的长短。

实验结果表明，在性能上，QuickSAT 与当今世界上最高效的非商用完备类 SAT 程序 zChaff 相差无几，而在健壮性上比起几个当今最先进的 SAT 程序都要稍胜一筹，即 QuickSAT 能够在规定时间内解决更多的 SAT 问题。

**关键字**

SAT，DPLL，分支启发式策略，BCP，智能回溯，冲突分析

# RESEARCH AND DEVELOPMENT OF AN EFFICIENT AND ROBUST SAT SOLVER

## ABSTRACT

Although Propositional Logic has been mathematically well studied, looking for efficient and robust algorithms to solve the propositional satisfiability (SAT) problem is still a field desiring us to explore, especially when the applications of SAT problems and algorithms are becoming more and more important in nowadays' real world life and industry.

In this thesis, I describe the work I have done in order to develop an efficient and robust SAT solver, based on the study on many SAT algorithms and their implementation techniques. The work mainly consist of the following parts:

First, in order to improve the productivity of the implementation of SAT solvers and to compare different SAT algorithms and techniques under the same condition, I designed a general DPLL-based framework for SAT solvers, 3D-DPLL Framework, which is a combination of the latest progresses in this field and my own inspiration. And I have developed a SAT solver, QuickSAT, of my own based on the 3D-DPLL Framework. Besides, I have utilized the 3D-DPLL Framework as a testbed for several SAT branching heuristics and SAT algorithms.

Second, I have developed an efficient and robust SAT solver, QuickSAT. Having successfully integrated several strategies and techniques proved effective to improve the run-time performance of SAT solvers, I made my own modifications and improvements to get better efficiency and robustness, including:

(1) In order to make the BCP procedure faster, inspired by the Head/Tail Lists of SATO, I devised a new mechanism, called 2-Literal Watched Schema, to deal with the states transition of clauses whenever variables get assigned;

(2) QuickSAT has a branching heuristic suitable for more comprehensive range of SAT problems, in which the calculation of the variables' activity in causing conflicts is based on a wider set of clauses. And in order to improve the dynamic of the decision procedure, I used two different decision procedures in different situations.

(3) Another improvement is a new clause database management. The innovation of this mechanism lies in that: whether a conflict-induced clause is to be deleted from the database depends on not only its length, but also its activity in conflicts making and how long it has existed in the database.

The result of experimental comparison shows that QuickSAT is close in performance to zChaff, one of the leading SAT solvers in the world today, and is better in robustness than several popular SAT solvers including GRASP, SATO and zChaff, which means QuickSAT can solve more SAT problems in given time than any of these solvers.

**KEY WORDS**

SAT, DPLL, Branching Heuristics, BCP, Intelligent Backtracking, Conflict Analysis

# 上海交通大学

# 学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：焦加鹏

日期：2003 年 2 月 11 日

# 上海交通大学

# 学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□，在___年解密后适用本授权书。

本学位论文属于

不保密☑。

（请在以上方框内打“√”）

学位论文作者签名：焦加鹏　　指导教师签名：

日期：2003 年 2 月 11 日　　日期：2003 年 2 月 11 日

# 第一章 绪论

## 1.1 什么是 SAT？

SAT 是英文 Propositional **Sat**isfiability 的简写。SAT 问题就是命题逻辑公式的可满足性判定问题，即：

给定一个由 $n$ 个命题变量 $v_1, v_2, ..., v_n$ 和 $\wedge$（与）、$\vee$（或）、$\neg$（非）、$\rightarrow$（蕴涵）、$\leftrightarrow$（等价）等连接符以及括号构成的命题逻辑公式 $f(v_1, v_2, ..., v_n)$，判断是否存在一个解释 $j: \{v_1, v_2, ..., v_n\} \rightarrow \{True, False\}$，使得该公式的值为真，如果存在，给出这样一个解释。

由于所有的命题逻辑公式都可以转化为合取范式，并且转换算法的时间复杂度是线性的[1]，而合取范式具有很好的特性，能简化可满足性的判定（要使一个合取范式为真，其包含的所有子句都必须为真，而要使一个子句为真，只需要其包含的文字中有一个为真），因此，SAT 问题一般是指合取范式的可满足性问题。在本文中，我们所谈到的命题逻辑公式，都是指合取范式。

解决 SAT 问题的算法称为 SAT 算法，实现 SAT 算法的计算机软件称为 SAT 程序。

## 1.2 相关概念定义

由于文章展开的需要，在这里有必要先介绍一些基本的概念。

定义 1.1 命题变量和原子公式

命题是表示知识的陈述性形式，命题变量是指任意的一个命题，每个命题变量可能为真，也可能为假。一般命题变量用小写的字母来表示。可以用 1，0 来分别表示命题的真假值，也可以用 T（True）、F（False）来表示。命题变量本身称为原子公式。

定义 1.2 文字、正文字和负文字

原子公式本身称为正文字，其否定称为负文字。正文字和负文字统称文字。文字的“正”、“负”有时称为“符号”(sign)，有时称为“相”(phase)。

定义 1.3 子句

若干个文字的析取构成一个子句。

定义 1.4 合取范式（Conjunctive Normal Form，简写为 CNF）

若干个子句的合取称为合取范式。

定义 1.5 子句长度

一个子句中文字的个数称为这个子句的长度。

定义 1.6 正子句、负子句和混合子句

只含正文字的子句称为正子句；只含有负文字的子句称为负子句；既含有正文字又含有负文字的子句称为混合子句。

定义 1.7 单子句

只有一个文字的子句称为单子句。

定义 1.8 空子句

不含文字的子句称为空子句，记为□。

定义 1.9 赋值和解释

从命题变量集合到真假值集合 $\{0,1\}$ 的函数称为真值赋值，简称赋值，也可以称为解释；如果这个函数没有完全定义（即只有一部分变量具有真假值），那么称为部分赋值或部分解释；为了清楚起见，在本文中，如果提及为单个命题变量确定一个值，使用“赋值”（动词和名词）一词，而谈及对整个命题逻辑公式的命题变量集合的赋值，则使用“解释”一词。一个变量 $x$ 的赋值可以记做 $v(x)$，（$v(x)\in\{0,1\}$），给 $x$ 赋值为 $v(x)$ 可以表示为 $(x,v(x))$ 或者 $x=v(x)$。

定义 1.10 模型、可满足的、不可满足的和重言式

如果有一个解释（可以为部分解释），使公式 $j$ 的值为 1，那么 $j$ 是可满足的，使得它满足的这个解释称为 $j$ 的一个模型；如果所有可能的解释都无法使到 $j$ 满足，则 $j$ 是不可满足的；如果对于所有可能的解释，公式 $j$ 的值都为 1，则称 $j$ 为重言式或永真公式。

定义 1.11 自由变量和自由文字

如果一个公式中的一个命题变量，其真值没有确定，则称为自由变量，自由变量对应的这个文字，称为自由文字。

定义 1.12 纯文字

在一个 CNF 中，如果一个命题变量只以一种形式的文字出现（要么全是正文字，要么全是负文字），则称相应的文字为纯文字。

定义 1.13 K-SAT 问题

如果 CNF 中所有子句的长度都为 K，则该公式所表示的问题称为 K-SAT 问题。

## 1.3 为什么要研究 SAT 问题及其算法？

王浩先生曾在《数理逻辑通俗讲话》[2]中指出：“命题逻辑有着很广的研究领域，但是除了对寻找更快的方法判定全部命题模式有着新的兴趣外，它已经不是研究的主流了。”王浩先生的意思，一方面是指出命题逻辑相关的理论研究已经相当成熟了，另一意思却是：高效的命题逻辑公式可满足性判定（SAT）算法的研究和开发却恰恰相反，是一种“新的兴趣”。在这一点上，我的看法跟王浩先生的看法基本一致，这是因为 SAT 问题和算法有着广泛的应用和重要的意义。

### 1.3.1 SAT 问题及其算法在理论计算机科学中的意义

SAT 是理论计算机科学中的一个经典问题。对于一个有 $n$ 个不同命题变量的命题逻辑公式，如果使用穷举所有可能解释的方法，在最坏情况下，需要穷举 $2^n$ 种不同的情况。它是被发现的第一个 NP 完全问题：在 1971 年 S.A.Cook 提出 NP 完全性理论的文章[3]中，SAT 问题被证明是 NP 完全的，从此，SAT 问题被打上了“intractable”的标记：除非 $P = NP$，否则，SAT 算法不可能有优于问题规模的指数级的时间复杂度，这同时也是所有 NP 完全问题共有的特性。但是，就人们现有的认识水平来看，$P \neq NP$，因此不可能存在具有多项式时间复杂度的 SAT 算法。既然如此，那为什么我还要研究它呢？这是因为：

首先，根据“Cook 定理”，能高效解决一个 NP 完全问题的算法，就能够高效地解决所有其他的 NP 完全问题，这是因为所有 NP 完全问题都可以在多项式时间内互相转化。因此，研究解决 SAT 问题的高效算法，在理论上有着重要的意义。

其次，人们发现，从现实世界的应用领域中“编码”而成的 SAT 问题大多数具有某种结构性，从而使得有针对性的算法能有效地（有效是指算法解决问题的实际运行时间是可以接受的）解决特定种类的 SAT 问题。人们已经发现了很多类问题可以在合理的时间内被解决。事实证明了，由于近年来在搜索剪枝技术上的发展，几个针对结构性 SAT 问题的高效 SAT 判定算法已经被提出并且实现（包括 GRASP[4]，relsat[5]，SATO[6]，Chaff[7]等），这些算法已经能够有效地解决传统方法所无能为力的大规模（大到含数万甚至数十万个命题变量）的 SAT 问题。

在实践中，要解决 NP 完全问题，有些东西必须要放弃，有些要求必须要降低：例如可以限制输入数据的范围和规模；要接受在一定范围内的错误输出；当计算资源耗尽时，必须使用缺省的输出；还要能够接受近似解等等。在有些

情况下，这些限制和牺牲是可以接受的，只要能够满足要解决的问题的最关键的方面，例如，如果算法的错误的输出的几率是极低的；合法的输入不包括那些规模极大的、不可能发生的或人为虚构的问题；近似的解能非常接近于真实解等情况下，用这样的算法去解决 NP 完全问题也是很有实用意义的。事实上，更多的时候，对于 NP 完全问题的算法，我们需要的是寻找那些在平均情况下较为高效的算法，而不必去追求在最坏情况下的高效算法。

### 1.3.2 SAT 问题及其算法在人工智能领域中的应用

SAT 问题及其算法在人工智能领域中有着十分广泛的应用，本节将简要介绍其中特别重要的几种。

（1）CSP 转化为 SAT 问题来解决

约束满足问题（Constraint Satisfaction Problem），简称 CSP，是人工智能领域中被广泛研究的一类问题。CSP 的基本形式是：给定有限个变量（或称为未知量）以及每个变量的取值范围，要求找出所有变量的值，使得一些给定的约束条件被满足。CSP 又分为有限域的 CSP 和无限域的 CSP。如果每个变量的取值集合都是有限集，则是有限域的 CSP，否则是无限域的 CSP。

SAT 问题可以看成 CSP 的一种特例，其中每个（命题）变量的取值范围都是 $\{0,1\}$，而要满足的约束条件就是所给出的命题逻辑公式。

反过来，有限域 CSP 也可以转化成 SAT 问题。一种简单的转化方法如下[8]：

假设 CSP 中的变量为 $x_1, x_2, ..., x_n$，变量 $x_i$ 的取值范围是 $D_i = \{a_{i1}, a_{i2}, ..., a_{in_i}\}$，则对于 CSP 的每个变量 $x$ 和每个可能取值 $a$，引入命题变量 $x:a$（注：这里整个式子代表一个命题变量），这个命题变量为真的时候，表示 $x$ 的值为 $a$。按照这种办法，首先产生如下的子句：

$$x_i : a_{i1} \vee x_i : a_{i2} \vee ... \vee x_i : a_{in_i},$$

$$\neg x_i : a_{ij} \vee \neg x_i : a_{ik} \quad (\forall j,k, 1 \le j < k \le n_i)$$

其中，第一个子句的含义是：变量 $x_i$ 必须从集合 $D_i$ 中取得一个值；第二个子句的含义是：一个变量不能有两个不同的值。

然后，对应于原 CSP 中的每个约束条件 $C$，根据需要，生成若干个如下形式的子句：

$$\neg x_1 : v_1 \vee \neg x_2 : v_2 \vee ... \vee \neg x_m : v_m$$

其中，$x_1, x_2, ..., x_m$ 是 $C$ 中所含的变量。每个这种子句对应着不满足条件 $C$ 的

一种取值组合$\langle x_1 = v_1, x_2 = v_2, \ldots, x_m = v_m \rangle$。

这样就可以把有限 CSP 转化成 SAT 问题，解决了转化所得的 SAT 问题，也就解决了原来的 CSP。

（2）自动定理证明（Automatic Theorem Proving）

我们知道：要证明$\Sigma \vdash a$，只需要证明$\Sigma \cup \{\neg a\}$是不可满足的。根据这一原理，定理证明问题可以转化为命题逻辑公式的可满足性问题。这也就是用消解法证明定理的基本思路：把已知的条件表示成一组子句，把要证明的目标表示成一个子句，把目标的非跟条件子句组合在一起，如果能够通过消解法推出矛盾（出现空子句），就证明了这个目标是成立的。

（3）规划（Planning）[9][10]

规划问题要求我们或机器人找出一个动作序列（规划），以实现某个给定的目标（比如使系统的状态满足一定的条件）。有限步的规划问题可以转化成 SAT 问题。用一些公理来表示问题的初始状态和目标状态，另外一些公理描述动作的前件和后果以及对这些动作的限制。满足所有这些公理的有限模型就是一个规划。因此，可以用可满足性判定过程来构造规划。

将 SAT 程序结合在规划中的应用已经得到了实施，图 1-1 是一个基于 SAT 算法的计算机智能规划系统的体系结构。

SAT 算法在人工智能中的应用还有许多，其他方面的应用包括图像理解（Vision Interpretation） [11]等，由于篇幅问题，在此就不再讨论了。

### 1.3.3 SAT 问题及其算法在 EDA 领域中的应用

SAT 问题及其算法在 EDA（Electronic Design Automation，电子设计自动化）领域中有着广泛且重要的应用，具体应用包括：集成电路的 CAD、 ATPG

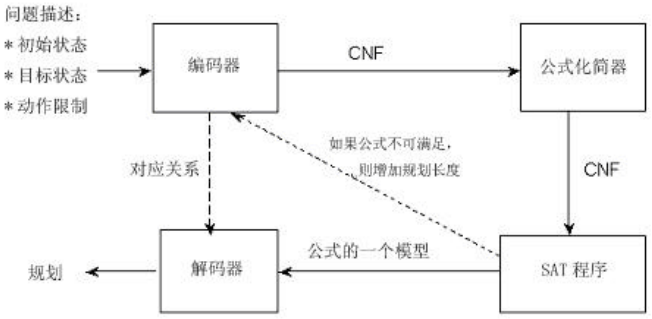



图 1-1 一个基于 SAT 算法的智能规划系统

Fig 1-1 A SAT-Based Planning System

（Automatic Test Pattern Generation，即测试码自动生成）、时序分析、时延故障测试、逻辑电路验证等。

一个例子是 VLSI（超大规模集成电路）的测试与验证。集成电路的可靠性非常重要，即使有细微的错误，也可能引起严重的后果。例如，Intel 公司刚推出 Pentium 处理器后不久，就发现这一批处理器在做浮点运算时会出错，迫使 Intel 公司不得不回收这批芯片，结果蒙受了巨大的经济损失。保障 IC 电路可靠性的主要手段包括等价性验证（Equivalence Verification）和测试生成技术。而 SAT 算法是这两个方面的基础。更详细的叙述，请参考[9]。

### 1.3.4 SAT 算法与算法工程

算法工程，即 Algorithm Engineering，是近年来国际上出现的，为了提高计算机算法的实际运行性能，而将算法的设计、分析、实现、实验测试、改进等过程一体化、工程化的一种新的思路和实践。它强调算法开发各阶段活动的相互作用和算法的实验分析。

SAT 问题及其算法与算法工作有着紧密的关系。这是因为：一方面， SAT 算法是一种复杂的算法，对时间（NP 完全问题）和空间（问题规模非常大）的需求极大，实现的好坏，对其实际运行性能有很大的影响，因此，算法工程方法在 SAT 算法的开发中有着很大的应用；另一方面，不少算法工程中的思想和方法最初都是来自于 SAT 算法的研究与实践中的。因此，研究 SAT 算法可以跟算法工程结合起来，SAT 算法的研究能够作为算法工程研究的一个很好的研究实例，而算法工程的方法与工具的研究反过来又能够指导 SAT 算法的研究与实验。

关于算法工程与 SAT 算法关系的更为详细的讨论请见作者写的文章《算法工程的思想、方法与工具——以 SAT 算法为研究实例》。

### 1.3.5 其它应用

SAT 问题及其算法还有许多应用，包括在运筹学中的资源调度[12]、在 Bounded Model Checking 上、在密码学上、知识工程等方面都有其重要应用，由于篇幅关系，我在这里就不再讨论了。

## 1.4 SAT 问题及其算法研究的历史和现状

SAT 问题及其算法的研究与实践经历了一个起伏的过程。早在 20 世纪 60 年代初，就出现了解决 SAT 问题的一种完备（complete）的基于回溯（backtrack）的搜索算法——Davis-Putnam 算法[13][14]，但是由于 SAT 本身的特性，Davis-Putnam 算法在最坏情况下具有以问题规模为指数的时间复杂性。在 S.A.Cook 证明了 SAT 问题的 NP 完全性之后，人们对 SAT 的研究的兴趣慢慢减弱；再加

上当时的逻辑学家和人工智能工作者普遍认为命题逻辑的表达能力过于有限而把主要精力放到一阶逻辑、模态逻辑等更强有力的形式化工具的研究中去。这一切都导致了对 SAT 的研究在较长的一段时间内没有得到足够的重视。但是，到了 20 世纪 90 年代，情况发生了变化，人们对 SAT 的研究兴趣空前高涨。从德国人 Michael Buro 和 Hans Kleine Büning 在 1991 到 1992 年间举办的 SAT 竞赛开始，世界上各地已举办了多次 SAT 竞赛，其中包括 1993 年美国 DIMACS 举办的第二届 DIMACS Implementation Challenge 和 1996 年在北京举办的 SAT 快速算法国际竞赛，现在，SAT 竞赛一般伴随着每年一届的 SAT 研讨会而举办；从 DIMACS 在 1996 年举办了一届名为“Satisfiability Problem: Theory and Applications ”的研讨会后，同年在意大利的 Siena，1998 年在德国的 Paderborn，2000 年在荷兰的 Renesee，2001 年在美国的 Boston，2002 年美国的 Cincinnati 都先后举办了 SAT 的研讨会；众多新的算法被提了出来；众多新的高效 SAT 解题程序被开发……可以说，在上个世纪的最后 10 年中，SAT 问题及其算法的研究与实践取得了极大的成果：一方面，SAT 算法在许多实际应用中发挥出其强大的作用，另一方面，SAT 算法及其实现程序在解决问题的效率方面也得到了极大的提高，请看下面的比较数字：

| 年代 | 相同时间内能解决的最大规模的问题 | |
|---|---|---|
| | 变量数 | 子句数 |
| 1990 年 | 100 | 200 |
| 1998 年 | 10，000-100，000 | 1，000，000 |

表 1-1 相同时间内能解决的问题规模增大了几个数量级（数据来源[15]）

| 基准测试实例 | SAT 算法 | | | |
|---|---|---|---|---|
| | POSIT（1994 年） | GRASP （ 1996 年） | SATO （ 1998 年） | Chaff （ 2001 年） |
| ssa2670-136 | 40.66s | 1.2s | 0.95s | 0.02s |
| bf1355-638 | 1805.21s | 0.11s | 0.04s | 0.01s |
| pret150_25 | >3000s 而终止 | 0.21s | 0.09s | 0.01s |
| dubois100 | >3000s 而终止 | 11.85s | 0.08s | 0.01s |
| aim200-2_0-no-1 | >3000s 而终止 | 0.01s | 0s | 0s |
| 2dlx_..._bug005 | >3000s 而终止 | >3000s 而终止 | >3000s 而终止 | 2.9s |
| c6288 | >3000s 而终止 | >3000s 而终止 | >3000s 而终止 | >3000s 而终止 |

表 1-2 解决同样的问题所需要的时间大大减少（数据来源[16]）

这些比较数字表明了在解决 SAT 问题的能力上的进展是非常惊人的，其间接反映出来的一面是在 SAT 问题及其算法上的研究取得突破的潜力是巨大的。

必须承认，在当今计算机科学研究的众多领域中，SAT 问题及其算法的研究不是热门，但是，随着我对这个领域的了解，我发现，在这个世界上，我并不孤独。虽然就我国国内的研究情况而言，除了在 1996 年举办北京 SAT 竞赛的时候曾经引起过一定的热潮以外，相关的研究并不是特别引人注意，但是，仍然能够找到一些研究成果。国内文章，据我所知的有两篇，一篇是清华大学

贺思敏的博士论文《可满足性问题的算法设计与分析》（1997 年），贺利用了吴文俊提出的一套在初等几何的机器证明中求解多项式方程组的算法来解决 SAT 问题。中科院计算技术研究所的硕士卜东波也曾以《命题逻辑的可满足性问题：复杂性和算法》（1997 年）作为其学位论文，张健博士更是出版了一本题为《逻辑公式的可满足性判定——方法、工具及应用》的专著。

放眼国际，跟国内不一样的是，SAT 方面的研究在最近几年发展比较大。著名的，美国普林斯顿大学、卡内基梅隆大学、德州大学奥斯汀分校中就有专门研究 SAT 算法及其应用的研究所。在欧洲，德国和葡萄牙的高校中，对 SAT 算法的研究也是非常流行。

## 1.5 本文的研究内容及文章的结构

本文研究的内容包括：对 SAT 算法及其实现技术进行研究，其中研究的重点在于完备类的 SAT 算法及程序；在此基础上，开发一个新的完备的 SAT 程序。在该 SAT 程序的开发中，结合当今 SAT 研究的最新理论和技术，对各种算法和技术进行整合和修正，并提出自己的改进方案，同时利用算法工程的思想、方法和工具来进行该 SAT 程序的开发。这一切的目的都是为了提高该 SAT 程序的效率和强壮性，使之成为一个高效、健壮的 SAT 程序。

下文的大致内容如下：第二章主要研究了 SAT 算法的基本理论和技术；第三章简要介绍了基于 DPLL 的 SAT 算法的实现技术，分析讨论了当前流行的 3 个高效的 SAT 程序的主要特点；第四章将给出我设计的基于 DPLL 算法的 SAT 程序的一个通用框架 3D-DPLL；第五章详细论述了我所实现的 SAT 程序 QuickSAT 的设计方案和实现细节；第六章是结论与展望；附录部分列出了主要的实验测试结果。

# 第二章 SAT 算法概述

## 2.1 概述

一直以来，解决 SAT 问题的算法，主要有两大思路，第一种是基于穷举的思想，系统化地考察公式的所有可能的解释，如果至少存在一个模型，则该公式是可满足的，否则，该公式是不可满足的。这种算法称为完备（complete）算法；第二种算法是基于局部搜索的算法，这种算法不能保证考察所有的解释，只是根据一定的策略去寻找一个模型，这种算法称为不完备（incomplete）算法。在理论上，如果一个公式是可满足的，完备算法一定能够找到一个模型，如果一个公式是不可满足的，完备算法能够证明其不可满足性，但是，完备算法在最坏情况下具有指数级的时间复杂度，对于极大规模的问题，在现有的计算资源条件下，可能无法在人们可以承受的时间内给出解答；不完备算法无法证明公式的不可满足性，当一个公式是不满足时，将陷入无休止的运行中，但是，对于可满足的问题，在很多情况下，能够高效地找出其模型。

完备算法主要包括 DPLL 算法及其变种。对 DPLL 算法的不断改进使得当今的完备算法能够有效地解决从现实问题建模而来的很大规模的 SAT 问题，对于这样的具有“结构性”的问题，改进了的完备算法具有不完备算法所难以比拟的优势，特别是对于不可满足的问题，不完备算法根本难以发挥作用。而在现实应用中，解决很多重要的问题，都要求证明公式的不可满足性。
因此，一直以来，大多数 SAT 算法都是属于完备类的，而且当今最高效的算法也是完备的。完备算法的优点是在理论上保证能获得解，能够同时判定可满足性和不可满足性，并能高效地解决结构化的 SAT 问题。基于这些原因，我的主要研究内容也是完备算法，因此，在本章中，我们将主要介绍完备算法。

## 2.2 完备算法

### 2.2.1 DPLL 算法

#### 2.2.1.1 历史

在 20 世纪 60 年代初，Davis 和 Putnam 提出了一种解决 SAT 问题的完备算

法[13]，最初的算法称为 Davis-Putnam 算法，借助于以下消解规则（resolution）来将公式的变量一个一个地消掉：

$$(A\vee p)\wedge(B\vee\neg p)\wedge R\Rightarrow(A\vee B)\wedge R$$

或

$$(x\vee C_1)\wedge(x\vee C_2)\wedge\ldots\wedge(x\vee C_p)\wedge(\neg x\vee D_1)\wedge(\neg x\vee D_2)\wedge\ldots\wedge(\neg x\vee D_q)\wedge U$$

$$\Rightarrow((C_1\wedge C_2\wedge\ldots\wedge C_p)\vee(D_1\wedge D_2\wedge\ldots\wedge D_q))\wedge U$$

$$\Rightarrow(C_1\vee D_1)\wedge(C_1\vee D_2)\wedge\ldots\wedge(C_1\vee D_q)\wedge(C_2\vee D_1)\wedge(C_2\vee D_2)\wedge\ldots\wedge(C_2\vee D_q)$$

$$\wedge\ldots\wedge(C_p\vee D_1)\wedge(C_p\vee D_2)\wedge\ldots\wedge(C_p\vee D_q)\wedge U$$

注意上式中的 $A$、$B$ 和 $R$ 不含 $p$ 或 $\neg p$，$C_i(1\le i\le p)$、$D_j(1\le j\le q)$ 和 $U$ 都不含 $x$ 或 $\neg x$。

当原公式被化简到无法再进行消解时，如果出现空子句，则公式不可以满足，否则公式可满足。但是这个算法通常产生指数级的空间复杂性（例如上面的第二条公式中，在消解之后，子句数目增加了 $p\times q-(p+q)$ 个子句）。因此，在 Davis-Putnam 算法提出不久，Davis，Logemann 和 Loveland 就将其消解规则去掉，取而代之的是一条“分裂规则”[14]：

$$(A\vee p)\wedge(B\vee\neg p)\wedge R\Rightarrow\begin{cases}p=0, A\wedge R\\ p=1, B\wedge R\end{cases}$$

或

$$(x\vee C_1)\wedge(x\vee C_2)\wedge\ldots\wedge(x\vee C_p)\wedge(\neg x\vee D_1)\wedge(\neg x\vee D_2)\wedge\ldots\wedge(\neg x\vee D_q)\wedge U$$

$$\Rightarrow\begin{cases}x=0, C_1\wedge C_2\wedge\ldots\wedge C_p\wedge U\\ x=1, D_1\wedge D_2\wedge\ldots\wedge D_q\wedge U\end{cases}$$

该规则将原问题分裂为两个较小的问题。经过这次修改后的算法，称为 DPLL 算法。

#### 2.2.1.2 DPLL 算法描述

假定给出的命题逻辑公式为 $S$（一组子句，也可以看成子句的一个数据库），DPLL 算法不断地按照如下规则对 $S$ 进行化简，直到无法再进行为止：

（1）重言式规则

删掉 $S$ 中所有的重言式，剩下的子句集记为 $S'$。$S$ 可满足，当且仅当 $S'$ 可

满足。

⑵）单子句规则

如果 $S$ 中有一个单子句 $L$（在这里，我们用 $L$ 同时表示一个单子句和构成这个单子句的那个文字），那么要使原公式满足，$L$ 必须取真值。我们可以从 $S$ 中删去所有包含 $L$ 的子句（包括单子句本身），得到集合 $S_1$，如果 $S_1$ 是空集，则 $S$ 可满足，否则，检查 $S_1$ 中的每个子句，如果包含文字 $\neg L$，则从该子句中去掉这个文字，由此得到子句集 $S_2$，$S$ 可满足，当且仅当 $S_2$ 可满足。

⑶）纯文字规则

如果文字 $L$ 是纯文字，则从 $S$ 中删去包含 $L$ 的所有子句，得到集合 $S'$。$S$ 可满足，当且仅当 $S'$ 可满足。

⑷）分裂规则

假设 $S$ 可以写成如下形式：

$$(A_1 \vee l) \wedge ... \wedge (A_m \vee l) \wedge (B_1 \vee \neg l) \wedge ... \wedge (B_n \vee \neg l) \wedge C_1 \wedge ... \wedge C_l$$

这里 $A_i, B_j, C_k$ 都是子句，而且都不含 $l$ 或 $\neg l$，在这种情况下，生成两个比 $S$ 简单的子句集：$S_1 = A_1 \wedge ... A_m \wedge C_1 \wedge ... \wedge C_l$ 和 $S_2 = B_1 \wedge ... B_n \wedge C_1 \wedge ... \wedge C_l$，则 $S$ 可满足，当且仅当 $S_1$ 可满足或者 $S_2$ 可满足。

我们可以如此来理解分裂规则：如果文字 $l$ 取真值，可将 $S$ 化简为 $S_2$（等价于先把单子句 $l$ 添加到子句集中，然后施用单子句规则）；而 $l$ 取假值时，$S$ 可以化为 $S_1$（等价于先把单子句 $\neg l$ 添加到子句集中，然后施用单子句规则）。

按照上述规则，算法不断地对子句集进行化简。算法在开始的时候，先执行重言式规则、单子句规则、纯文字规则，直到不可再使用这些规则化简为止，此时，如果子句集变成空集，则原公式可满足，终止算法；如果子句集中出现空子句，则表示原公式不可满足，亦终止；如果不是上述两种情况，则此时必须执行分裂规则来推动 DPLL 过程的进行。分裂规则必须根据一定的策略，选出一个变量 $l$（称为分支变量或决策变量，而选择这个变量的过程，往往称为决策），根据它取真和假两种不同的值来将原问题分化为两个子问题，再分别对两个子问题施行 DPLL 算法，……，依此类推，这两个子问题又可能再生成自己的子问题，再对这些子问题分别施行 DPLL 算法，……，如果原问题根据 $l$ 而生成的两个子问题中有一个是可满足的，则原问题可满足，否则，原问题不可满足。算法显然是终止的，其执行过程可以看成是深度优先（Depth First)遍历一

棵二叉决策树。 在树中的每个结点都有一个变量被赋值，左右子树分别对应该变量获得不同赋值（0 或 1）后再施行一定的规则（单子句规则、纯文字规则）所得的子句集。在搜索过程中，一旦在某个结点上出现了冲突（即出现了空子句），就不必再搜索该结点的子树了（剪枝），回溯到该结点的父结点，如果父结点的另一子树没有遍历过，则遍历该子树，否则，继续回溯到父结点的父结点，如此下去……直到遇到一个满足公式的解释（模型）或者遍历完整棵树为止（此时，公式是不可满足的）。DPLL 算法的伪码表述如下：

```
DPLL ( S )
    if ( S 包含一个空子句)
        return unsatisfiable;
    if ( S 为空) {
        输出当前的解释;
        return satisfiable;
    }
    if ( S 包含单子句 { L1 }) {
        从 S 中删掉所有含有 L1 的子句，并从剩下的子句中
        删掉所有的 ¬L1，得到新的子句集 S';
        return DPLL ( S');
    }
    if ( S 包含有纯文字 L2 )
        return DPLL ( S ∪ { L2 } ) ;
    选择一个分支变量 L3 ;
    if ( DPLL ( S ∪ { L3 } ) == satisfiable )
        return satisfiable
    else
        return DPLL ( S ∪ { ¬L3 } )
```

#### 2.2.1.3 BCP 过程

反复使用单子句规则对子句集进行化简的过程，称为单子句增殖（unit propagation）或布尔约束增殖（Boolean Constraint Propagation），简称 BCP。BCP 是 DPLL 中使用的一种重要的前瞻（lookahead）技术。在 BCP 过程中，冲突表现为空子句的出现，而公式被满足则表现为整个子句集变为空。

#### 2.2.1.4 分支启发式策略（Branching Heuristics）

DPLL 算法中一个重要的内容是在搜索树的每个结点上选择分支变量（决策变量）以及选定了分支变量后选择对这个变量首先进行的赋值时使用的启发式策略。分支启发式策略是否高效对整个搜索树的大小有很大的影响，因此分支策略的好坏对整个算法的效率有极大的关系，所以人们提出了各种各样的许多分支策略。多年来，尽管人们做了大量的实验比较各种分支策略，但是，仍然缺乏足够的统计证据来表明某种决策策略明显优于另外一种策略，因此，各种分支策略的优劣都是相对而言的，有些在解决某类问题的时候优于其他策略，但是在解决另外一类问题的时候却又显得不是那么高效。在这样的情况下，找出评估各种策略的方案就显得非常重要的。常用的分支策略评估尺度包括：（1）SAT 程序解决一个问题所需执行的决策次数；这种度量尺度的合理性在于：较少的决策次数一般意味着决策策略是比较优化的；但是，从另外一个观点来看，由于并不是每次决策之后都会导致同样数量的 BCP 操作，因此有可能少的决策数反而导致较多的 BCP 操作，同理，同样的决策次数可能会导致不同次数的冲突的出现。本质的问题也就是，不同的决策策略具有不同的计算开销。因此根据决策次数来评估分支策略并不一定是最好的，其中一种不适用的情况就是该种分支策略对应的运算开销很大的时候。（2）针对上一种评估方案的缺陷得出了一种更加实际的策略，那就是从运行时间上来比较各种决策策略的优劣。

最简单的分支策略莫过于从所有未赋值的变量中随机选出下一个决策变量；另外一个极端的做法是，选择使得某种相当复杂的以当前变量状态和子句数据库为输入的评价函数取得最大值的变量作为分支变量（这种例子包括 BOHM[17]和 MOM[18]）。另外一种启发式策略 DLIS（Dynamic Largest Individual Sum）[19]介于上述两个极端中间，其做法是选择出现在未定值的子句中次数最多的文字作为分支文字，这种策略被用于算法 GRASP 中。更为复杂的启发式策略包括 JW-OS[20]等等。

人们在设计一种新的启发式分支策略的时候，总是有着某种指导思想，例如使得选出的分支变量被赋值后能够消掉更多的子句，选择所对应的正文字个数跟负文字个数一样多的变量作为分支变量……大多数分支策略是与变量状态相关（Variable State Dependent）的，也就是说，以前的变量的赋值会影响到后面的新的决策变量的选择。但是，由于变量状态相关的策略每次计算决策变量的时候，都必须考察变量的状态，因此计算开销非常大，计算分支变量的决策过程本身成了整个计算的一个瓶颈。因此，越来越需要一种更高效的分支启发策略。zChaff 的作者提出了一种新的启发式策略，称为“变量状态独立的计数值老化策略（Variable State Independent Decaying Sum，缩写为 VSIDS）”[7]。VSIDS

是与变量状态无关的半动态的策略，实验证明：VSIDS 能有效提高整个算法的性能。关于 VSIDS 的更详细的介绍，见第 3.3.3 节。

#### 2.2.1.5 回溯

原始 DPLL 算法使用的回溯方法是传统的“按照时间先后顺序的”回溯，也就是每次回溯都只是回溯到紧接着的上一层结点，回溯之后必须要取消相应的赋值以及由于这些赋值而对子句集所作的一切修改。为了跟后文中的智能回溯区别开来，我把这种回溯称为简单回溯。

### 2.2.2 对 DPLL 算法的改进

#### 2.2.2.1 智能回溯

智能回溯（Intelligent Backtracking），也就是非时间先后顺序的回溯（Non-Chronological Backtracking），也有人称为冲突制导回跳（Conflict-Directed Backjumping）。智能回溯的思路是这样的：假设在搜索树中的某个结点先遍历的一棵子树中不存在公式的模型，则显然无论在该棵子树中如何搜索都是毫无意义的，在子树内部的任何简单回溯都是无法跳出这个不存在解的区间，如果这棵子树是一棵很深很大的子树，那么简单回溯只会导致大量计算的浪费。针对这种情况，一种智能的回溯策略是：如果能够认识到当前的这个结点是造成其一棵子树中所有赋值都会导致冲突的根源，就可以在遇到冲突的时候，直接“回跳”到这个结点，放弃对这棵子树其它部分的搜索，马上搜索这个结点的另外一棵子树。

智能回溯引入了决策层次的概念：

定义 2.1 决策层次（Decision Level）

每个赋值都对应一个决策层次，表明它在决策树中的深度。同一个决策树结点上的决策赋值和导出赋值（就是那些不是决策赋值，而是在 BCP 过程中推导出来的赋值）具有相同的决策层次。在决策树根的赋值对应的决策层次为 1，第 $n$ 层结点的决策层次为 $n$；一个变量 $x$ 如果在决策层次 $d$ 上被赋值 $v$，则记为 $x = v@d$；对于决策树上的结点 $V$，其决策层次记为 $dl(V)$。例如，$x = v(x)@dl(x)$ 表示变量 $x$ 在决策层次 $dl(x)$ 上被赋值为 $v(x)$。

在原始 DPLL 算法中，任何一个分支结点，设决策变量为 $x$，则当赋值 $x = v$ 导致冲突后，会自动选择其反相赋值 $x = \neg v$。但是，很多时候赋值 $x = \neg v$

仍然会导致冲突，这是因为导致冲突的原因并不是决策变量 $x$，而是某个（些）决策层次小于 $dl(x)$ 的决策变量，因此，只有回溯到这种导致冲突的结点处，改变其赋值，才有可能找到满足的解，“剪枝”掉了无用的搜索。

#### 2.2.2.2 冲突分析（Conflict Analysis）与学习（Learning）

智能回溯的实现是依赖于冲突分析的。不对冲突进行分析，找出真正导致这个冲突的原因，根本不可能进行智能回溯。所谓冲突分析，通俗地说，就是在产生冲突的时候，分析冲突的原因，分析的结果主要有两个方面的用途：一是作为智能回溯的依据；二是通过把分析到的结果编码成子句，添加到子句数据库中，起到学习的作用，这样就可以帮助以后的搜索工作对某些搜索区间进行剪枝，避免再次发生同样原因而导致的冲突。

为了进一步的讨论，我们先介绍一下有向蕴涵图的概念。

定义 2.2 蕴涵（导出）

如果当前状态下公式 $j$ 的一个子句 $w$ 中唯一的自由变量是 $x$，其他文字都为 0，并且设这些文字对应的赋值集合是 $s:\{y \mid y\in w\wedge y\neq x\}\to\{0,1\}$，则要使公式 $j$ 得到满足，$x$ 的取值必须使它在子句 $w$ 中出现的文字为真，假设 $x$ 的取值必须为 $v$，这种情况下，我们就说“$s$（由子句 $w$）蕴涵（或导出）$x=v$”。

定义 2.3 前提赋值集合和前件子句

设一个变量 $x$ 是子句 $w=(l_1\vee l_2\vee\ldots\vee l_k)$ 当前状态下唯一的自由变量，则根据单子句规则，可以推出 $x$ 的某种赋值 $v(x)$。定义 $x=v(x)$ 的“前提赋值集合”（Antecedent Assignment）为子句 $w$ 中的除了 $x$ 以外的所有变量的赋值集合，记为 $A^w(x)$。$A^w(x)$ 包含那些对（由子句 $w$）导出 $x$ 的赋值负有直接责任的变量赋值。我们规定，对于一个决策变量，其前提赋值集合为空。另外，如果 $x$ 是非决策变量，我们称子句 $w$ 为赋值 $x=v(x)$ 的前件子句。

例如：对于子句 $w=(x\vee y\vee\neg z)$，分别有 $A^w(x)=\{y=0,z=1\}$，$A^w(y)=\{x=0,z=1\}$ 和 $A^w(z)=\{x=0,y=0\}$。

BCP 过程中导出各个赋值的过程可以由一个有向蕴涵图 $I$ 来描述，$I$ 的定义如下：

（1）$I$ 中每一个结点都对应于一个变量赋值 $x=v(x)$；

（2）在 $I$ 中结点 $x=v(x)$ 的所有前趋结点就是 $x$ 的前提赋值集合 $A^w(x)$（对应于导致 $x$ 赋值的一个单子句 $w$），所有从 $A^w(x)$ 中的结点指向 $x=v(x)$ 的有向边上都标记上 $w$。在 $I$ 中，没有前趋结点的结点就是对应于决策赋值的结点。

（3）特殊的冲突结点（即并没有对应的赋值，只是用来表示一个冲突的出现）被添加到 $I$ 中。一个冲突结点 $K$ 的所有前趋结点是导致相应的子句 $w$ 变得不可满足（即变成空子句）的原因，这些前趋结点构成冲突结点的前提赋值结合 $A^w(K)$。

在算法的实现中，有向蕴涵图的维护方法是：为每个赋值了的非决策变量设置一个指向其前件子句的指针，通过索引这种指针，就可以把蕴涵图构造出来。

基于上述定义，一个导出变量 $x$ 的决策层次是：

$$dl(x)=\max\{dl(y)\mid(y,v(y))\in A^w(x)\} \qquad (2\text{-}1)$$

原始的 DPLL 是没有冲突分析的，当遇到冲突时，如果决策变量的另一种可能的赋值还没有尝试，则将该决策变量取相反的值，如果两种赋值都已经尝试过了，则回溯到再上一层结点。冲突分析的思想最先被应用到解决 SAT 问题是在算法 relsat[5]中，并且在 GRASP[4]得到了更好的发展。冲突分析的一个成果是一个或者多个额外的子句被生成，这些子句包含了导致这个冲突的原因。把这些子句记录下来的过程就称为学习。从一个冲突中学习，可以阻止算法在以后的搜索中再次搜索会导致相同冲突的区间。

定义 2.4 冲突赋值集合和冲突导出子句

令 $S$ 代表当前的子句集，它包含了初始的子句集 $S^*$ 和在搜索过程中推出的子句。设子句 $C\in S$ 和一个变量赋值集合 $R$，如果执行了 $R$ 中的赋值之后再运行 BCP 会导致子句 $C$ 变为 0，则称 $R$ 为导致 $C$ 不满足的冲突赋值集合（简称为冲突赋值集合）。

假设子句 $C$ 为 $\neg a\vee x\vee\neg c$，则导致 $C$ 出现冲突的一种简单的冲突赋值集合为 $\{a=1,x=0,c=1\}$（我们可以称这样的冲突赋值集合为“直接冲突赋值集合”），但是，这个冲突赋值集合并没有什么意义，因为它并没有提供任何新的信息（这些信息都反映在子句 $C$ 本身里面了）。如果 $R=\{x=0,y=1,z=1\}$ 也是导致 $C$ 最后变成 0 的一个赋值集合（这样的赋值集合我们可以称为“间接冲突赋值集合”），

则由 $R$ 构造的子句 $x \vee \neg y \vee \neg z$ 就是当前子句集 $S$ 推出的一个新子句，并且包含了 $S$ 中所没有的信息。通过将子句 $x \vee \neg y \vee \neg z$ 加入到子句集 $S$ 中（称为“子句记录”），我们就把冲突赋值集合 $R$ 给记录了下来。这个子句 $x \vee \neg y \vee \neg z$ 称为冲突导出子句。

假设 $R=\{x=0,y=1,z=1\}$ 中的赋值 $x=0$ 是在当前决策结点 $v$ 上获得的（$x=0$ 可以是决策赋值，也可以是导出赋值），赋值 $y=1$ 和 $z=1$ 分别是其他决策层次上的结点 $v'$ 和 $v''$ 上的决策变量，显然有 $dl(v)>dl(v')>dl(v'')$ 或者 $dl(v)>dl(v'')>dl(v')$。在把子句 $x \vee \neg y \vee \neg z$ 记录下来之后，回溯过程就可以“回跳”到 $v'$ 或者 $v''$ 上（跳过了位于 $v$ 和 $v'$ 或者 $v''$ 之间的所有结点），尽管回跳后施行了恢复操作，回跳之后的子句集跟原来搜索过程第一次到达这个结点时的子句集是有区别的，因为此时子句集中至少多了一个新的子句 $x \vee \neg y \vee \neg z$。假设回跳到 $v'$ 上，并且 $dl(v)>dl(v')>dl(v'')$，回跳到 $v'$ 后，由于新添加的子句 $x \vee \neg y \vee \neg z$ 的存在，而 $y=1$ 和 $z=1$ 分别在 $v'$ 和 $v''$ 上获得赋值，则可以推出 $x=1$，这是在把子句 $x \vee \neg y \vee \neg z$ 添加到子句集之前不可能做出的推断。

关于冲突分析和冲突导出子句生成的详细论述，可以参看[4]。在这里我们只是介绍其主要思想。假设在搜索过程中决策树上的当前结点 $v$ 上，子句 $\neg a \vee x \vee \neg c$ 变得不可满足。此时，我们需要找出一个间接冲突赋值集合 $R$，使得其中只有一个变量赋值是在当前结点 $v$ 上获得的，而其他的变量赋值都必须是在比 $v$ 更小的决策层次上的结点上获得的。

这样的一个间接冲突赋值集合可以通过从当前的直接冲突赋值集合 $R_1=\{a=1, x=0, c=1\}$ 开始往回“逆向”执行 BCP 过程而得到。例如，假设 $R_1$ 中的每个变量赋值都是在当前结点 $v$ 获得的，并且其中的 $a=1$ 是由单子句 $a \vee x \vee \neg z$ 所导出的，前提赋值集合为 $\{x=0, z=1\}$，现在，$R_1$ 可以被一个间接冲突赋值集合 $R_2=\{x=0, c=1, z=1\}$ 来替代（$R_2$ 是通过把 $R_1$ 中的 $a=1$ 用 $x=0, z=1$ 来替代所得）。假设其中的赋值 $z=1$ 是在结点 $v'$（$dl(v)>dl(v')$）上获得的。现在冲突赋值集合 $R_2$ 已经由原来所有变量赋值都是在当前结点获得的变成了只剩

下两个变量赋值是在当前结点获得的了（即 $x=0, c=1$）。继续按照这样的方法逆向施行 BCP 过程，我们将最终得到一个只有一个赋值是在当前结点上获得的冲突赋值集合，例如，假设赋值 $c=1$ 是由于子句 $c \vee \neg y \vee \neg z$ 所导出的，并且 $y=1$ 是在结点 $v''$（$dl(v) > dl(v'')$）上获得的，则通过把 $R_2$ 中的 $c=1$ 用 $y=1, z=1$ 来代替就可以得到新的冲突赋值 $R_3 = \{x=0, y=1, z=1\}$，现在 $R_3$ 只含有一个变量赋值 $x=0$ 是在当前结点获得的，这就满足冲突分析的要求，由 $R_3$ 构造的冲突导出子句是 $x \vee \neg y \vee \neg z$。

本质上，上述新子句 $x \vee \neg y \vee \neg z$ 的导出过程可以由下图的消解链来描述：

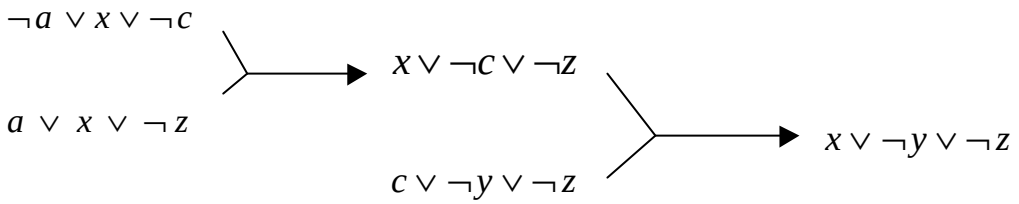



图 2-1 生成冲突导出子句的消解链

Fig 2-1 The Resolution of a Conflict-Induced Clause

而只有这个消解链末端的冲突导出子句才被添加到当前的子句集中，其他中间结果（例如 $x \vee \neg c \vee \neg z$）将被舍弃。在这个消解过程中所用到的在当前子句集中的子句都被称为“对这个冲突有责任的子句”。在这个例子中，$\neg a \vee x \vee \neg c$、$a \vee x \vee \neg z$、$c \vee \neg y \vee \neg z$ 都是这种子句。

#### 2.2.2.3 随机化与重启动

原始 DPLL 算法另的一个问题是，在搜索过程早期所犯错误的代价是非常大的。在 Gent 和 Walsh 的文章[21]中举出了这样一个例子，用 DPLL 过程解决该问题的时候，在搜索的第一个分支处就进入了不可满足的子问题，而搜索过程在这个子树上一直扩展了 350，000，000 个分支才回溯到根结点，然后才开始另外一边子树的搜索，而这次，却可以毫不费力地找到了一个模型。对于这种不足，Gomes 等人[22]提出的基于随机化（Randomization）和快速重启动（Rapid Restart）思想的策略能有效地在减少这样的早期错误。

所谓随机化是指将随机性引入到分支变量的选择中，例如，当使用某种启

发式分支策略的时候，如果有多个候选的分支变量具有同样大的分值，则随机地从中选择一个作为分支变量。其实，随机化在整个 DPLL 过程中可用的地方很多，除了可以用于分支变量的选择中，还可以用于例如智能回跳中，从多个可能的回跳目的层次中随机选出一个作为当前的回跳目的层次等等。

所谓重启动是指：如果在搜索过程中遇到了冲突后回溯的层数超过了这个某个预设的值或者冲突的次数超过了特定的次数仍然没有找到一个模型，就直接回到树的根结点，转而搜索另外一个分支，或者完全抛弃当前的搜索树，重新选择第一个决策变量，开始新的搜索。完全放弃了原来正在搜索的搜索树，好像造成了计算的浪费，但是通过跟学习结合起来，可以对重启动后的搜索有所帮助。

随机化和重启动中所使用的参数都是可以设置的，包括随机化中的概率和重启动中的最大冲突次数等；而具体值的设定必须依赖理论计算和经验分析，而证明其有效性的唯一手段就是实验测试和比较。

### 2.2.3 完备算法的时间复杂度

人们已经证明了原始的 DPLL 算法在最坏情况下的时间复杂度的上界是 $O(1.696^n)$（$n$ 为命题变量数），而有文章[23]指出，只要对 DPLL 进行一些细微的修改就可以将算法在最坏情况下的时间复杂度优化为 $O(1.618^n)$。Rodosek 给出了一个上界为 $O(1.476^n)$ 的改进的 DPLL 算法，其要点是在每次分支的时候，都要消去至少两个变量[24]。Beigel 和 Eppstein 给出了一个将 3-SAT 问题先转化为 2-SAT 问题，在得到解后再尝试将该解扩展回原问题的解[25]。这种算法在最坏情况下的时间复杂度是 $O(1.381^m)$（$m$ 是子句数）。然而，SAT 问题中的子句数往往远远大于命题变量数，所以这种算法其实并不大高效，上述的 Rodosek 的算法在 $m/n > 1.206$ 的时候就比该算法好。

## 2.3 不完备算法

### 2.3.1局部搜索

局部搜索（Local Search）是解决许多 NP 难问题（特别是最优化问题）的一种常用方法。比较解决 SAT 问题的回溯搜索法与局部搜索法：前者是一个一个地确定命题变量的值，逐渐地将部分解释扩充为完整的解释。而后者的基本思想是，先任意地给每个命题变量取一个值，得到一个解释，它可能使一部分

子句的值为真，另一部分子句的值为假。然后反复地对现有解释进行局部调整（例如改变一个或若干个变量的值），使得被满足的子句个数越来越多。在理想情况下，最终得到一个解，它使所有的子句被满足。

基于上述思想，Jun Gu 给出了解决 SAT 问题的一系列搜索过程[26] [27]，分别命名为 SAT1.0、SAT2.0、SAT3.0 等等。

### 2.3.2贪心算法 GSAT

Bart Selman 等人[28][29]也于 1992 年前后提出了贪心搜索过程 GSAT，其思路如下：对问题的一种解释可以看成是 $n$ 维空间（假设子句集 $S$ 中有 $n$ 个命题变量）中的一个点，一共有 $2^n$ 个点。对每一个点（即公式的一种解释）$q$，在该解释下为假的子句个数记为 $g(q)$。如果我们能够找出一个点 $a$，使得 $g(a)=0$，那么该子句集就是可满足的。搜索开始的时候，我们从搜索空间中随机地选取一点，然后在每一步，我们检查当前点的相邻点，如果某个相邻点的目标函数 g 的值比在当前点的值小，就以那个相邻点替换当前点。而相邻点的定义可以有多种，在 GSAT 中，如果两个赋值仅有一个变量的值不同，那么它们就是相邻点。因此，该过程的一个基本操作是将某个变量的值取反，称为翻转（flipping）。GSAT 算法进行 MAX-TRIES 次的尝试，在每次尝试开始的时候都随机生成一个完整的解释，然后对这个解释进行 MAX-FLIPS 次的翻转，如果找到一个模型，则结束，否则开始下一次尝试。

执行局部搜索过程时，如果碰到局部极值点，很可能会有一些变量的值不断地翻过来翻过去，这种现象称为“颠簸”（Threshing）。为了避免这种现象，Mazure 等人[30]提出了基于 GSAT 的 TSAT 过程，它采用一个先进先出的“禁忌队列”（tabu list），将最近翻转过的变量按照先后顺序记录下来。在搜索过程的每一步，如果按照贪心策略所选出的变量已经在队列中了，那就不翻转它，而是按照贪心策略翻转下一个不在队列中的变量。这样可以在一定程度上减少翻转。

对 GSAT 过程的另一种改进是，在其中加上“随机游走”（Random Walk）[29][31]。所谓的随机游走是指，从不被当前赋值满足的所有子句中，随机地选择一个子句 $C$，从 $C$ 中随机地或者是根据贪心策略选择一个变量 $q$，然后将其值取反。这些随机性，使得搜索有可能是朝着不是使目标函数值变小的邻接点移动，有助于逃出局部极值点。

### 2.3.3 拟物拟人算法

我国的李末、黄文奇等人[32][33]采用拟物拟人的思想，实现了一个高效的

SAT 程序 Solar，该程序在 1996 北京举办的 SAT 程序竞赛中获得了第一名。所谓拟物，就是观察物理世界，寻找与 SAT 问题等价的物质运动形式，从中找出解决问题的有效办法。李未、黄文奇等人发现并且证明了：SAT 问题等同于带电质子在静电场中的势函数是否为零的问题，判定一个合取范式是否可满足等价于判定一个带电质点在相应的静电场中是否有使其势函数为零的位置。由于带电质点在静电场中总是沿着使其势能下降最快的方向，也就是沿其势函数梯度指引的方向运动，并最终达到势能最低的位置。因此，对一个客观上可满足的合取范式所对应的势函数，使用梯度算法是求使该合取范式为真的解释的高效算法。后来，黄文奇和金人超发现单纯依靠拟物方法容易落入局部最小值陷阱的困难境地，又提出了将拟物方法与拟人方法结合的思想。所谓拟人方法就是把人类的社会经验形式化为算法用以求解某些特殊困难的数学问题的方法。

## 2.4 完备算法和不完备算法的混合算法

张键和张浩瀚[34]提出了一种将回溯法和不完备方法结合起来的方案。它将问题求解过程分成两个阶段，前一阶段，选取一部分变量，采用不完备的方法得到一个部分赋值；在后一阶段，采用回溯法将这个部分赋值扩展为一个完整的解。

Mazure 等[30]用另外一种方式将 DPLL 算法和局部搜索过程 TSAT 结合起来。该方法的主要框架为 DPLL，而 TSAT 用于分支变量的选择，其指导思想是：不可满足的子句集合 $S$ 往往有个不可满足的真子集 $S' \subset S$，在局部搜索过程中取假次数最多的子句很可能属于 $S'$，因此，该算法先对 $S$ 施以单子句规则，然后调用 TSAT 过程，选择那些在不满足的子句中出现次数最多的变量作为分支变量，得到两个新的子句集，并重复上述过程。

## 2.5 其它方法

除了上述的两大类和其结合算法外，人们还想出了各种各样的其他算法，这些算法很多是因为受到其他领域中的算法的启发而设计出来的。这些算法包括基于蒙特卡罗方法的随机算法、将 SAT 问题转化为 0-1 整数规划问题[35]或者多项式优化问题[36]来解决等等。

# 第三章 基于 DPLL 算法的 SAT 程序

## 3.1 概述

伴随着 DPLL 算法及其变种的研究的发展，人们研究与开发基于 DPLL 算法的 SAT 程序已经有将近 40 年的历史了。而最近 10 多年来，在这类 SAT 程序上的研发取得了较大的进展。20 世纪 90 年代中期的这类 SAT 程序中突出的包括：Tableau[37]、POSIT[18]、CSAT[38]等，接着，Silva 和 Sakallah [39]以及 Bayardo 和 Schrag [5]等将智能回溯、冲突驱动学习（Conflict-Driven Learning）等技术结合到 DPLL 算法中。这些技术大大提高了 DPLL 算法解决来自现实应用的结构化的 SAT 问题的效率和能力。SAT 程序在大量的生产、生活中得到应用，这反过来又推动了对更为高效的 SAT 程序的研究，从而导致了包括 SATO、Chaff 等新一代更高效的判定程序的诞生，这些新程序强调对 DPLL 类型算法的各个方面的改进和优化。

一个 SAT 程序是一个相对较小的计算机软件。上面提到的程序中，许多都只有数千行的源代码（由于性能上的考虑，这些程序大多数都是使用 C 或者 C++ 程序设计语言来编写的）。然而，编写这些程序决不是一件简单的工作，因为其中涉及的算法相当复杂，并且要在程序的各方面的细节上花上许多的心思，这些方面主要包括：编码、数据结构的设计、算法策略的选择、可调参数的设置等。高效、健壮的 SAT 程序的研究与开发，仍然是值得我们努力的一大方向。

## 3.2 SAT 程序的实现

### 3.2.1 数据结构

SAT 程序中常用的数据结构包括邻接链表（adjacency list）、赋值矩阵、懒惰数据结构（lazy data structure）等。

#### 3.2.1.1 邻接链表

一个子句，如果包含变量 $x$，则称该子句与变量 $x$ 相“邻接”。在本方案中，每个变量 $x$ 设置了一个链表，其中放置的是指向与 $x$ 相邻接的子句的所有指针，

这个链表称为邻接链表。此外，本方案中，子句用存放文字的链表来表示。

邻接链表也有许多变种。考察各种不同的邻接链表的变种时，要着重考察它们是否能够准确、高效地识别出子句的状态变化（满足，不满足或者单子句）。

（1）隐藏已赋值文字

从子句的文字链表中，把那些已经赋值的文字“隐藏”起来。在这种实现中，每个子句除了自由文字链表外，还有另外 2 个链表，分别是取真的文字链表和取假的文字链表；当一个文字被赋值为真时，从自由文字链表“摘”掉，加入到取真文字链表中，当一个文字被赋值为假时，从自由文字链表中“摘”掉，加入到取假文字链表中。

这种方案中，当产生单子句时，很容易确定要被赋值的文字，而不必为了找到这个文字而对子句进行搜索。但是，实验结构表明，这种方式组织的邻接链表数据结构，在性能上无法跟其他方案相比较。

（2）基于计数器的方案

另一种监察子句的状态变化的方法是为每个子句设置两个文字计数器。这两个文字计数器记录了一个子句中有多少个文字是为真，有多少个文字为假，当然也就间接地记录了自由文字的个数。通过考察这些计数器的值，就可以知道子句当前是否变成单子句、满足或者不满足了。

这种方案中，当一个子句变为单子句时，必须遍历该子句对应的文字链表来找出唯一的自由文字。使用基于计数器的邻接链表的典型例子是 GRASP。

邻接链表的一个主要的缺陷是，每个变量对应的邻接链表可能会很大，并且会随着搜索过程中新子句的添加而越变越大。因此，每当一个变量被赋值，就潜在着要遍历一个很大的子句链表的可能性（考察含有这个变量的子句的状态的变化）。可以设计不同的方法来克服这些缺陷。例如，每当一个子句$w$的状态变成满足，则所有与之邻接的变量的邻接链表中的$w$都被“隐藏”起来。这样，通过把已满足子句“隐藏”起来，则当一个变量 $x$ 被赋值时，只需要分析那些与 $x$ “邻接”的未定值子句。

#### 3.2.1.2　赋值矩阵

也可以使用稀疏矩阵来存放子句集。矩阵的每行表示一个子句，每列给出一个变量在子句集中的所有出现。在算法执行的过程中，子句有以下四种状态：

（1）被满足（子句中有一个文字变为真）；

（2）为假（子句中所有文字都变为假）；

（3）长度缩短（子句中有一部分文字变为假）；

（4）未受影响。

在后两种情况下，称子句是活跃的。对于整个子句集，统计每种子句的个

数，如果所有子句都被满足，公式就是可满足的。对每个变量 $p$，程序记录 $p$ 和 $\neg p$ 出现在哪些单子句中。如果 $p$ 出现在某个活跃的单子句中，而 $\neg p$ 也出现在另一个活跃的单子句中，那么子句集就是不可满足的。除了考察变量外，这种方案还统计每个文字出现在活跃子句中的次数，如果是 0，其补文字就是一个纯文字。

#### 3.2.1.3 "懒惰"数据结构

正如前面提到的，各种基于邻接链表的数据结构共同存在的问题是：每一个变量 $x$ 都必须维护数量可能很多的邻接子句的链表，并且这些链表的大小还可能在搜索的过程中不断地变大。显然，这将增加对 $x$ 进行赋值时的工作量。而事实上，当 $x$ 被赋值时，大多数含有 $x$ 的子句都不必处理，因为在大多数情况下，它们的状态都不会改变。

所谓懒惰数据结构中"懒惰"是推迟实现的意思。即给一个变量赋值时，并不一定要马上处理含有该变量的所有子句，而是当这些子句是有可能变成单子句的时候才加以处理。懒惰数据结构的典型例子是 SATO 的 Head/Tail Lists（头尾文字链表）。

### 3.2.2 开发语言和平台

出于性能的考虑，SAT 程序的开发一般使用高效的程序开发语言，常用的是 C 和 C++。开发平台一般是 Unix 或者 Linux；也有用 Windows 的。
至于使用的硬件，由于 SAT 是 NP 完全问题，SAT 程序的开发和运行环境，在过去的历史上，并不是使用一些最高端的高能计算机，各国研究者使用得最多的是单处理器的 SUN SPARCStation 系列的工作站。瑞典人 Lars-Henrik Eriksson 在 2000 年的时候做过一个对 GSVT、HeerHugo、NP-Tools 和 SATO 四个程序的性能比较的评测报告，所使用的运行平台更是一个具有单个 Intel Pentium III 450MHz 的处理器和 384M 主存的 IBM PC 兼容机。

## 3.3 基于 DPLL 算法的三种重要的 SAT 程序

### 3.3.1 GRASP

GRASP（General seaRch Algorithm for Satisfiability Problem）由 J. P. Marques-Silva 等人提出并实现，其源代码和二进制可执行文件可以从 http://sat.inesc.pt/~jpms/grasp/下载。GRASP 集成了几种已经证明是强有效的搜

索剪枝技术。GRASP 的最突出的特性是它用来辅助智能回溯的强有力的冲突分析过程，具体包括 3 个方面的内容：（1）通过冲突分析，找出冲突的原因，使得可以实现智能回溯；（2）通过记录引起冲突的原因，能够在未来的搜索中识别并且避免相似冲突的发生；（3）跟踪导致冲突的“原因链”使得 GRASP 能够识别出那些要使公式满足所必需的变量赋值。

GRASP 用 C++程序设计语言来编写，在 Linux 环境下用 gcc 来编译。在大量 SAT 问题实例上的实验表明，GRASP 的性能超越了其之前的大部分 SAT 程序。

### 3.3.2 SATO

SATO（SAtisfiability Testing Optimized）是由美国 Iowa 大学计算机系的 Hantao Zhang 开发的。其源代码可以从 http://www.cs.uiowa.edu/~hzhang/sato.html 上下载。

SATO 也是一个基于 DPLL 算法的 SAT 程序。SATO 结合的两种能有效提高性能的技术是其所使用的分支决策策略和冲突分析技术。尽管这两种技术都并不是 SATO 所首创的，但是 H. Zhang 在实现 SATO 时所做的贡献主要是很好地把这两种技术在不削弱 SATO 中其他技术的前提下将它们集成起来。

在分支启发策略上，SATO 的做法是同时提供几种流行的高效的分支策略，每种这些策略都对某些种类的 SAT 问题特别有效。

SATO 使用了基于冲突分析的智能回溯技术，其思想其实非常简单：当出现冲突时，SATO 找出所有与这个冲突的产生有关系的决策变量所对应的文字，并把这些文字保存起来。如果当前的决策变量的文字是这些文字之一，则 SATO 不再尝试搜索其另外一个分支，进行回溯。而为了避免在以后的搜索中重复收集同样的文字，SATO 保存由这些文字的补文字的析取所构成的一个新的子句到原来的子句集中。这跟 Silva 和 Sakllah 在 GRASP 中实现的子句记录基本一致。

为了加速 BCP 过程，SATO 使用了懒惰数据结构 Head/Tail literals，这将在第五章中介绍。

### 3.3.3 Chaff

Chaff 是 M.W.Moskewics 等人提出的一个高效的 SAT 程序[7]，当用来解决困难的现实世界的 SAT 问题时，该算法在性能上超过了现存的大多数其他的公开 SAT 程序。

有两种版本的 Chaff，分别是 mChaff 和 zChaff，分别由 M. W. Moskewicz（mChaff）和 L. Zhang（zChaff）实现，两者的主要思想都是相同，只是在实现的一些细节上有所区别。由于 zChaff 曾在多次 SAT 竞赛上夺得第一名，所以我们在此主要介绍 zChaff。zChaff 的源代码可以从

http://ee.princeton.edu/~chaff/zchaff.php 上下载。

zChaff 的优化哲学是从实现细节上下功夫，通过细心、巧妙、优化的实现来获得较大的性能提高，其特别的改进在于:

（1）对 BCP 过程的特别高效的实现；

（2）提出了一种新颖的半动态的分支决策策略 VSIDS。说它是静态是因为它不依赖于变量的状态；说它具有动态性是因为其中部分计数器值会随着新子句的添加而更新并且所有变量的计数器值都会周期性地“老化”（使得与最近的冲突相关的变量具有更高的优先级）。

其具体做法如下：

（1）对应每个变量的每种文字形式（正、负两种形式）都设置一个计数器，其初始值为该文字在原公式的子句中出现的次数；

（2）当一个新的冲突导出子句被添加到子句数据库中时，该子句中文字对应的计数器值增加一定的量（这个量时可以调节的）；

（3）那些具有最大的一个计数器值的自由变量，被选择为分支变量，对应的赋值为使该变量的具有最大计数器值的文字为真的赋值；

（4）在遇到各值相等的时候，随机选择其中一个作为分支变量（随机化）；

（5）定期地，将各计数器值除以一个常量（这也是可以调节的，Chaff 常用的值是 2）。

从整体上来看，在选择分支变量时，这种策略“优待”出现在冲突导出子句中的变量，特别是最近添加的冲突导出子句中的变量。由于难解的问题在求解的过程中会出现许多冲突，从而生成许多冲突导出子句，这些冲突导出子句在文字数目上占了很大的比例，所以 VSIDS 的突出优点是其对小的计数器值的“低波过滤”作用，结果就是重视了最近生成的冲突导出子句所含的信息。其作者的理由是他们相信冲突是难解的 SAT 问题的搜索过程继续进行的主要推动力，VSIDS 则是一种直接把这种推动力结合到决策过程中的一种方案。

VSIDS 的另外一个特性是：由于它是变量状态独立的，所以它具有较小的计算开销，因为计数器值只有在冲突发生的时候才需要被更新。

在实现 VSIDS 的时候，为了在决策的时候更快地选择出具有最高计数器值的变量，在 BCP 和冲突分析的过程中，zChaff 维护一个按照计数器值的大小排序的自由变量的链表（使用 C++的 STL 中的 set 来实现）。

实验证明，在不少的测试问题上，zChaff 比起其他的 SAT 程序，包括 GRASP 和 SATO，在运行性能上有了高达两个数量级的提高。

# 第四章 一个 SAT 程序的通用框架 3D-DPLL

在参考前人的研究成果和实践经验的前提下，结合我自己的想法，我提出了一个 SAT 程序的通用框架 3D-DPLL。

借助这样一个通用框架，我们可以：

（1）快速生成 SAT 程序；

（2）对其中各个模块的不同实现就得到了不同的算法，通过实现各种不同的算法和技术，该框架提供了一个在同等条件下比较各种算法和技术的测试台。

## 4.1 框架的组成和结构

这是一个循环形式的 DPLL 算法框架，由于其中的 3 个核心部分分别为决策引擎（Decision Engine）、推理引擎（Deduction Engine）和诊断引擎（Diagnosis Engine），其英文名称都是以字母 D 开头，因此我将其称为 3D-DPLL 框架。原始的 DPLL 算法流程采用递归的形式，但是考虑到递归的开销，本算法框架采用循环形式来实现 DPLL 算法。除了上述 3 个核心引擎之外，还有一个预处理模块。3D-DPLL 框架的伪码表示如下：

```
status = preprocess ( );
if ( status != UNKNOWN ) return status;
while (1) {
    decide ( );
    while (1) {
      status = deduce ( );
      if ( status = = CONFLICT ) {
        blevel = diagnose ( );
        if (blevel = = 0)
            return UNSATISFIABLE;
        else backtrack ( blevel );
      }else if ( status = = SATISFIABLE )
        return SATISFIABLE;
      else break;
    }
}
```

其中 status 存放当前子句数据库的状态，blevel 存放的是回溯的目的决策层次。其流程图如下：

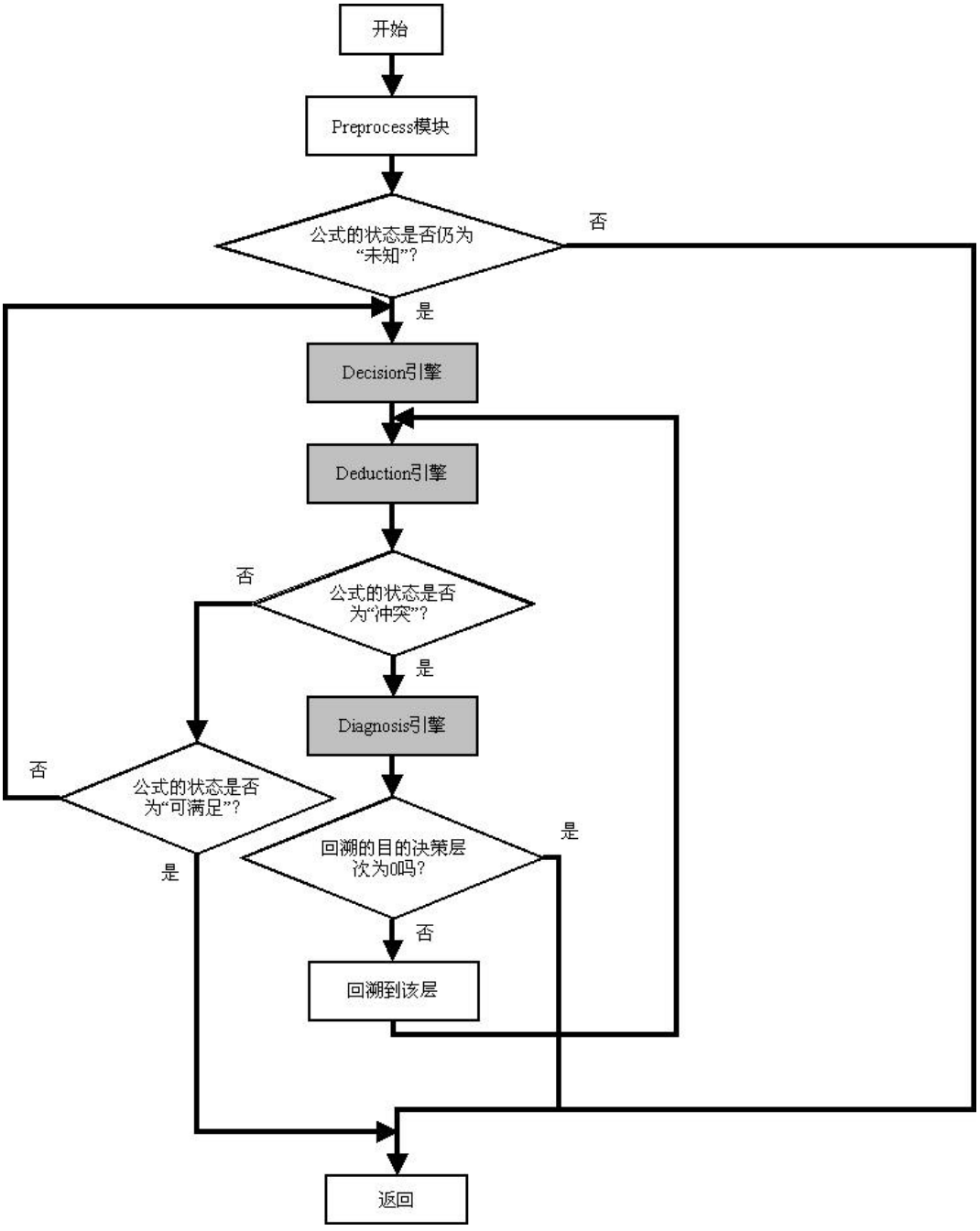



图 4-1 3D-DPLL 框架

Fig 4-1 The 3D-DPLL Framework

## 4.2 预处理模块（Preprocess Module）

预处理的目的是在正式的搜索开始之前，先对输入的公式进行化简以达到

加速搜索过程的目的。通常，一个 SAT 程序的预处理过程只是在开始搜索之前执行一些额外的推理机制。具体执行一些什么工作是可以灵活处理的。例如在我所实现的 QuickSAT 中，执行的预处理工作包括：

（1）找出所有定义了，但是实际上没有使用的变量。具体做法是通过检查变量数组_variables 里的每一个元素，如果该变量对应的两种文字的个数都是 0，则为无用变量。对于这样的变量，我将它们放入 un_used 数组中，同时令其值为 1，其决策层次为-1。最后，将自由变量的数目减去这些无用变量的数目；

（2）找出所有纯文字。将这些纯文字分别按照正的纯文字和负的纯文字入栈，并且把它们作为一个（不存在的）导出子句放入到导出子句队列_implication_queue 中，这就可以在以后的 BCP 中统一处理；

（3）找出所有的单子句，并且把这些单子句放入_implication_queue 中。

（4）调用一次 deduce ( )，即执行第一次 BCP 过程，然后返回。

由于预处理过程在整个计算过程中，只会执行一次，因此通常可以在预处理模块中运行一些开销比较大的推理操作，因为这些操作如果放在推理引擎中，则由于搜索树上的每个结点上都执行一次，会导致极大的开销。预处理模块其实还可以放在整个判定程序之外，即先用它对输入公式进行处理。

## 4.3 决策引擎（Decision Engine）

决策引擎执行分支决策过程。决策过程通过选择一个自由变量及其赋值来扩展当前的部分解释。决策过程是探索搜索空间的新区域的基本机制。

在搜索算法进行的过程中，当 BCP 不再能够进行的时候，决策引擎就要负责从所有自由变量中选出一个分支变量及其相应的赋值。选择一个好的分支变量的重要性是众所周知的——同样的搜索算法、同样的问题，不同的分支策略本身就能造成搜索树大小上的极大的差异，从而大大地影响着程序的效率。多年来，大量的不同的分支启发式策略被提了出来并且被实现，人们也对这各种各样的分支策略进行了大量的实验比较[19][40]。

早期的分支策略，例如 Bohm 启发策略[17]、MOM、J-W 等策略可以看成一种贪心策略，其指导思想是要使到在分支之后能够生成最多的单子句或者使得最多的子句被满足。所有这些策略都借助于某种估价函数来对各个自由变量进行评估，并且选择使这个函数获得最大值的变量作为分支变量。这种类型的启发策略对于特定类型的 SAT 问题非常有效，然而，所有这些评估函数都是基于问题的子句数据库的一些统计数据（例如，子句长度等）上的，这些统计数据，虽然对于随机类的 SAT 问题是非常有用的，但是，往往无法捕捉到结构化问题中的“相关性”信息。

有人提出使用基于文字计数的分支策略[19]。基于文字计数的分支策略记录包含一个命题变量的未定值子句的个数。同时，人们发现，基于两种可能形式的文字的计数的 DLIS 策略选择命题变量也有较高的性能。注意，这个计数值是变量状态相关的，因为不同的变量赋值会得到不同的计数值，其原因是一个子句是否未定值，取决于当前的变量赋值。正由于这个计数值是变量状态相关的，所以，每次计算分支变量的时候，都必须重新计算自由变量的计数值。

随着 SAT 程序变得越来越高效，计算分支变量的工作的开销在整个运算时间中所占的比率显得越来越大。因此，人们需要开发更加高效的分支策略。

在我的 3D-DPLL 框架中的 decide ( )其实是一个外包函数，里面可以根据需要调用 decision_zChaff ( ), decision_SATO ( ), decision_GRASP( )等，这些函数分别对应 zChaff、SATO 和 GRASP 程序中所使用的决策策略，而这些具体的函数可以独立实现。

## 4.4 推理引擎（Deduction Engine）

根据到目前为止的赋值集合进行逻辑推理（利用单子句规则、纯文字规则等）来扩展当前的部分解释。由于实践表明，纯文字规则不一定能够提高算法的实际效率，因此，大多数算法实际上都不使用纯文字规则，因此推理一般主要只使用单子句规则，故推理引擎主要执行 BCP 操作。由这个过程得到的赋值称为导出赋值；另外，推理引擎也可能导致空子句，这样得到的空子句代表了一个冲突。

推理引擎通过“前瞻”（look ahead）动作来起到对搜索空间剪枝的目的。当一个分支变量被赋予一个值时，整个子句数据库在 deduce( )中被化简。函数 deduce( )要根据上一次的决策过程的结果来向着使公式满足的方向进行推理，并且可能返回三种不同的状态：如果在当前的变量赋值之下，公式是可满足的，则返回的状态为 SATISFIABLE，如果公式含有一个空子句，则返回状态为 CONFLICT，如果不是上述两种状态，则返回 UNKNOWN，程序会继续进行下一次决策过程。不同的推理机制具有不同的推理能力和运行时间开销。只要推理过程所使用的推理规则是正确的（即不会在公式产生了一个空子句的时候却返回 SATISFIABLE 等矛盾的情况），则具体如何实现推理过程都不会影响整个算法的正确性，然而，不同的推理机制，甚至同样的推理机制的不同的实现，都会在很大程度上影响着程序的运行性能。

一直以来，人们提出了几种不同的推理机制。然而，在这些不同的推理机制中，单子句规则是最高效的一种，其原因是单子句规则能够在需要相对少的计算开销的前提下剪枝掉较大的搜索空间。几乎所有现代的 SAT 程序都集成了

单子句规则。在一个 SAT 程序中，BCP 过程的运行通常占用了整个运行过程的大部分时间。因此，SAT 程序的效率与其 BCP 过程（也就是推理引擎）的性能有着直接的关系。

除了单子句规则之外的其他规则，虽然都被证明了能够有效地解决一定种类的 SAT 问题，但是比起单子句规则，这些规则都无法在解决一般的 SAT 问题时不降低程序运行的整体性能。

其它机制中最有名的是纯文字规则。纯文字规则的问题是在解决问题的过程中找出纯文字的过程本身就有很大的开销。研究者的共识是：使用纯文字规则在一定程度上能加速搜索过程之外，其本身带来的开销也非常大，总的效果是，对于大多数的测试实例，引入了纯文字规则反而会使得程序变慢。

其它的推理机制包括所谓的等价性推理（Equivalence Reasoning）[41]、递归学习（Recursive Learning）[42][43]等，由于不是我研究的重点，在此就不再讨论了。

## 4.5 诊断引擎（Diagnosis Engine）

诊断引擎中执行的是冲突分析的工作。这也是一个外包函数，其中可以调用实现各种不同的冲突分析的函数。

诊断引擎用于在搜索遇到冲突的时候找出冲突的原因和解决冲突的方法。它告知 SAT 程序在一定的搜索空间里不存在原 SAT 问题的解，然后指出一个继续进行搜索的新的区间。

人们已经把诊断引擎设计得可以很好地适应包括单子句规则和其它规则在内的各种推理机制。然而，由于单子句规则通常是大多数现代 SAT 程序中集成的唯一的推理机制，因此大多数的诊断过程都是面向单子句规则的。在这样的 SAT 程序中，如果一个变量的赋值由一个单子句所蕴涵，则这个单子句对应的子句称为这个变量赋值的前件子句。每个导出赋值都有一个前件子句，而决策变量都没有前件子句。

## 4.6 回溯

函数 backtrack ( )可以根据需要来实现简单回溯或者智能回溯。如果上面的诊断引擎不进行冲突分析，则返回的 blevel 只可能是当前决策层次减去 1。这样进行的就是简单回溯。

回溯的主要工作是子句集状态的恢复，必须恢复到回溯到的目的决策层次上原来的状态。

例如，在我实现的 SAT 程序 QuickSAT 中的回溯函数里所作的工作就包括：

（1）确认 blevel（回溯到的决策层次）必须小于等于_dlevel（当前的决策层次）；

（2）按照赋值栈中保存的赋值序列一层层地往回取消相关变量的赋值；

（3）然后，将_dlevel 设为 blevel；

（4）将_stats.num_backtracks 增 1；

（5）执行其他的相关恢复工作。

## 4.7 3D-DPLL 框架的使用

### 4.7.1 3D-DPLL 作为 SAT 程序开发的快速原型

3D-DPLL 是一个模块化的 SAT 程序框架。它是一个“可定制”原型，即可以通过对各个模块的不同的实现来得到不同的程序。利用这个框架，可以实现各种基于 DPLL 的算法，因为各种基于 DPLL 的算法都具有这些通用的结构，也就是说我所提出的 3D-DPLL 算法已经抽象了基于 DPLL 的算法的公共特征了。

我所开发的 QuickSAT 就是在这个通用框架上开发出来的。实践表明，3D-DPLL 通用框架的提出，能有效加快 SAT 程序的开发，可以作为基于 DPLL 算法的 SAT 判定程序开发的快速原型。

### 4.7.2 3D-DPLL 作为算法比较的测试台

正如前面所提到的，3D-DPLL 是一个“可定制”的程序框架，可以通过对各个模块的不同实现来得到不同的程序。例如，如果决策模块中实现的是 GRASP 的决策策略，推理引擎中实现的是 GRASP 的 BCP 过程，诊断引擎中实现的是 GRASP 的冲突分析策略，则我们得到的就是程序 GRASP。

可以利用 3D-DPLL 作为原型，实现了各种不同的 SAT 算法来进行实验比较，也可以在保持其他部分相同的情况下，对框架的某个部分进行不同的实现，这样就可以单独地对 SAT 算法中的某个技术进行比较了。

通过这个框架所实现的各种算法，可以看成是同等条件下实现的不同算法，对这样的不同的算法的比较，可以得到比较中肯的结果。

这个框架除了可以作为实现 SAT 程序的快速原型以及比较各种技术的测试台外，还可以与重启动策略很好地结合起来，具体的做法是：设置多种策略（多种决策策略、多种推理策略、多种诊断策略、多种预处理策略），再每次重启动之后，使用另外的一系列策略。

# 第五章 QuickSAT 的设计和实现

## 5.1 概述

QuickSAT 是我设计并实现的一个 SAT 程序，从设计之初就立足于提高其效率和健壮性，并且将这一指导思想贯彻到 QuickSAT 的整个设计和实现的过程中。QuickSAT 是基于前面提出的 3D-DPLL 通用框架而构建的。

QuickSAT 并不是一切从头开始的，它借用了许多前人的被证明了能有效提高程序的运行性能的技术，并将它们成功地集成在一起。QuickSAT 从 GRASP 那里继承了冲突分析和智能回溯，从 SATO 继承并改进了快速 BCP，从 Chaff 继承了分支决策过程中冲突导出子句“老化”的思想。此外，QuickSAT 还使用了重启动策略。

另一方面，QuickSAT 引入了如下的新技术和策略，主要是在 BCP 过程、分支决策策略和子句数据库管理这三个方面进行了改进：

1）在 BCP 过程的实现中，QuickSAT 受 SATO 的头尾链表（Head/Tail Lists）模式的启发，设计了同样基于“推迟实现”思想的称为“2-文字监视模式”的子句状态变化监控方案；

2）保存的冲突导出子句被组织成一个反映其生成的时间先后顺序的栈（栈顶的子句是最近生成的，栈底的子句是最久之前生成的）。如果在当前时刻栈中有未满足的冲突导出子句，则下一个分支变量就在出现在栈顶的未满足的冲突导出子句的自由变量中选择；

3）在选择分支变量的决策策略中，QuickSAT 在计算变量引发冲突的“活跃性”时采取了与 Chaff 不同的策略。对于变量 $x$，Chaff 把它在冲突导出子句中出现的次数作为其“活跃性”。但是，我发现 Chaff 的这种做法忽略了那些没有出现在冲突导出子句中但对冲突的发生却有很大关系的变量（某些非决策变量就是这样的一个例子）。为了弥补这种不足，QuickSAT 考虑更广泛的与冲突相关的子句来作为选择决策变量的依据。

4）当选定了分支变量后，对分支变量先进行的赋值的选择，也就是对先检查一个分支结点的哪一个子树的选择，QuickSAT 也引进了一种新的启发式策略；

5）QuickSAT 在当前的搜索树被舍弃之后（即重启动之后）使用新的子句数据库的管理策略。这种策略的创新在于决定一个子句是否要从子句数据库中删掉的依据不仅仅是它的长度，同时也考虑这个子句导致冲突的“活跃性”和

它存在于子句数据库中时间的长短。

实验表明，QuickSAT 跟 GRASP、SATO 和 Chaff 比较起来具有更好的健壮性，也就是说在规定的合理的时间内，QuickSAT 能解决比 Chaff 和 SATO 更多的问题。在效率方面，跟当今世界上最快的 SAT 程序 Chaff 也是在伯仲之间，不相上下。

## 5.2 快速 BCP 过程

实验分析表明，一个 SAT 程序在解决 SAT 问题的时候，其中花在 BCP 过程中的时间大约占了总运行时间的 90%，所以 BCP 过程是一个 SAT 程序中的关键部分。而在执行 BCP 的过程中，如果按照传统的方法，当一个变量获得赋值时，逐个检查所有的子句，看哪个子句由于这个变量的赋值而变成了单子句，显然，在这个时刻并不可能每个子句都会变成单子句，所以，对所有子句进行搜索会造成计算的大量浪费。为了高效地实现 BCP 过程，要求寻求新的方案。

SATO 使用了一种称为头尾链表（Head/Tail Lists）的方案。在这种方案中，每个子句对应有两个特殊的指针，分别称为头指针和尾指针（注意区别于指向子句中第一个文字的指针和最后一个文字的指针），头指针指向的文字称为头文字，尾指针指向的文字称为尾文字，（要注意区别这里的头、尾文字与子句中的第一个文字和最后一个文字的区别，子句中的第一个文字和最后一个文字总是固定不变的，但是头、尾文字就是头、尾指针所指向的文字，当头尾指针移动了，头尾文字也就变化了）。子句的所有文字都存放在一个数组中，除了上述的两个特殊的头尾指针外，每个子句用指向第一个和最后一个文字的指针来标识。开始的时候，头指针指向子句的第一个文字，尾指针指向子句中的最后一个文字（只有在这个时候，头尾文字才跟子句的第一个文字和最后一个文字是同样的东西）。

此外，每个变量都对应 4 个链表，链表中的存放的是指向含有该变量对应的文字做为头或尾文字的子句的指针，这 4 个链表分别是：

$clauses\_of\_pos\_head(v)$：存放指向头文字为 $v$ 的子句的指针；

$clauses\_of\_neg\_head(v)$：存放指向头文字为 $\neg v$ 的子句的指针；

$clauses\_of\_pos\_tail(v)$：存放指向尾文字为 $v$ 的子句的指针；

$clauses\_of\_neg\_tail(v)$：存放指向尾文字尾 $\neg v$ 的子句的指针。

当变量 $v$ 赋值为 1 的时候，不必理会链表 $clauses\_of\_pos\_head(v)$ 和

$clauses_of_pos_tail(v)$中的子句，只需要处理在$clauses_of_neg_head(v)$和$clauses_of_neg_tail(v)$中的子句。对于每个在$clauses_of_neg_head(v)$中的子句$C$，SATO 会从$C$的头文字开始到向尾文字的方向搜索一个值不为 1 的文字。由于这样的子句的头文字是$\neg v$，因此，可能出现如下 4 种情况：

1）如果在搜索的过程中，第一个遇到的文字是一个值为 1 的文字，则子句已经满足，不必做任何工作；

2）如果在搜索的过程中，首先遇到一个自由文字$l$，且$l$不是尾文字，则将子句$C$从链表$clauses_of_neg_head(v)$中“拿掉”，加到文字$l$对应的变量的头链表中（假如文字$l$是一个正文字，对应的变量就是$l$，则加到$clauses_of_pos_head(l)$中，假如文字$l$是一个负文字且$l=\neg x$，则子句$C$被添加到$clauses_of_neg_head(x)$中），然后使头指针从原来的位置指向新的头文字$l$，这个操作称为移动头文字。

3）如果在头尾指针之间的所有文字都为 0，但是尾文字是一个自由文字，则该子句成为单子句；

4）如果在头尾指针之间的所有文字都为 0，并且尾文字也是 0，则该子句是一个冲突子句。

对于$clauses_of_neg_tail(v)$中子句的操作也是相似的，只不过是从尾文字开始往头文字方向搜索。

这种 SATO 的头尾链表的方法比传统的基于计数器的方法要高效，因为当变量被赋值时，并不需要搜索所有的子句：当变量被赋值为 1 时，以该变量对应的正文字为头或尾文字的那些子句都不需要搜索了，同样，当变量被赋值为 0 的时候，以该变量对应的负文字为头或尾文字的那些子句就都不必搜索了。另外，由于每个子句都只有总共两个头尾指针，所以一个变量被赋值时，如果假设头尾文字的正负分布是均匀的话（即所有正的头文字和尾文字的数目之和大约等于负的头文字和尾文字的数目之和），则平均只有$m/n$（$m$为子句总数，$n$为变量总数）个子句的状态需要被修改，尽管在每次状态更新的时候，需要做的工作与基于计数器的方法不同，但是，实验表明，总体来说，基于头尾文字的机制具有较快的速度。

可以说，SATO 的头尾文字链表的方案已经在很大程度上提高了性能，减少了不必要的搜索。但是，我们发现这种方案中还是存在一些不必要的开销，仍然可以改进，因此提出了一种我称之为“2-文字监视模式”的方案，这种方案是基于如下的观察的：

**对于一个具有 N 个文字的子句，仅当其中的 N-1 个文字的值都为 0 时，才是**

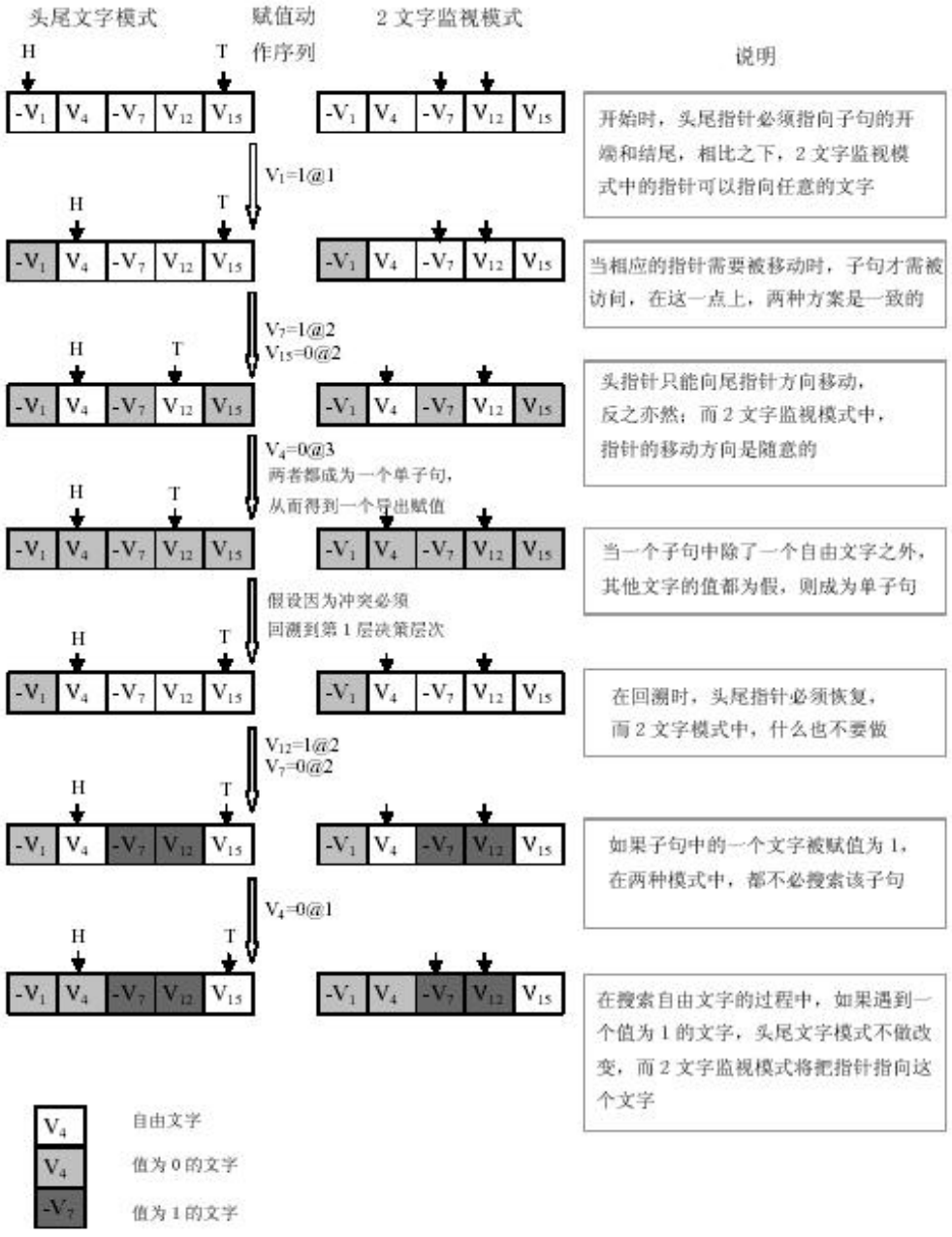



**图 5-1 2-文字监视模式与头尾文字模式的比较**

Fig 5-1 The Comparison between 2-Literal Watched Schema and Head/Tail Lists

**一个单子句，也可以说，仅当其中 N-2 个文字的值都已经为 0 时，该子句才有可能变为一个单子句。因此，我们完全没有必要在这个子句中有 1， 2， 3，……，或 N-1 个文字为 0 时都访问这个子句一次。我们只需要在子句的假文字的个数从 N-2 变成 N-1 的时候访问这个子句。按照这个思路，我们可以节省大量的运**

算时间。

在“2-文字监视模式”中，在任意给定的时刻，从每个子句中任意选出两个值不为 0 的文字来作为监视的对象，这样，只要这两个文字都没有被赋值为 0，则该子句中不可能含有多于 N-2 个值为 0 的文字，从而我们可以保证只有当这两个文字中之一被赋值为 0 的时候，该子句才有可能成为单子句。现在，我们只需要当子句中的这两个被监视的文字中的一个的值变为 0 的时候才搜索这个子句。当我们搜索这样一个子句时，会遇到以下两种可能情况之一：

1）该子句并非单子句，因此该子句中至少有两个文字的值不为 0，包括另外一个被监视的文字在内。这意味这至少有一个被监视文字之外的文字没有被赋为 0 值。我们可以选择这样的一个文字去代替两个被监视的文字中刚刚被赋值为 0 的那一个，成为一个新的被监视文字。这样就维持了两个被监视文字都是不为 0 的要求。

2）该子句变为单子句，则很显然，该单子句中的唯一的自由变量必定是另外一个被监视的文字。

跟 SATO 的头尾链表模式比较起来，“2-文字监视模式”的优点首先是指向被监视文字的指针可以随意移动，没有方向限制，亦即当被监视的一个文字被赋 0 值后，可以找子句中任何位置的自由变量来作为新的被监视文字，但是，在头尾链表模式中，必须按照一定的方向来找出新的头尾文字；其次是回溯的时候，不必恢复指向被监视文字的指针（即指针保持指向原来的位置），然而在 SATO 中，必须使头尾指针重新恢复到指向原来的文字。这就是使得在我这种方案中，取消对一个变量的赋值可以在常量时间内完成。另外，对一个最近被赋值后来又被取消赋值的变量进行再次赋值也会比对该变量进行第一次赋值的时候更快，这是因为回溯时不需要还原指向被监视文字的指针，所以被监察的文字都是最近被访问过的文字，这就能够大大地降低总的内存访问量，而在 SAT 程序的实现中，影响性能的一个很大瓶颈就是由于大量的高速缓存访问的不命中导致的内存访问。“2-文字监察模式”跟 SATO 的头尾文字模式的比较见图 5-1。

在 QuickSAT 中，“2-文字监视模式”的实现是通过为每个变量设立两个分别存放指向被监视的对应正、负文字的指针的数组，其示意图如图 5-2。

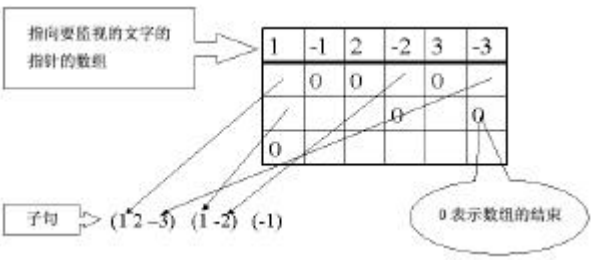



图 5-2 2-文字监视模式的实现

Fig 5-2 The Implementation of 2-Literal Watched Schema

注意，从图 5-2 中，我们还可以看出，2-文字监视模式不监视单子句。

## 5.3 冲突分析和智能回溯

在运行 BCP 过程中出现了一个冲突的时候，通过对收敛于一个冲突结点 $K$ 的蕴涵图的结构进行分析来找出那些对这次冲突负有直接责任的变量赋值——这些变量赋值的合取就构成一个代表了引起这个冲突的充分条件的蕴涵项。这个蕴涵项的非就是一个并不存在于原来的子句数据库中的但却是原 CNF 的一个逻辑推论的新子句。这个新子句，我们称为冲突导出子句。冲突导出子句是冲突分析的主要结果，是智能回溯的必要工具。

我们用 $A^{w_C}(K)$ 来表示冲突结点 $K$ 的冲突前提赋值集合，其相应的冲突导出子句用 $w_C(K)$ 来表示。赋值集合 $A^{w_C}(K)$ 是通过从冲突结点 $K$ 为起点逆向访问蕴涵图而得（逆向 BCP 过程）。除了可以包含当前决策层次的决策变量之外，$A^{w_C}(K)$ 中包含的变量只能是那些决策层次低于当前决策层次的变量，其中的原因是：在当前决策层次的决策变量对当前决策层次的所有其他赋值负有直接责任，所有其他赋值都是该决策赋值的推论，所以只需要把当前决策层次的决策变量包括在内即可。这样，当前决策层次的决策赋值与具有较低决策层次的变量赋值一起构成了当前冲突的一个充分条件。

为了计算 $A^{w_C}(K)$，先定义变量 $x$ 结点的前提赋值集合（记为 $A(x)$）的一种划分：（注意：下面的 $x$ 可以代表 $K$ 或者在当前决策层次赋值的任意变量）：

$$\begin{aligned}\Lambda(x) &= \{(y, v(y)) \in A(x) \mid dl(y) < dl(x)\} \\ \Sigma(x) &= \{(y, v(y)) \in A(x) \mid dl(y) = dl(x)\}\end{aligned} \qquad (5.1)$$

例如，如图 5-3 中，$\Lambda(x_6) = \{x_{11} = 0@3\}$，$\Sigma(x_6) = \{x_4 = 1@6\}$。显然有：

$$\Lambda(x) \cup \Sigma(x) = A(x) \qquad (5.2)$$

现在冲突赋值集合 $A^{w_C}(K)$ 可以表示为：

$$A^{w_C}(K) = causes\_of(K) \qquad (5.3)$$

其中，$causes\_of(.)$ 的定义如下（递归定义）：

$$causes\_of(x) = \begin{cases} (x, v(x)), \text{如果} A(x) \text{即为决策变量} \\ \Lambda(x) \cup \left[ \bigcup_{(y, v(y)) \in \Sigma(x)} causes\_of(y) \right], \text{否则} \end{cases} \qquad (5.4)$$

则由 $A^{w_C}(K)$ 生成的冲突导出子句为：

$$w_C(K) = \bigvee_{(x,v(x))\in A^{wC}(K)} x^{v(x)} \qquad (5.5)$$

其中，对于变量 $x$，$x^0 \equiv x, x^1 \equiv \neg x$。

对于图 5-3 中的例子，通过应用上述公式（5.1）-（5.5），我们可以得到在决策层次 6 上的冲突赋值和冲突导出子句：

$$A^{w_C}(K) = \{x_1 = 1@6, x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3\},$$

$$w_C(K) = (\neg x_1 \vee x_9 \vee x_{10} \vee x_{11}) \qquad (5.6)$$

生成一个冲突导出子句 $w_C(K)$ 使得我们可以进一步导出更多的新子句，这些新子句能够帮助我们对决策树进行剪枝。分析一个冲突导出子句 $w_C(K)$ 的结论包括：

（1）迫使当前的决策变量赋值取反。

如果 $w_C(K)$ 包含当前的决策变量，则取消了当前决策层次的所有赋值后，$w_C(K)$ 就会变成一个单子句，其中唯一未定值的变量就剩下当前的决策变量，根据单子句规则，马上就会迫使该决策变量的值取反。这样的一个过程，人们有时也称为 FDA（Failure-Driven Assertion）。我发现，这种情况能够被 QuickSAT 的

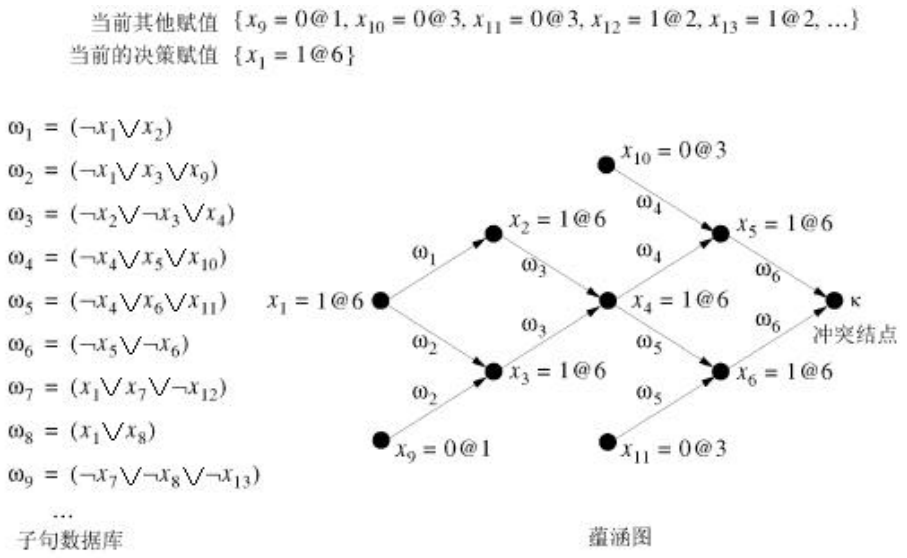



图 5-3 一个示例

Fig 5-3 An Example

基于 BCP 的 Deduction 引擎自动处理，而不必另行专门处理（即 Diagnose 引擎返回 SUCESS，把 FDA 直接交由下一次循环的 BCP 过程处理）。这跟其它大多数的基于 DPLL 的 SAT 程序的情况刚好相反，后者必须把当前决策变量的另外一个分支当作又一次的决策赋值。

在我们的例子中，当取消了决策层次 6 上的推理序列之后，上述的冲突导出子句 $w_C(K)=(\neg x_1 \vee x_9 \vee x_{10} \vee x_{11})$ 将成为一个只含自由文字 $\neg x_1$ 的单子句，这样根据单子句规则，马上就能够推出 $x_1=0$。

（2）冲突制导回溯（Conflict-Directed Backtracking）。

如果所有冲突导出子句 $w_C(K)$ 中的文字对应的变量都是在比当前决策层次低的决策层上获得的赋值，我们就马上可以推断搜索过程需要回溯。这种情况只可能发生在当前的冲突是前一次冲突的分析的直接结果的时候（例如，假设前一次冲突导致的 FDA 是 $x_1=0$，而这 $x_1=0$ 导致了当前的冲突）。对于我们的例子，这种情况如图 5-4a。

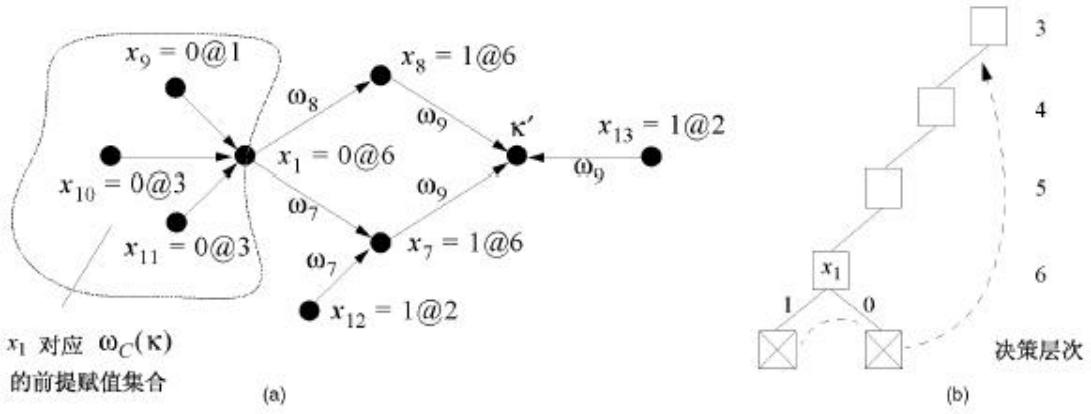



图 5-4 冲突制导回溯

Fig 5-4 Conflict-Directed Backtracking

假设在通过 FDA 断定 $x_1=0$ 之后，又导致了另外一个冲突 $K'$。与这个新的冲突相关的冲突赋值集合和冲突导出子句是：

$$A^{w_C}(K')=\{x_9=0@1, x_{10}=0@3, x_{11}=0@3, x_{12}=1@2, x_{13}=1@2\},$$

$$w_C(K')=\{x_9 \vee x_{10} \vee x_{11} \vee \neg x_{12} \vee \neg x_{13}\} \tag{5.7}$$

这就清楚地表明了，导致这第二次冲突的所有赋值都是在当前决策层次之

前获得的，因此除非搜索过程回溯到 $A^{w_C}(K')$ 中具有最大决策层次 $b$ 的变量所在的决策层次上，否则不可能找到令公式满足的解释。显然

$$b = \max\{dl(x) \mid (x, v(x)) \in A^{w_C}(K')\} \qquad (5.8)$$

如果 $b = d - 1$（这里，$d$ 是当前决策层次），则搜索过程简单地按时间先后顺序回溯到前一决策层次；如果 $b < d - 1$，则搜索过程将会智能地“回跳”过若干个决策层次：所有在决策层次 $b$ 之后获得的赋值都会令到最近被识别出来的冲突导出子句 $w_C(K') = \{x_9 \vee x_{10} \vee x_{11} \vee \neg x_{12} \vee \neg x_{13}\}$ 变得不可满足，因此一个常规的按照时间先后顺序回溯的策略，将会浪费大量的时间在一个无用的搜索区间里搜索，最后只会发现一切都是徒劳。相反，QuickSAT 的搜索引擎将会直接“回跳”到决策层次 $b$，回跳以后，在那个层次的决策结点，QuickSAT 将会根据 $w_C(K')$ 来得到一个 FDA 或者计算一个新的回溯层次。

在我们的例子中，在遇到了第二个冲突之后，通过将公式（5.8）应用到在（5.7）中的冲突前提赋值和冲突导出子句中，结果得到回跳的目的层次是 3。回跳到决策层次 3 的同时，在蕴涵图上添加一个对应于这个冲突的一个冲突结点；回跳之后对这个冲突的诊断导致了在决策层次 3 上的一次 FDA（见图 5-4b）。

我们可以直接利用 $w_C(K)$ 的这些推论，然后抛弃这个冲突导出子句，但是，把这个新子句保存下来，添加到子句数据库中，很可能能够在将来的搜索中产生更多无法由 $w_C(K)$ 推出的新约束（子句）。另外，将 $w_C(K)$ 添加到子句数据库中能够防止以后再次得到导致当前冲突的那些赋值。

## 5.4 在分支决策策略上的改进

QuickSAT 使用的分支决策策略是在 Chaff 的 VSIDS 的基础上提出的一种改进方案。跟 VSIDS 相似，决策策略“偏爱”那些最近在导致冲突上比较“活跃”的变量。这其中其实含有两个衡量尺度：

（1）变量的活跃性

在 VSIDS 中，一个变量的“活跃性”由与该变量对应的两种文字出现（在冲突导出子句、特别时最近生成的冲突导出子句中）的次数相关的一个计数值来衡量。与 VSIDS 不同，在我的改进方案中，我用与冲突的关系来衡量一个变

量的活跃性，具体地说，当一个冲突发生的时候，所有与这次冲突相关的子句（即对这次冲突负有责任的子句）中的文字所对应的变量的计数值都要被增大一定的量。什么样的子句是与一个冲突相关的呢？如果一个子句参与了生成这次冲突的导出子句的消解过程，则该子句是与这个冲突相关的。

（2）变量的时效性

在 VSIDS 中，“最近活跃的”中的“最近”通过周期性地用一个数去除各计数值来体现。在 QuickSAT 中，计数值也要定期地“老化”，而且分支变量的选择被限制在那些最近添加的值未确定的子句所包含的变量之中进行。

在设计我的决策策略的时候，我受到了如下两点事实的启发：

（1）众多的实验表明，当今的许多 SAT 程序已经能够快速地解决规模相当大的来自现实生产生活中的 CNF。这其实暗示了在这样的 CNF 中，通过在命题变量的一个小的子集中选择分支变量，可以从子句的一个小的子集中导出短的冲突导出子句（命题变量的小的子集、子句的小的子集都意味者较为集中的内存访问，这样可以减少访问高速缓存甚至内存的不命中，而生成短的子句意味着小的内存需求量以及强的信息）。

（2）与冲突有关的变量集合（亦即对冲突负有责任的变量的集合）可能是变化的，并且变化很快。例如，假设待判定其可满足性的 CNF 中的一个变量 $x$ 代表的是一个逻辑门电路的输出，该输出又同时作为许多与门电路的输入之一，则在 $x=0$ 这一决策分支子树中，与门电路的 $x$ 之外的其他输入对应的所有变量都与冲突的出现无关。然而，在 $x=1$ 的决策分支子树中，所有这些变量都马上再次变得可能与冲突相关起来，重新“活跃”起来了。

上述两个事实表明：解决现实生产生活中产生的 CNF，分支变量的选择策略应该是动态的，这个策略应该能够迅速地根据因新的变量赋值引起的与冲突相关的变量的集合的所有改变来进行调整（即随着搜索的进行，新的变量赋值会使得与冲突相关的变量集合发生改变，则变量导致冲突的活跃性也随之改变）。Chaff 的冲突导出子句的“老化”就是源于同样一种指导思想的。

在 QuickSAT 中，我对这种思想的实现如下：

为公式中的每个变量 $x$ 都设置一个计数器（记为 $count(x)$），用来存放那些至少与一个冲突相关（即至少对一个冲突的发生负有责任）并且含有变量 $x$ 的子句的数目。$count(x)$ 值在施行逆向 BCP 过程的时候被更新。每当新增一个与当前冲突有关的冲突导出子句时，该子句包含的每个变量 $x$ 的计数器 $count(x)$ 值就会被增 1。所有这些计数值都会周期性地（按照一定的间隔，例如遇到若干次冲突之后）除以一个大于 1 的小常量（在 Chaff 中，这个常量为 2，而在 QuickSAT 中，这个常量为 4）。通过这种方式，“老”的冲突导出子句对决策的

影响将被降低，而最近生成的冲突导出子句的影响起了主导作用。

在 Chaff 中，计算变量的“活跃性”的时候，只有那些出现在新增的冲突导出子句中的相应文字才被考虑在内，也就是说，如果一个变量 $x$ 的一个相应文字（$x$ 或 $\neg x$）出现在一个与当前冲突有关的普通子句（即存在于原 CNF 中的子句）中，但是却没有出现在一个冲突导出子句中时，Chaff 不会更新 $x$ 的“活跃性”。与 Chaff 不同，计算变量的“活跃性”时，我通过把更大的一个子句集合考虑在内使得 QuickSAT 能够更加准确地评估变量的“活跃性”。因为如果一个变量 $x$ 频繁地在 BCP 过程中获得赋值，那么，就算它没有出现在冲突导出子句中，也可能导致许多冲突。所以，如果在计算变量 $x$ 的活跃性的时候，单只考虑冲突导出子句，则相应变量的活跃性就容易被低估。

为了提高决策过程的动态性，在 QuickSAT 中，我引入了两个决策过程。其中一个过程非常简单，直接选择具有最大 $count(x)$ 值的变量 $x$ 作为下一个分支变量。然而这种方案并不是 QuickSAT 的主要决策策略。QuickSAT 的主要决策过程是基于冲突导出子句生成的时间先后顺序的，关于这个决策策略，我将在第 5.4.1 节介绍。

使用这样的分支决策策略的原因是我觉得 Chaff 的分支策略是一种“半动态”的策略，其“动态性”主要表现在每新添一个子句时都更新该子句所包含的文字的相应计数值并且周期性地将各计数器值除以一个常量，这种动态性还不足以满足解决广泛的 SAT 问题的需要（这就是出于健壮性的考虑）。Chaff 的策略尤其无法适应前面提到的相关变量集合的快速改变。对于这一点，可以举例来说明如下：假设 Chaff 以 100 次冲突作为周期来用 2 去除所有的计数器值，又假设在某次除 2 之后，就马上发生了相关变量的一次改变，则在紧跟着的 20 到 30 次冲突之内，分支变量的选择将会在一些已经“作废”了的变量集合中进行（只有在一定次数的冲突发生之后，发生了变化后的变量集合中的变量的相应计数器值才得以慢慢更新）。

### 5.4.1 分支变量的选择

在 QuickSAT 中，有一个栈用来存放所有子句的标识。初始 CNF 的所有子句放在这个栈的底部，并且每个新添加的冲突导出子句都被加到当前的栈顶。在搜索的过程中，由于对变量赋值，栈中的某些子句被满足。然而，栈中总会有一些仍未满足的子句，其中最接近栈顶的一个未满足的子句称为当前栈顶子句。如果当前栈顶子句是一个冲突导出子句，则它一定也就是最近生成的冲突导出子句，并且还没有被满足。

令 $C$ 是当前栈顶子句，则 QuickSAT 对 $C$ 的两种可能的情况分别处理如下：

（1）如果 $C$ 是一个冲突导出子句，则从 $C$ 中的自由变量中选择下一个分支变量，其方法是选择具有最大 $count(x)$ 值的变量 $x$。这样选出来的分支变量 $x$ 可能并不一定是当前所有自由变量中最“活跃”的。但是，这些当前的自由变量的活跃性可能是在冲突导出子句 $C$ 生成之前或者之后发生的不同的冲突的集合的结果，这些冲突可能牵涉到完全不同的变量集合。选择 $x$ 作为分支变量，我们已经考虑到了这些活跃的变量可能与子句 $C$ 的导出无关，甚至与跟 $C$ 相似的冲突（即涉及一个与 $C$ 中变量集合相近的变量集合的冲突）也无关的这样一个事实。

假设 $\neg x$ 是变量 $x$ 在当前栈顶子句 $C$ 中出现的文字，并且 $x$ 已经被选定为下一次的分支变量。如果当前栈顶子句并不是单子句，即除了含 $x$ 这个自由变量外，还有其他的自由变量，则 QuickSAT 不会为了满足 $C$ 而马上令 $x$ 赋值为 0。在选定了分支变量后，选择先搜索该分支变量的哪一个分支是属于另外一个独立的过程的工作（见 5.4.2）。然而，我们不难看出，在进行了不多于 $|C|-1$ 个分支变量的赋值后，子句 $C$ 最终将会被满足，这是因为：如果在选择了对 $x$ 的某种赋值之后，子句 $C$ 没有被满足，同时也没有产生冲突，则子句 $C$ 就会仍然是当前栈顶子句，则下一个分支变量的选择又将会在子句 $C$ 包含的自由变量进行，这样下去的结果一定是：或者 $C$ 由于某个分支变量的赋值或者由于 BCP 过程中的导出赋值而被满足，或者最后 $C$ 只剩下一个自由的变量，此时单子句规则将迫使该唯一的自由变量为真，因而子句 $C$ 得以满足。

（2）如果当前栈顶子句不是一个冲突导出子句，而是一个原来的 CNF 中就含有的子句，则在栈中所有的未满足的子句必定也都是原 CNF 中的子句。在这种情况下，使用前面提到的第一种分支决策方案，也就是具有最大的 $count(x)$ 值的自由变量 $x$（在这种情况下，这个变量也就是最活跃的自由变量）被选择为分支变量。

### 5.4.2 分支的选择

下面我们讨论在选定了分支变量之后，对其赋值的选择（也就是决定在搜索树上先遍历分支结点的哪一个分支）。显然，如果原 CNF 是可满足的，则不管算法是否利用了重启动策略，一个分支变量的两个分支子树一般是不对称的。然而，如果原 CNF 是不可满足的并且搜索算法没有使用重启动策略（即整个过程中只建立过一棵决策树），则一个分支变量的两个分支具有一定对称性（例子见图 5-5），也就是说，对于不可满足的 CNF，先选择分支变量的哪一个分支对搜索树的大小没有影响。

但是，如果算法利用了记录冲突子句的策略，则情况就有所不同。假设 $x$ 是

选出来的下一个分支变量，并且首先遍历的是 $x=0$ 对应的分支。另外一个分支子树 $x=1$ 的大小通常受到在分支 $x=0$ 中生成的冲突导出子句的影响，当然，并不是所有这些冲突导出子句都会影响到分支 $x=1$ 的子树的大小，因为如果一个冲突导出子句包含有文字 $x$，则在分支 $x=1$ 中这些子句都被满足。影响到分支 $x=1$ 中的计算的冲突导出子句是那些不含 $x$ 的子句。但是，由这些冲突导出子句

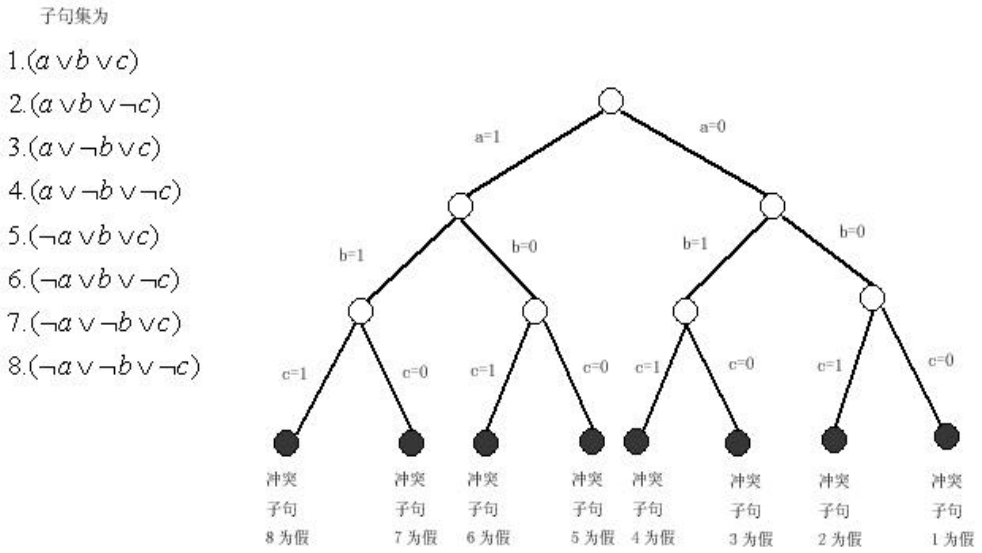



图 5-5 不可满足的问题所显示出来的对称性

Fig 5-5 The Asymmetry of an Unsatisfiable Instance

所表示的冲突是按 $x$ 对称的并且可以在遍历两个分支中的任何一个分支时生成，所以分支 $x=0$ 和分支 $x=1$ 是仅仅通过按 $x$ 对称的子句来“相互影响”的，而先遍历任一个分支所产生的子句对另外一个分支的影响都是相同的。所以，在分支 $x=0$ 和分支 $x=1$ 中的子树的大小与先遍历哪一个子树无关。然而，如果算法运用了重启动策略，即使原 CNF 是不可满足的，在分支变量的两个可选分支上也存在着不对称性。其原因是当在一个变量 $x$ 上分支时，如果在第一个遍历的分支上没有找出一个满足的解释，则算法不会搜索另外一个分支的子树（而采取了重启动）。

QuickSAT 的分支选择办法如下：

（1）当前栈顶子句是一个冲突导出子句时：

在这种情况下，对于选定的分支变量 $x$，选择对其先进行的赋值的指导思想是要使得正文字和负文字能够均匀地分布在将来生成的各冲突导出子句上。为了实现这一点，对于每个文字 $l$，按照如下方法计算其代价函数 $lit\_count(l)$：

$lit\_count(l)$ 就是到目前为止含有文字 $l$ 的冲突导出子句的数目。$lit\_count(l)$ 的初始值为 0。每当一个冲突导出子句 $C$ 被添加到子句数据库中，对于 $C$ 中的每个文字 $l$，其 $lit\_count(l)$ 都增大 1，$lit\_count(l)$ 不必除以一个常量。另外，当子句数据库中的冲突导出子句被删掉时（参见下文的子句数据库管理），$lit\_count(l)$ 的值不必相应地减少。

设 $x$ 是选定的下一个分支变量，$l$ 是 $x$ 对应的一个文字（$l \in \{x, \neg x\}$），如果 $l$ 具有较大的 $lit\_count(l)$（即 $lit\_count(l) > lit\_count(\neg l)$），则令 $x$ 取使得 $l$ 为 1 的赋值；如果 $lit\_count(x) = lit\_count(\neg x)$，则随机选择 $x$ 对应的一个文字 $l$（$l \in \{x, \neg x\}$）（这又是随机化的一个应用），然后给 $x$ 赋值为使 $l$ 为真。如此，所有带有文字 $l$ 的子句都被满足，并且此后如果遇到冲突，生成的冲突导出子句不会含有文字 $l$，则不会使 $lit\_count(l)$ 变大。与此同时，相反的文字（即 $\neg l$）变为 0，以后生成的冲突导出子句可能含有 $\neg l$，这将会有可能使到 $lit\_count(\neg x)$ 值增大。因此，通过设置具有最大 $lit\_count(l)$（$l \in \{x, \neg x\}$）值的文字为 1，我们缩小了各冲突导出子句中 $x$ 和 $\neg x$ 出现次数的差，使得其分布向着平均的方向发展。

（2）当前栈顶子句不是冲突导出子句，而是原 CNF 中包含的子句时：

对于每个自由文字 $l$，按照如下方法计算其评价函数 $nb\_two(l)$：

1）计算所有含有自由文字 $l$ 的二元子句（即长度为 2 的子句或者除了两个自由文字之外，其他的文字都为 0 的子句）的数目；

2）对于每个包含自由文字 $l$ 的二元子句 $C$，设 $C$ 中除了 $l$ 以外的另一个自由文字为 $v$，计算所有含有文字 $\neg v$ 的二元子句的数目；

3）将上述两个数目相加，得到 $nb\_two(l)$。

评价函数 $nb\_two(l)$ 可以作为在把文字 $l$ 设为 0 之后执行 BCP 的效果的估计。$nb\_two(l)$ 的值越大，在把子句 $C$ 中的文字 $l$ 设为 0 后，BCP 过程就能导出越多的赋值。如果 $nb\_two(x) = nb\_two(\neg x)$，则随机选择其中之一（随机化的又一个应用）。

根据这样方案选出来文字 $l$ 后，给变量 $x$ 赋使 $l$ 为 0 的值。

为了减少 QuickSAT 用于计算 $nb\_two(l)$ 的时间，可以通过为 $nb\_two(l)$ 规定一个上限值的办法，在 QuickSAT 的这个实现中，这个上限值设为 100，也就是说，一旦 $nb\_two(l)$ 的值大于 100，则计算终止，令 $nb\_two(l)$ 停留在 100。

QuickSAT 的决策过程相对以往的 SAT 程序显得有点复杂。但是，通过大量的实验比较证明，这是一种高效的方法，并且具有较高的健壮性（见附录 A）。

## 5.5 子句数据库管理方案

因为 QuickSAT 使用了重启动，这就存在一个问题，那就是在重启动之后，对上一次搜索结果的保留问题：如果保留得太多，就会占用太多的内存，如果保留的太少，就白白浪费了许多工作，特别是某些赋值，在重启动的前后都应该是相同的，这样的赋值其实是应该保留的，以避免以后再次重复计算。

出于这方面的考虑，QuickSAT 在子句数据库方面的管理方案如下：

在重启动之后，开始下一次的搜索（创建一棵新的搜索树）之前，有些子句会被永久地从子句数据库中删掉，其目的是为了降低程序对内存的需求。在 QuickSAT 删除子句的过程中，其中的数据结构被部分或者全部重新计算以便能够被放入更小的内存块中。

保持上一次搜索中得到的一些有用的赋值的结果就是使到有些子句会自动地被删除（因为这些赋值使到这些子句满足）。这样的赋值包括：在上一次搜索中，所有从冲突导出的单子句中推出的赋值和由冲突导出单子句导出的这些赋值再触发 BCP 过程而推出的赋值。所有因为这些保留赋值而被满足的子句都会被从原来的 CNF 子句数据库中删掉。

其它子句的删除根据以下启发式策略来进行。这个策略的基础是这样的一个假设：越是最近导出的子句的价值越大，因为程序需要花费更多的时间来将它们从原来的子句集合中导出。

QuickSAT 将冲突导出子句的集合看成一个 FIFO 的队列（这跟前面将子句组织成一个反映子句生成的时间先后顺序的栈并没有冲突，因为无论是前面的栈还是这里的队，里面存放的都只是指向子句的指针）。新的冲突导出子句被添加到这个队列的末端，而要考虑删除的子句都位于队列的头部。在队头的共占整个队列长度 1/16 的子句都在考虑删除之列，只要其中的子句的长度超过 8 个文字，就将被删除；而剩下的 15/16 的子句，只有当其长度超过 42，才会被删除。

另外，QuickSAT 总是保留最后添加的那个冲突导出子句和一部分特别“活跃”的冲突导出子句，而不管这些子句有多少个文字。一个子句 $C$ 的活跃性由它所直接负有责任的冲突的次数来衡量（如果子句 $C$ 参与了导致这个冲突的 BCP 过程，就说它对这个冲突负有直接责任）。在队头部的 1/16 的子句中，只有活跃

性大于 60 的子句才是“活跃”的，才能得以无条件地保留，而在队的其他子句中，只要活跃性大于 7 的子句就可以被认为是“活跃”的了。搜索过程每搜索 1024 个结点，队头部（1/16）的子句被认为“活跃”的有关的冲突次数的最小值就增 1（从 60 开始）。根据这个策略，不再导致冲突的长的子句将会被删掉。

在 QuickSAT 中，当前的搜索树在生成 550 个冲突导出子句（这些导出子句被添加到队尾）之后就会被放弃。在开始构建一棵新的搜索树之前，QuickSAT 就开始删掉构成这个队的至少 1/16 的冲突导出子句。如果在施行了上述的规则之后，被删除的子句数少于队中子句总数的 1/16，则子句删除的长度限制将会减少 1（但是，如果减少到了 4，就不再减）。在我的实验测试中，QuickSAT 几乎总是能够删掉队的 1/16 的子句，结果保持了一个较精简的子句数据库。

但是，必须注意，上述的删除子句的过程有可能会导致 QuickSAT 变得不完备，这是因为存在产生循环回路的可能性——上述算法可能删掉了某些子句，然后又重新生成了同样的子句，然后又删除了这些子句，然后又重新生成了同样的子句……举个例子，假设所有在当前搜索中导出的子句都具有多于 42 个文字并且其活跃性都小于 7，则所有这样的子句都将被从子句数据库中删除；然后在算法重启动之后，又会生成同样的一些子句，然后算法又将它们删除，……

一种消除这种循环回路的简单方法如下：给从上次重启动以来生成的一个冲突导出子句做标记，这个子句将永远被禁止从子句数据库中删除（除非它被一个从上一次搜索中保留下来的赋值所满足）。然后，我们只要保证在子句数据库中这种标记的子句数目是单调增长的，这样就避免了循环回路。

## 5.6 QuickSAT 的实现

QuickSAT 使用 C++语言来实现，实现中充分利用了 STL 中提供的容器类和库函数。程序在 RedHat Linux 平台（Linux 内核版本 2.4.18-3）上开发，所使用的编译器是 gcc-3.1。

### 5.6.1 基于 Profiling 的实现哲学

Profiling，可以翻译为程序运行分析，是指通过某种方法监测程序运行时期的性能。

为什么要进行 profiling 呢？这是因为通过 profiling，程序员可以：

（1）研究一个程序一次运行的性质，包括：运行时间都花在什么上面了；运行过程中哪些函数调用了其他的哪些函数；各个函数被调用了多少次；找出不正常的地方（例如某个小函数却占用了大部分的运行时间）和故障。一些 profiling 的有用经验包括：如果一个函数的运行时间比它所调用的各个子函数的运行时间的总和要高出很多，则要注意优化这个函数；否则就可以专注于优

化各子函数和对子函数调用的次数；当一个开销很大的子函数被调用的次数很多，则要用心降低调用的次数，否则要注意优化子函数本身……

（2）可以用来分析非常复杂的程序。

程序员要自己实现 profiling 的一种简单的方法是在程序中加入获得系统时间的系统调用。但是，效果并不太好，存在以下问题：

（1）要求程序员事先知道要对那一部分代码进行测量；

（2）结果只是给出测量的整个代码区域的总的时间；

（3）只是给出运行时间，而不给出执行的次数；

（4）只是得出总的时间，没有各个被调用的子函数的运行时间；

（5）给出的是经过的时间，而不是用户时间。

而专用的工具如 UNIX、Linux 环境下的 gprof、Windows 环境下的 PREF 和 PROFILE 就不存在这些不足。因此，我在开发 QuickSAT 过程中进行 profiling 工作所用的就是在 Linux 环境下的专用工具 gprof。gprof 在 1982 年最初出现在 BSD UNIX。

gprof 的输出主要有以下内容：

（1）平面分析（flat profile）及其描述。平面分析是程序的各个函数所花费的总时间，它提供的是运行时间的一个一维的视图，具体的内容包括：函数运行时间占整个程序总运行时间的百分比、各函数的累计运行时间、各函数单独的运行时间、函数被调用的次数、总调用次数等等；

（2）调用图（call graph）及其描述。调用图通过函数调用的父子关系来表示运行时间。具体内容包括：各个函数被哪个（些）函数所调用（父函数信息）、函数被调用的次数、各个函数调用了哪个（些）函数（子函数信息）、总的运行时间以及递归调用的次数等等；

（3）函数名索引。函数名索引是所有被分析的函数的一个按字母表顺序的名字索引，用来对调用图中函数的交叉索引。

进行 profiling，必须遵循如下几点基本原则：

（1）必须运行程序，只有执行了的代码才能被分析到；

（2）细心、精心地设计输入；

（3）应该运行一段较长的时间（可以使用大规模的输入或者运行多次）。

我在开发 QuickSAT 的过程中，大量使用了 profiling 的手段。例如，我发现整个程序中，占总运行时间最大百分比的部分是 deduce ( )，其次是 decide ( )，因此，我在改进和优化 QuickSAT 的时候，把主要精力放在了快速 BCP 过程和分支决策策略的改进上。通过不断的调整、修改，然后又再次 profiling，然后又根据 profiling 的结果再次调整、修改，终得到了比较理想的性能。

### 5.6.2 数据结构

在 QuickSAT 的实现中，每个文字由一个 32 位的整数来表示。用其中最低的 1 位表示其“相值”（0 表示是正文字，1 表示是负文字）。

一个文字是带有“相值”（也就是“符号”）的一个变量。一个文字由两个要素决定：文字的符号（sign）和对应变量的序号（index）。由于符号只有两种可能：正或负，所以我们只需要使用一个位（bit）来标记：如果该位为 0，表示正文字，如果该位为 1，表示负文字。

每个文字由其变量序号和文字符号（正或负）一起确定，文字的编码转换公式为：

$$j : V \times \{0,1\} \rightarrow Z^{+}$$

$$j(v,s) = 2 \times v + s, \qquad (5.9)$$

其中$V$是从 1 开始的变量序号集合，$v \in V, s \in \{0,1\}$，正文字的$s$为 0，负文字的 s 为 1。简单地说，如果一个文字，其对应的变量为$v \in V$，其符号是$s \in \{0,1\}$，则该文字对应的整数为$2 \times v + s$。

可见，若一个文字号为正偶数，则是一个正文字，其文字符号为 0，其对应的变量序号＝文字号÷2；若一个文字号为正奇数，则是一个负文字，其文字符号为 1，对应的变量号＝（文字号－1）÷2。

所有的文字都存放在一个称为文字池的大数组中。在一个文字池中的一个元素要么是一个文字，要么是用于标识一个子句的结尾的占位元素，占位元素的值为对应子句号的负值。

子句由一个整型数组表示。数组中的每个整数表示一个文字。每个子句由一个子句 index（索引）来标识。

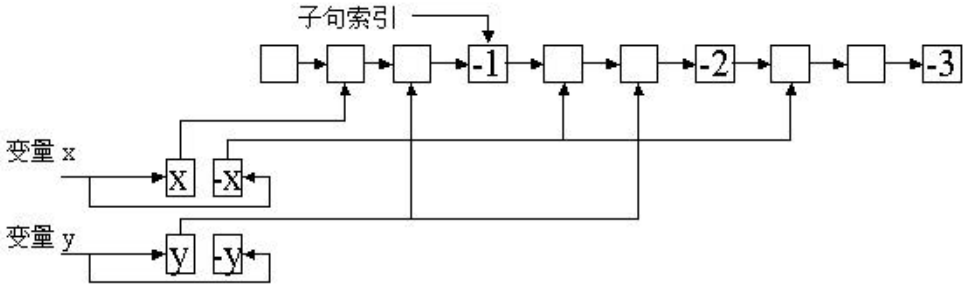
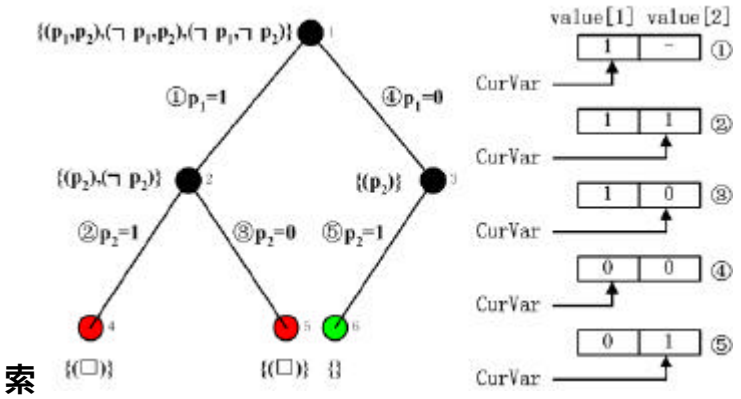



图 5-6 文字池

Fig 5-6 The Literal Pool

为了对赋值和回溯过程进行控制，构造一个数组 value 来存放对各变量的赋值，同时跟踪赋值的顺序，例如，对于公式$\{(p_1, p_2), (\neg p_1, p_2), (\neg p_1, \neg p_2)\}$，其搜




索

图 5-7 赋值数组 value 的使用

Fig 5-7 The Assignments Array

过程和 value 数组变化过程如图 5-7（变量 CurVar 标记当前赋值的变量）。

此外，还有一个存放处于单子句状态的子句的队列_implication_queue。

### 5.6.3 QuickSAT 的主要类和函数

QuickSAT 是用 C++语言实现了，采取的是面向对象的开发方法。QuickSAT 包含的主要类包括：

- 类 CLitPoolElement

定义一个文字。在类 CLitPoolElement 中，只有一个数据成员_val，其他都是成员函数。_val 是一个 32 位的整型变量。代表的就是前面所提的对一个文字的编码。该类中的成员函数主要就是对这个变量的一些存取函数。

- 类 CClause

定义一个子句。一个子句由若干个文字组成。公式的所有文字都集中存放在单个大的 vector（这是一种 C++ STL 中的标准类）里，这个 vector 就是前面提到的文字池。每一个子句有一个指向其头文字的指针。

- 类 CVariable

定义一个变量。这个类中包含了一个命题变量所不可缺少的信息。

- 类 CDatabase

定义子句数据库。子句数据库是 SAT 程序存放许多信息的地方，它是类 CSolver 的父类。类 CDatabase 包含如下的重要数据成员：

vector<CVariable> _variables：存放变量序号的数组；

vector<CClause> _clauses：子句序号数组。

- 类 CSolver

这个类包含了解决 SAT 问题的过程和数据结构，是 CDatabase 的子类，即还包含了 CDatabase 的数据成员和成员函数。这个类中的重要数据成员有：

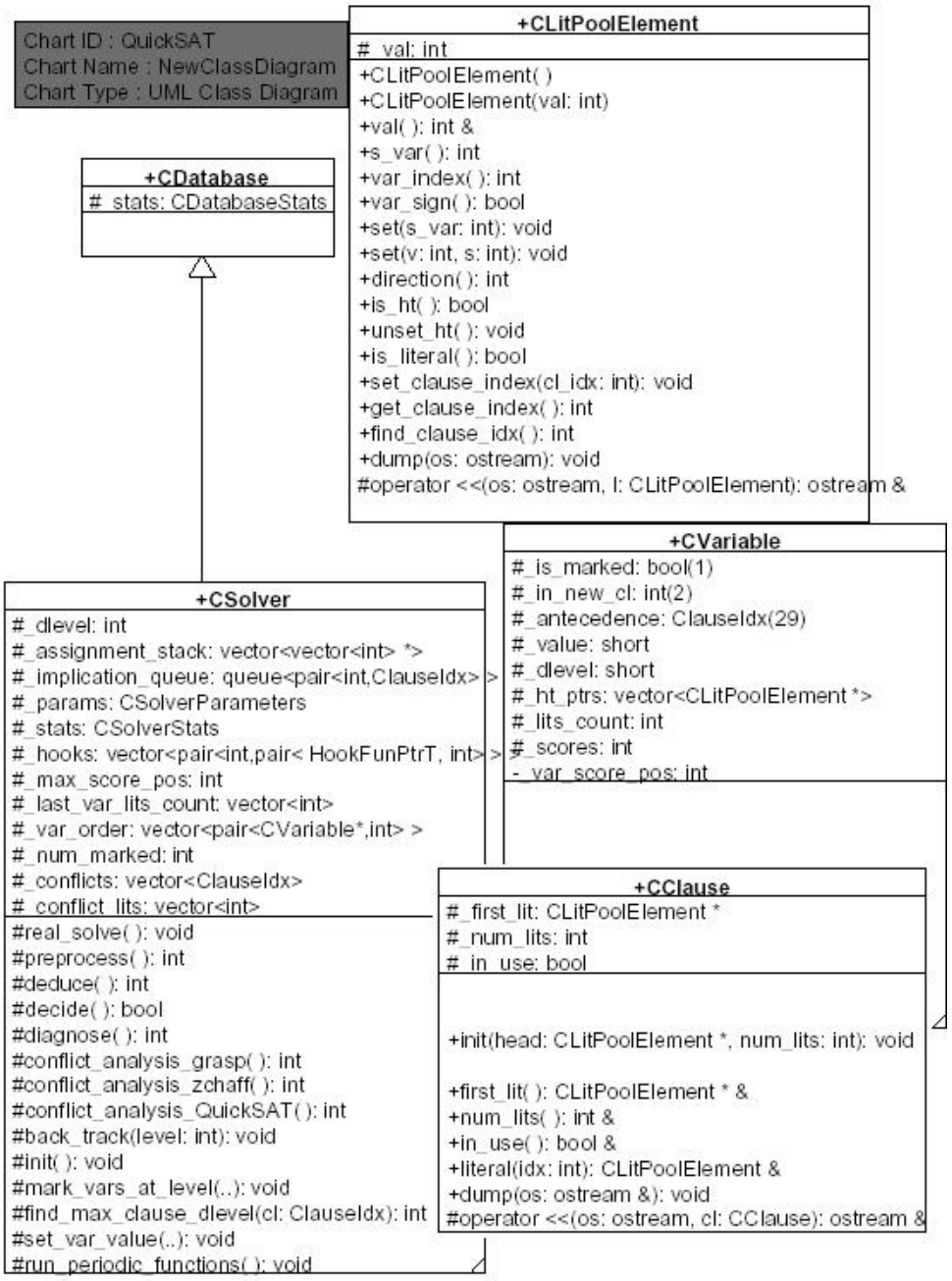



**图 5-8 QuickSAT 的主要类图**

Fig 5-8 The Significant Classes of QuickSAT

vector<vector<int> *> _assignment_stack：**前文提到的** value **数组**；

queue<pair<int,ClauseIdx> > _implication_queue：**单子句的队列**。

- **类** CDatabaseStats

**定义子句数据库的统计数据**。

- **类** CSolverParameters

**定义** SAT **程序的各种参数。注意** CSolverParameters **与后边的** CSolverStats **的区别，前者是** QuickSAT **的用户可以设置的参数，后者是运行** QuickSAT **后的**

统计数据。

- 类 CSolverStats

定义 SAT 程序的统计数据。

QuickSAT 中主要类的类图见图 5-8。

### 5.6.4 QuickSAT 的程序流程

QuickSAT 是在通用框架 3D-DPLL 的基础上开发的，除了增加了一个执行周期性工作的函数之外，基本没有大的不同。其流程图如图 5-9：

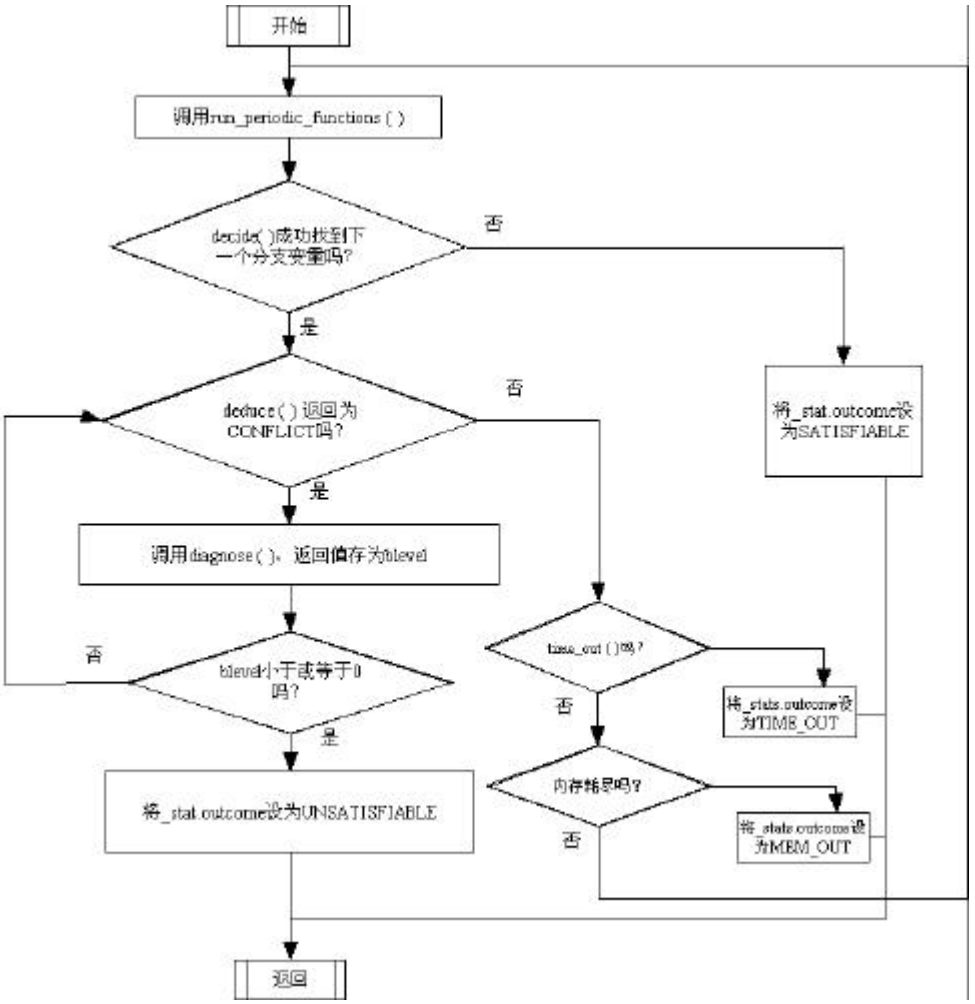



图 5-9 QuickSAT 的基本流程

Fig 5-9 The Work Flow of QuickSAT

其中的 decide ( )、deduce ( )和 diagnose ( )函数就不再解释了，我们这里主要说明一下函数 run_periodic_functions ( )，这是一个执行重启动和周期性工作（主要是子句删除）的函数，其流程如图 5-10。

### 5.6.5 QuickSAT 的用法

QuickSAT **是一个通用的** SAT **程序，其他应用中的程序可以使用** QuickSAT **来解决** SAT **问题。**QuickSAT **提供了如下的接口：**

SAT_Manager SAT_InitManager ( void )：**初始化一个公式管理器；**

void SAT_ReleaseManager ( SAT_Manager mng )：**释放公式管理器；**

void SAT_SetNumVariables ( SAT_Manager mng, int num_vars )：**设置变量的数目；**

int SAT_AddVariable ( SAT_Manager mng )：**添加一个变量，返回值为该变量的序号；**

void SAT_AddClause ( SAT_Manager mng, int * clause_lits, int num_lits)：**添加一个子句；**

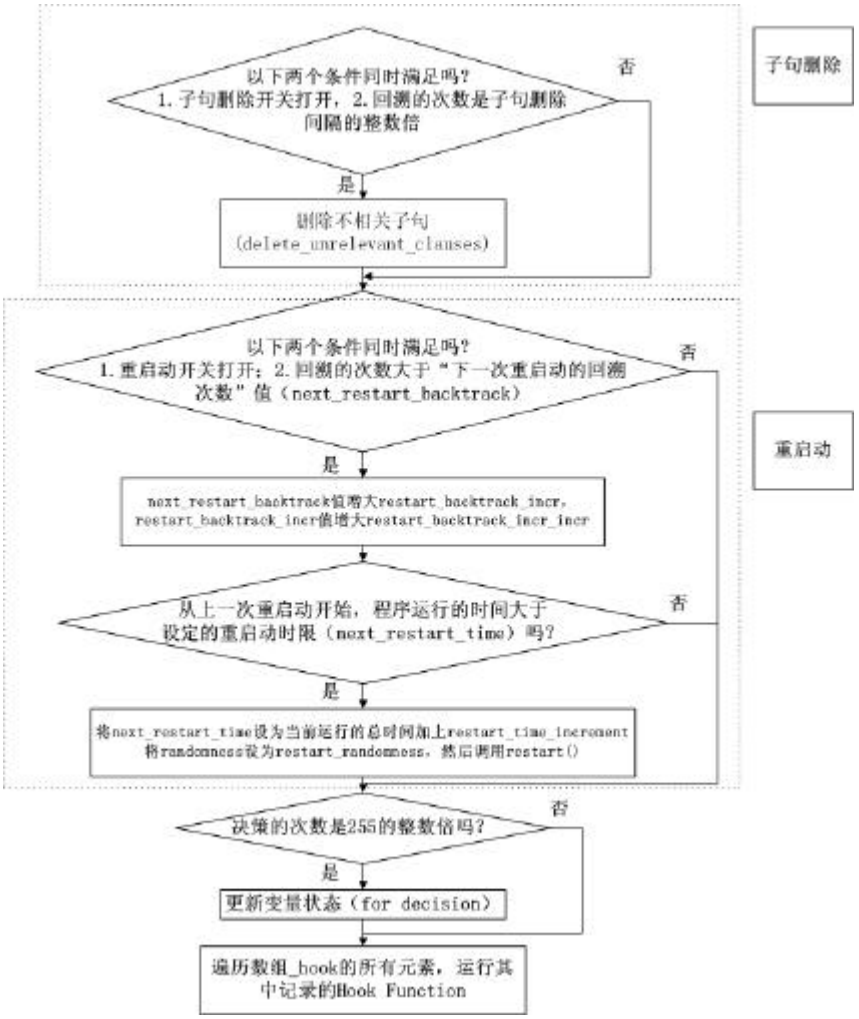



**图 5-10 函数 run_periodic_functions ( )的流程**

Fig 5-10 The Flow of Function run_periodic_functions ( )

void SAT_SetTimeLimit ( SAT_Manager mng, float runtime )：**设置运行时间上限；**

void SAT_SetMemLimit ( SAT_Manager mng, int num_bytes )：**设置程序运行所用内存的上限；**

enum SAT_StatusT SAT_Solve ( SAT_Manager mng )：**真正开始运行程序，返回程序的状态，也就是：公式是可满足、不可满足的或无法断定的；**

int SAT_GetVarAsgnment ( SAT_Manager mng, int v_idx )：**获得一个变量的赋值；**

void SAT_SetRandomness ( SAT_Manager mng, int n )：**用来设置在决策过程中所使用的随机性参数；**

void SAT_AddHookFun ( SAT_Manager mng, void(*fun)(void *), int interval )：**添加一个辅助函数** fun；

……

QuickSAT **的一种使用方式如下：**

（1）**调用函数** SAT_InitManager ( )**以生成一个新的公式“管理器”。 用户可以在此时预设文字的数目，也可以在以后通过调用函数** SAT_AddVariable ( ) **来添加文字。**

（2）**调用函数** SAT_AddClause ( )**来添加子句。**

（3）**此时，可以设置** QuickSAT **的运行时间上限和使用内存的上限，但是要注意的是这样设置的时间和内存值并不是准确值，只是近似的。同时，也可以设置其他的参数，例如子句删除的长度上限等。**

（4）**用户可以在此时添加一些辅助函数来执行在完成了若干次决策后需要做的一些额外的工作。**

（5）**调用函数** SAT_Solve ( ) **解决问题。其返回值是输入公式的状态（可满足的，不可满足的）。**

（6）**如果问题是可满足的，则可以调用函数** SAT_GetVarAsgnment ( )**来取得一个满足的解释。**

（7）**此时，可以调用状态统计函数来获得一些统计数字，例如运行时间、内存利用率等等。**

（8）**调用** SAT_ReleaseManager **来释放“管理器”**（manager）。

**要使用** QuickSAT，**用户程序需要连接函数库** libsat.a，**同时，尽管用户可以用** C **编译器（例如** gcc）**来编译其** C **程序，但是始终需要用** C++**链接器（例如** g++）**来链接这个函数库。**

# 第六章 结论与展望

## 6.1 实验结果分析

附录给出了我所进行的三个主要的实验的测试结果。其他的实验，包括为了优化 QuickSAT 的实现而在大量测试实例上进行的 profiling，由于涉及太多实现中的细节，并且其结果与我们的结论无关，也就不再给出了。

首先是关于三种分支决策策略，即 DLIS、VSIDS 和 QuickSAT 所使用的策略之间的比较。这是使用 3D-DPLL 来作为测试台的，即除了决策引擎根据 DLIS、VSIDS 和 QuickSAT 的决策方案而有所不同之外，其他部分都是一样的，采用基于 zChaff 的方案。实验采取抽样测试的方式，即在若干重要的基础测试类中各选出几个代表问题作为测试实例。实验结果显示(见附录 A)，QuickSAT 的决策策略除了在 VLSI 基准测试类中的 dlx2_aa 问题上比 VSIDS 要慢之外，其他的所有测试都是最优的。这在一定程度上证明了 QuickSAT 所使用的分支决策策略是 QuickSAT 的健壮性的原因所在。

第二个比较实验是我利用 3D-DPLL 框架实现的三种主要的 SAT 算法：GRASP、zChaff 和 QuickSAT 之间的比较。为什么要这样做？这是因为只有在同样的开发环境下（包括同样的机器、同样的编译器、同样的操作系统、相同的主要程序结构、同一个开发者等），对不同的算法进行比较有可能反映出它们本质上的性能差别。实验在 17 个大类共 143 个不同的基准测试实例上进行，结果表明（见附录 B），QuickSAT 在性能上远远超出了 GRASP，在大多数情况下跟 zChaff 相近，而在 VLIW-SAT 1.0 类问题中的一个实例，zChaff 无法在规定的时间内（1000 秒）完成，而 QuickSAT 可以完成。这也是 QuickSAT 比 zChaff 更加健壮的一个表现。

当然，上一个实验中对各算法的实现都是在有限时间内完成的，很多时候直接借用 GRASP 和 zChaff 的代码，实现的程序是比较粗糙的，而对 SAT 算法的优良的实现也是一种加速最后生成的 SAT 程序解决 SAT 问题的很重要的非算法手段。我既然在这次毕业设计开始之初立志于构建一个高效、健壮的 SAT 程序，则要的是最后生成的“成品”是比较高效和健壮的，因此第三个测试是四个成品 SAT 程序 GRASP、SATO、zChaff 和 QuickSAT 之间的比较。实验结果（见附录 C）再次表明了 QuickSAT 在效率和健壮性上都是非常好的：因为在进行实验的 34 大类共 261 个基准测试问题中的大部分上，QuickSAT 在速度上都跟 SATO 和 zChaff 不相上下，但是 QuickSAT 能在控制时间内（这次是 10000

秒）解决的问题数量比其他三个程序都要多。

## 6.2 结论

经过与当今世界上最好的几个 SAT 程序之间的实验比较，证明了 QuickSAT 是一个高效而且健壮的 SAT 程序。

通过借助 3D-DPLL 通用框架，经过大量的实验和改进，QuickSAT 成功地高效集成了 GRASP、SATO、Chaff 等的许多先进技术，包括记录冲突子句、快速 BCP 过程、重启动以及子句删除技术。

同时，为了提高效率，在设计和实现 QuickSAT 的过程中，我进行了如下的工作：

（1）从算法上改进，具体包括：

1）对快速 BCP 过程的改进，其中提出了一种新的在变量赋值时判断子句状态变化的方案“2-文子监视模式”；

2）QuickSAT 使用一种新的子句数据库管理方案，考虑是否删除一个冲突导出子句时，这种方案不但考虑子句的长度，而且考虑它可能导致冲突出现的活跃性和它存在子句数据库中时间的长短。

（2）在实现上使用了新的技术和方法。主要是贯彻了算法工程的思想和方法，其中在实现和优化 QuickSAT 的过程中主要依赖于 profiling 技术，另外在数据结构的选择和程序运行逻辑的设计上都注重了提高程序访存的局部性，以减少程序运行过程中对 Cache 或者内存的不命中。

另一方面，为了提高健壮性，我主要是在 QuickSAT 的分支决策策略上做功夫，其改进包括：

（1）所有添加的子句都是被按照生成时间的先后排序的，这使得 QuickSAT 的分支策略总是企图满足最近导出但仍没有满足的冲突导出子句。

（2）通过考虑更为广泛的子句的集合，QuickSAT 能够更好地评估变量的活跃性。

实验表明，我所作的这些工作是确实能够有效地提高 QuickSAT 的效率和健壮性的。

## 6.3 将来的工作

当然，我的工作还有许多不足之处，其中包括：

（1）各 SAT 程序性能之间的比较所使用的主要手段仍然是基于程序运行的 CPU 时间。这种做法在国际上被称为“赛马式”（Horse-Racing）的程序比较。这种做法是必要的，但是，要从更深的层次上比较两种算法的性能，除了 CPU

时间外，应该借助于更多的其他衡量手段。对于 SAT 算法和程序来说，这些其他的手段包括比较解决同一问题时程序所执行的决策次数、BCP 过程的次数或者出现冲突的次数等等；但是，由于时间关系，作者没有做这一方面的工作；

（2）虽然 QuickSAT 跟 GRASP、SATO 和 zChaff 在具有代表性的一定数量的基准测试程序上进行了比较，并且证明了 QuickSAT 是比较高效而且健壮的。但是，QuickSAT 的在算法上没有彻底的突破，特别在性能上，QuickSAT 没有能够带来引人注意的突破，其性能基本上跟 zChaff 程序还是处于同一个数量级上。

而至于将来的工作，除了包括在上述两个方面改进之外，还可以在以下方面努力：

（1）要更好地量化各种搜索树剪枝技术本身的计算代价，只有通过准确、科学的手段来衡量一种新技术到底是否能给 SAT 算法带来加速，才能降低算法设计的盲目性；

（2）更好地结合底层计算机系统的体系结构来开发 SAT 程序，要充分利用好硬件的特点，让算法能够更高效地使用硬件；

（3）考虑采用可以动态调整的分支启发式策略，能够针对不同类型的问题来选择不同的策略；

（4）对基准测试实例进行研究；

对于像 SAT 问题这样的 NP 完全问题，测试实例的设计与选择非常重要，测试实例的好坏会影响到对算法的评价；对测试实例规律的研究，也能够促进 SAT 算法的改进。在 SAT 问题及其算法的研究中，其基准测试实例的研究也成了一个重要的研究分支。

（5）利用高能计算技术（分布计算、并行处理）来提高解决 SAT 问题的效率。

# 参考文献

[1]J. M. Wilson. Compact normal forms in propositional logic and integer programming formulations. Computer and Operation Research 17(3), pages 309-314, 1990

[2] 王浩，《数理逻辑通俗讲话》，科学出版社，1981

[3]S.A. Cook. The complexity of theorem proving procedures. In Proceedings of Third ACM Symposium on Theory of Computing, pages 151-158, 1971

[4] J.P. Marques-Silva and K.A. Sakallah. GRASP: a search algorithm for propositional satisfiability. IEEE Transactions on Computers, vol.48, (no.5), pp.506-21, 1999

[5] R. Bayardo and R. Schrag, Using CSP look-back techniques to solve real-world SAT instances, presented at National Conference on Artificial Intelligence (AAAI), 1997.

[6] Hantao Zhang, SATO: an Efficient propositional prover. Proc. of International Conference on Automated Deduction (CADE-97).

[7]Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In Proceedings of the 38th Design Automation Conference (DAC'01), June 2001.

[8]J. de Kleer. A comparison of ATMS and CSP techniques, Proceedings of IJCAI-89, pages 290-296, 1989

[9]张键，《逻辑公式的可满足性判定——方法、工具及应用》，科学出版社，2000

[10]Henry Kautz, Bart Selman. Planning as satisfiability. In Proceedings of the 10th European Conference on Artificial Intelligence (ECAI' 92), pages 359-363,1992

[11]R. Reiter, A.K.Mackworth. A logical framework for depiction and image interpretation. Artifi. Intel. 41. pages 125-155, 1989/90.

[12]J.M. Crawford, A.B. Baker, Experimental results on the application of satisfiability algorithms to scheduling problems, Proc 12th AAAI, p 1092 - 1097, 1994

[13]M. Davis and H. Putnam. A computing procedure for quantification theory, Journal of ACM 7(3), pages 201-215, 1960

[14]M. Davis, G. Logemann, D. Loveland. A machine program for theorem proving. Communications of the ACM 5(7), pages 394-397, 1962

[15] Bart Selman, Understanding Problem Hardness: Recent Developments and Directions, speech on ISAT'99

[16] Joao Marques-Silva，The Utilization of Randomization in Backtrack Search SAT Algorithms，AAAI' 02-PAS, July 2002

[17] M. Buro and H. Kleine-Buning, Report on a SAT competition, Technical Report, University of Paderborn 1992.

[18] J.W. Freeman. Improvements to Propositional Satisfiability Search Algorithms. PhD Dissertation, Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA, 1995

[19] J. P. Marques-Silva, The Impact of Branching Heuristics in Propositional Satisfiability Algorithms, the 9th Portuguese Conference on Artificial Intelligence (EPIA),1999.

[20]R.G. Jeroslow, J. Wang: Solving Propositional Satisfiability Problems. Annals of Mathematics and Artificial Intelligence 1: 167-187 (1990)

[21]I.P. Gent, T.Walsh. Easy problems are somtimes hard. Artificial Intelligence, pages 335-345, 1994

[22]C. Gomes, B. Selman, H. Kautz. Boosting combinatorial search through randomization. Proceedings of 15th National Conference on Artificial Intelligence, pages 431-437. 1998

[23]S.A.Cook, D.G.Mitchell. Finding hard instances of the satisfiability problem: A survey. Satisfiability problem:

theory and applications. DIMACS series in Discrete Mathematics and Theoretical Computer Science, Vol. 35, 1997

[24]R.Rodosek. A new approach on solving 3-satisfiability. Proceedings of 3rd International Conference on Artificial Intelligence and Symbolic Mathematical Computation, pages 197-212. LNCS 1138, 1996

[25]R.Beigel, D.Eppstein. 3-coloring in time o(1.3446n): a No-MIS algorithm. Technical report, ECCC, 1995. TR95-33

[26] J. Gu. Efficient local search for very large-scale satisfiability problems, ACM SIGART Bulletin 3(1), pages 8-12, 1992

[27]J.Gu. Local search for satisfiability (SAT) problem. IEEE Transaction on Systems, Man, and Cybernetics, 23(4), pages 1108-1129, 1993. Corrigenda, IEEE Trans. On Systems, Man, and Cybernetics, 24(4), pages 709, 1994.

[28]B. Selman, H.Levesque, D.Mitchell. A new method for solving hard satisfiability problems. Proceedings of 10th AAAI, pages 440-446, 1992

[29]B. Selman, H.A. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiablity problems, Proceedings of IJCAI-93, pages 290-295, 1993

[30] Bertrand Mazure, Lakhdar Sais, Éric Grégoire. System description: CRIL platform for SAT. Proceedings of CADE-15, LNAI 1421, pages 124-128, 1998

[31]B. Selman, H.A. Kautz, B.Cohen. Noise strategies for improving local search. Proceedings of 12th AAAI, pages 337-343, 1994

[32]李未、黄文奇 一种求解合取范式可满足性问题的数学物理方法. 中国科学〔E〕, 24(11), 1208 页－1217 页, 1994

[33]黄文奇、金人超 求解 SAT 问题的拟物拟人算法——Solar. 中国科学（E）, 27（2）, 179 页－186 页, 1997

[34] J. Zhang, H. Zhang. Combining local search and backtracking techniques for constraint satisfaction. Proceedings of 13th AAAI, Vol 1, pages 369-374, 1996

[35]J.N.Hooker. A quantitative approach to logical inference. Decision Support Systems 4(1), pages 45-69, 1988

[36]J. Gu, Q. Gu, and DZ Du: On optimizing the satisfiability (SAT) problem, Journal. of Computer Science and Technology, 14:1 (1999) 1-17

[37]J.M. Crawford and L.D. Auton. Experimental Results on the Cross-Over Point in Satisfiability Problems. Eleventh National Conference on Artificial Intelligence (AAAI-93),1993

[38] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26, pages 415-433, 1996.

[39]J. P. Marques-Silva and K. A. Sakallah, Conflict Analysis in Search Algorithms for Propositional Satisfiability, IEEE International Conference on Tools with Artificial Intelligence, 1996.

[40] J. N. Hooker and V. Vinay, Branching rules for satisfiability, Journal of Automated Reasoning, vol. 15, pp. 359-383, 1995.

[41]C. M. Li, Integrating equivalency reasoning into Davis-Putnam Procedure, National Conference on Artificial Intelligence (AAAI), 2000.

[42]Inês Anselmo, M. João Carrilho and João P. Marques-Silva, Solving Satisfiability with Backtrack Search and Recursive Learning, Technical Report RT/06/99, INESC, Portugal, December 1999.

[43] J. P. Marques-Silva, Improving Satisfiability Algorithms by Using Recursive Learning, International Workshop on Boolean Problems (IWBP), 1998.

# 附 录

以下的所有实验都是在处理器为 Intel® Celeron® 1.2G (L1 Cache 32K，L2 Cache 256K)，内存是 512M 的 SDRAM，操作系统为 Linux（内核 2.4.18-3）的环境下进行的；另外，一切我开发实现的程序都是是用 gcc-3.1 来编译的，编译器的优化开关都设为 O2。

## A.三种分支启发式决策策略的比较

利用 3D-DPLL 框架，在其他条件相同的情况下，分别实现了 DLIS、VSIDS 和 QuickSAT 的分支启发式决策策略，其他的策略都跟 zChaff 的配置相同。

| 问题实例集 | 问题 | 分支策略 | | |
|---|---|---|---|---|
| | | DLIS | VSIDS | QuickSAT |
| 电路测试问题（DIMACS） | bf0432-079 | 3.17 | 2.24 | 0.89 |
| | ssa2670-141 | 23.22 | 1.06 | 0.12 |
| 归纳推理问题（DIMACS） | ii16e1 | 0.5 | 0.42 | 0.23 |
| 奇偶性问题（DIMACS） | par16-1-c | 198.51 | 178.36 | 123.7 |
| 图着色问题 | flat200-39 | 125.95 | 12.96 | 9.62 |
| | sw100-49 | 10.3 | 2.35 | 2.21 |
| 积木问题 | 4blocksb | 323.74 | 85.94 | 62.3 |
| 规划问题 | logistics.c | 30.99 | 26.8 | 27.32 |
| | facts7hh.13.simple | 9.91 | 7.28 | 4.26 |
| Bounded Model Checking 问题 | barrel5 | 175.74 | 41.07 | 30.14 |
| | queueinvar16 | 13.36 | 16.39 | 14.82 |
| 超大规模处理器检验问题 | dlx2_aa | 12.31 | 10.12 | 11.59 |
| | dlx2_cc_a_bug17 | 3.58 | 2.86 | 1.46 |
| | 2dlx_cc_mc_ex_bp_f2_bug005 | 197.55 | 41.66 | 36.51 |
| 数据加密问题 | cnf-r3-b4-k1.2 | 16.78 | 15.39 | 13.67 |
| 北京问题集 | 3bitadd_32 | 12.94 | 8.76 | 4.89 |
| | e0ddr2-10-by-5-1 | 96.74 | 82.3 | 46.55 |
| 均匀分布的随机 3-SAT 问题 | uf200-01 | 11.74 | 13.76 | 10.67 |
| | uuf200-01 | 23.96 | 41.58 | 22.36 |
| | uf225-01 | 17.63 | 32.0 | 16.5 |
| | uuf225-01 | 42.37 | 67.23 | 39.83 |

表1 DLIS、VSIDS 与 QuickSAT 所用的决策策略之间的比较

# 附 录

以下的所有实验都是在处理器为 Intel® Celeron® 1.2G (L1 Cache 32K，L2 Cache 256K)，内存是 512M 的 SDRAM，操作系统为 Linux（内核 2.4.18-3）的环境下进行的；另外，一切我开发实现的程序都是是用 gcc-3.1 来编译的，编译器的优化开关都设为 O2。

## A.三种分支启发式决策策略的比较

利用 3D-DPLL 框架，在其他条件相同的情况下，分别实现了 DLIS、VSIDS 和 QuickSAT 的分支启发式决策策略，其他的策略都跟 zChaff 的配置相同。

| 问题实例集 | 问题 | 分支策略 | | |
|---|---|---|---|---|
| | | DLIS | VSIDS | QuickSAT |
| 电路测试问题（DIMACS） | bf0432-079 | 3.17 | 2.24 | 0.89 |
| | ssa2670-141 | 23.22 | 1.06 | 0.12 |
| 归纳推理问题（DIMACS） | ii16e1 | 0.5 | 0.42 | 0.23 |
| 奇偶性问题（DIMACS） | par16-1-c | 198.51 | 178.36 | 123.7 |
| 图着色问题 | flat200-39 | 125.95 | 12.96 | 9.62 |
| | sw100-49 | 10.3 | 2.35 | 2.21 |
| 积木问题 | 4blocksb | 323.74 | 85.94 | 62.3 |
| 规划问题 | logistics.c | 30.99 | 26.8 | 27.32 |
| | facts7hh.13.simple | 9.91 | 7.28 | 4.26 |
| Bounded Model Checking 问题 | barrel5 | 175.74 | 41.07 | 30.14 |
| | queueinvar16 | 13.36 | 16.39 | 14.82 |
| 超大规模处理器检验问题 | dlx2_aa | 12.31 | 10.12 | 11.59 |
| | dlx2_cc_a_bug17 | 3.58 | 2.86 | 1.46 |
| | 2dlx_cc_mc_ex_bp_f2_bug005 | 197.55 | 41.66 | 36.51 |
| 数据加密问题 | cnf-r3-b4-k1.2 | 16.78 | 15.39 | 13.67 |
| 北京问题集 | 3bitadd_32 | 12.94 | 8.76 | 4.89 |
| | e0ddr2-10-by-5-1 | 96.74 | 82.3 | 46.55 |
| 均匀分布的随机 3-SAT 问题 | uf200-01 | 11.74 | 13.76 | 10.67 |
| | uuf200-01 | 23.96 | 41.58 | 22.36 |
| | uf225-01 | 17.63 | 32.0 | 16.5 |
| | uuf225-01 | 42.37 | 67.23 | 39.83 |

表1 DLIS、VSIDS 与 QuickSAT 所用的决策策略之间的比较

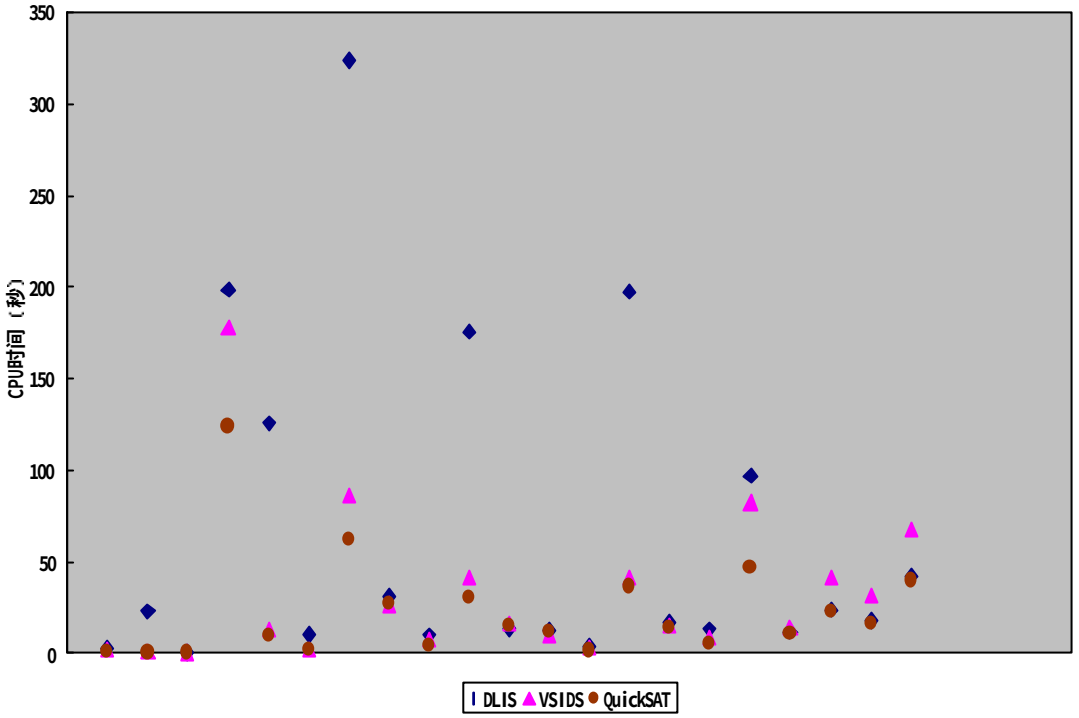



图 1 表 1 中数据的散点示意图

## B.同样用 3D-DPLL 实现的 GRASP、zChaff 和 QuickSAT 的比较

| 基准测试集 | 测试实例组 | 测试的实例数 | GRASP | | zChaff | | QuickSAT | |
|---|---|---|---|---|---|---|---|---|
| | | | 超时次数 | 总运行时间 | 超时次数 | 总运行时间 | 超时次数 | 总运行时间 |
| DIMAC 基准测试集[1] | ii8 | 4 | 0 | 1.4 | 0 | 0.1 | 0 | 0.1 |
| | ii16 | 8 | 1 | 326.8 | 0 | 7.6 | 0 | 8.2 |
| | ii32 | 10 | 0 | 2.7 | 0 | 0.6 | 0 | 0.6 |
| | aim100 | 12 | 0 | 0.7 | 0 | 0.1 | 0 | 0.1 |
| | aim200 | 12 | 0 | 7.9 | 0 | 0.3 | 0 | 0.3 |
| | pret | 8 | 0 | 7.2 | 0 | 0.8 | 0 | 0.6 |
| | par8 | 10 | 0 | 0.1 | 0 | 0.1 | 0 | 0.1 |
| | par16 | 10 | 7 | 1195.7 | 0 | 54.9 | 0 | 37.8 |
| | ssa | 8 | 0 | 3.2 | 0 | 0.3 | 0 | 1.1 |
| | jnh | 12 | 0 | 6.9 | 0 | 0.7 | 0 | 0.7 |
| | dubois | 10 | 0 | 0.3 | 0 | 0.2 | 0 | 0.3 |
| | hole | 5 | 2 | 299.9 | 0 | 128.7 | 0 | 124.4 |
| CMU 基准测试集[2] | SSS 1.0 | 10 | 5 | 1084 | 0 | 62 | 0 | 65 |
| | SSS 1.0a | 8 | 6 | 9190 | 0 | 25 | 0 | 37 |
| | SSS-SAT 1.0 | 10 | 20 | - | 0 | 632 | 0 | 601 |
| | PVP-UNSAT 1.0 | 4 | 2 | 2944 | 0 | 1033 | 0 | 977 |
| | VLIW-SAT 1.0 | 10 | 20 | - | 1 | 5665 | 0 | 3985 |

表 2 同等条件下实现的 GRASP、zChaff 和 QuickSAT 的比较

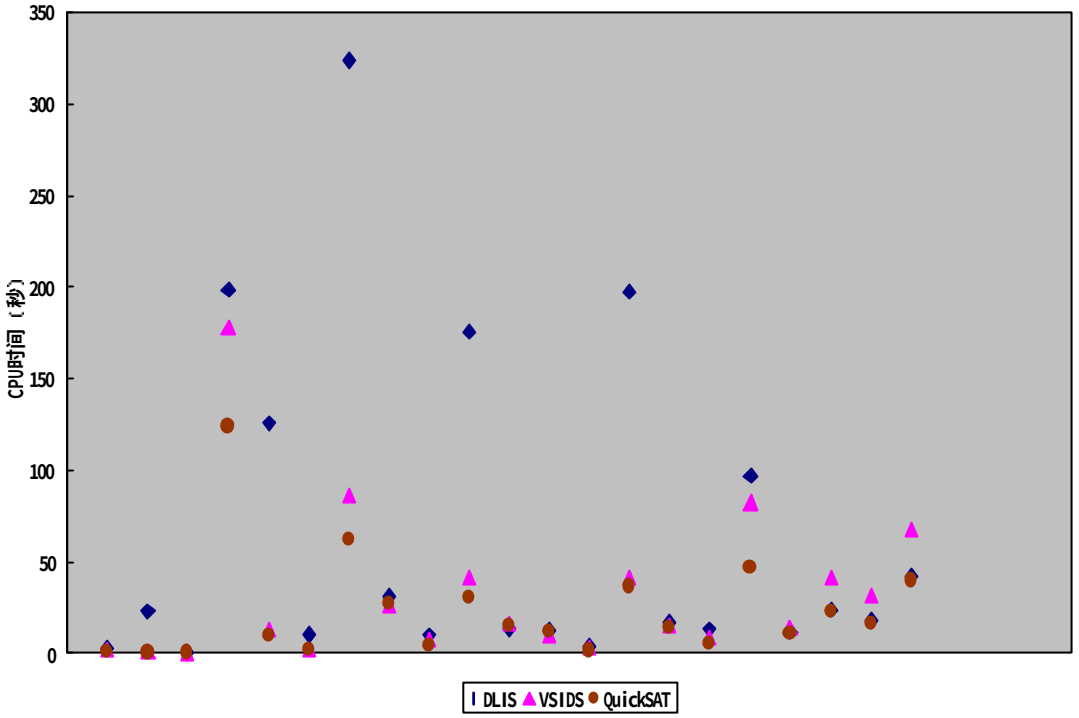



图 1 表 1 中数据的散点示意图

## B.同样用 3D-DPLL 实现的 GRASP、zChaff 和 QuickSAT 的比较

| 基准测试集 | 测试实例组 | 测试的实例数 | GRASP | | zChaff | | QuickSAT | |
|---|---|---|---|---|---|---|---|---|
| | | | 超时次数 | 总运行时间 | 超时次数 | 总运行时间 | 超时次数 | 总运行时间 |
| DIMAC 基准测试集[1] | ii8 | 4 | 0 | 1.4 | 0 | 0.1 | 0 | 0.1 |
| | ii16 | 8 | 1 | 326.8 | 0 | 7.6 | 0 | 8.2 |
| | ii32 | 10 | 0 | 2.7 | 0 | 0.6 | 0 | 0.6 |
| | aim100 | 12 | 0 | 0.7 | 0 | 0.1 | 0 | 0.1 |
| | aim200 | 12 | 0 | 7.9 | 0 | 0.3 | 0 | 0.3 |
| | pret | 8 | 0 | 7.2 | 0 | 0.8 | 0 | 0.6 |
| | par8 | 10 | 0 | 0.1 | 0 | 0.1 | 0 | 0.1 |
| | par16 | 10 | 7 | 1195.7 | 0 | 54.9 | 0 | 37.8 |
| | ssa | 8 | 0 | 3.2 | 0 | 0.3 | 0 | 1.1 |
| | jnh | 12 | 0 | 6.9 | 0 | 0.7 | 0 | 0.7 |
| | dubois | 10 | 0 | 0.3 | 0 | 0.2 | 0 | 0.3 |
| | hole | 5 | 2 | 299.9 | 0 | 128.7 | 0 | 124.4 |
| CMU 基准测试集[2] | SSS 1.0 | 10 | 5 | 1084 | 0 | 62 | 0 | 65 |
| | SSS 1.0a | 8 | 6 | 9190 | 0 | 25 | 0 | 37 |
| | SSS-SAT 1.0 | 10 | 20 | - | 0 | 632 | 0 | 601 |
| | PVP-UNSAT 1.0 | 4 | 2 | 2944 | 0 | 1033 | 0 | 977 |
| | VLIW-SAT 1.0 | 10 | 20 | - | 1 | 5665 | 0 | 3985 |

表 2 同等条件下实现的 GRASP、zChaff 和 QuickSAT 的比较

注：[1]超时时限为 100s；[2]超时时限为 1000s ；

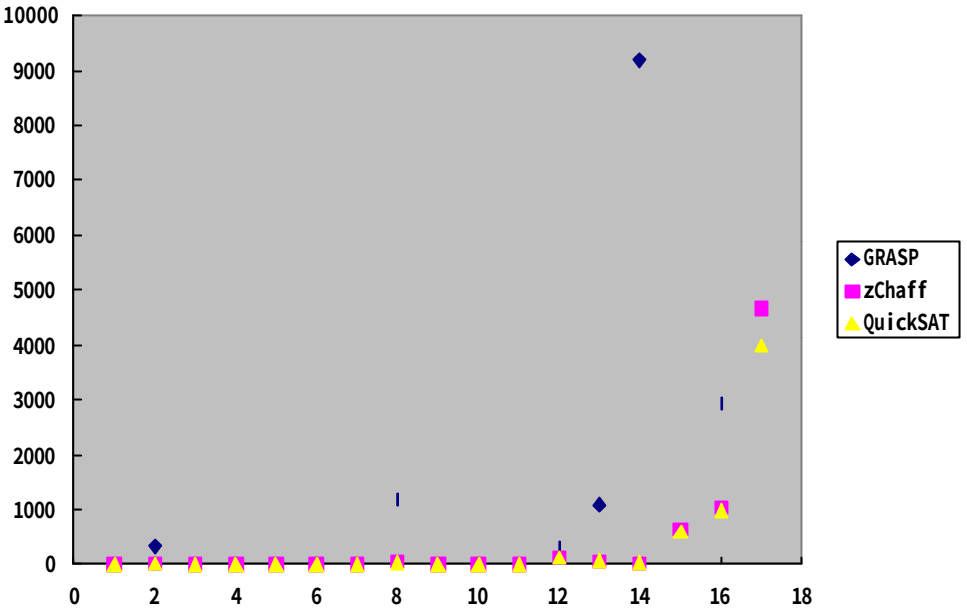



图 2 表 2 中数据的散点示意图

## C.最后得到的 QuickSAT 与 GRASP、SATO 和 zChaff 的比较

（1）SATO 用的是 3.2.1 版本，其源代码可以从 http://www.cs.uiowa.edu/~hzhang/sato.html 上下载。

（2）GRASP 使用的是 2000 年 2 月的版本，其源程序和 Linux 二进制版本可以从 http://sat.inesc.pt/~jpms/grasp/下载；使用以下开关参数：+T100 +B10000000 +C10000000 +S10000 +V0 +g40 +rt4 +dMSMH +dr5。

（3）zChaff 使用的是 2001 年的版本，源程序或 Linux 二进制代码可以从 http://ee.princeton.edu/~chaff/zchaff.php 下载；

（4）超时上限为 10000 秒；所有 GRASP、SATO、zChaff 的超时的运算都不是在我的实验中运行，而是直接采用 SAT-Ex 的数据，其可行性在于：使用同样的二进制程序，同样的设置，SAT-Ex 所使用的硬件系统比我的实验平台的性能配置要高，SAT-Ex 的运行中超时的，在我的实验中也必定超时。如果不是这样处理，本人的 PC 机无法提供足够的机时。

| Benchmark | GRASP | SATO | zchaff | QuickSAT |
|---|---|---|---|---|
| 1dlx_c_mc_ex_bp_f.cnf | 9.3 | 10000 | 0.26 | 0.16 |
| 2bitadd_10.cnf | 10000 | 3300.25 | 210.56 | 169.63 |
| 2bitadd_11.cnf | 1.67 | 143.58 | 7.24 | 5.44 |
| 2bitadd_12.cnf | 1.79 | 238.72 | 4.52 | 3.88 |
| 2bitcomp_5.cnf | 0.02 | 0.02 | 0.01 | 0.01 |
| 2bitmax_6.cnf | 0.05 | 0.03 | 0.02 | 0.01 |

| | | | | |
|---|---|---|---|---|
| 2dlx_ca_mc_ex_bp_f.cnf | 10000 | 10000 | 6.51 | 4.25 |
| 2dlx_cc_mc_ex_bp_f.cnf | 10000 | 10000 | 14.4 | 9.76 |
| 3bitadd_31.cnf | 10000 | 10000 | 10000 | 2782.65 |
| 3blocks.cnf | 7.69 | 0.16 | 0.26 | 0.3 |
| 4blocks.cnf | 4515.72 | 49.81 | 2.97 | 2.05 |
| 4blocksb.cnf | 86.36 | 1.18 | 1.03 | 0.92 |
| 9vliw_bp_mc.cnf | 10000 | 10000 | 1203.69 | 632.1 |
| add3.boehm.cnf | 0.47 | 0.24 | 0.07 | 0.04 |
| add4.boehm.cnf | 1.3 | 11.64 | 0.13 | 0.34 |
| addsub.boehm.cnf | 1.02 | 2.66 | 0.13 | 0.11 |
| aim-200-6_0-yes1-2.cnf | 0.1 | 0.02 | 0.05 | 0.03 |
| aim-200-6_0-yes1-3.cnf | 0.16 | 0.01 | 0.04 | 0.01 |
| ais10.cnf | 6.46 | 0.06 | 2.28 | 0.12 |
| ais12.cnf | 246.61 | 0.22 | 55.54 | 49.63 |
| ais8.cnf | 0.07 | 0.01 | 0.09 | 0.02 |
| anomaly.cnf | 0.01 | 0 | 0.01 | 0 |
| avg-checker-2-7.cnf | 0.21 | 0.04 | 0.07 | 0.1 |
| avg-checker-2-8.cnf | 0.35 | 0.02 | 0.05 | 0.3 |
| avg-checker-3-14.cnf | 115.7 | 10.15 | 1.25 | 2.32 |
| avg-checker-3-14-tcsimp.cnf | 93.83 | 9.57 | 2.71 | 3.26 |
| avg-checker-3-15.cnf | 21.69 | 3.07 | 1.51 | 1.56 |
| avg-checker-3-15-tcsimp.cnf | 15.5 | 4.83 | 0.89 | 1.21 |
| avg-checker-4-23-tcsimp.cnf | 10000 | 10000 | 83.66 | 66.91 |
| avg-checker-4-24.cnf | 10000 | 10000 | 55.71 | 42.38 |
| barrel3.cnf | 0.11 | 0.03 | 0.03 | 0.05 |
| barrel4.cnf | 0.5 | 0.13 | 0.1 | 0.1 |
| barrel5.cnf | 1203.87 | 14.81 | 5.59 | 3.64 |
| barrel6.cnf | 10000 | 84.4 | 25.91 | 21.74 |
| barrel7.cnf | 10000 | 306.54 | 63.45 | 32.06 |
| barrel8.cnf | 10000 | 819.5 | 140.98 | 103.5 |
| bf0432-007.cnf | 1.45 | 6.02 | 0.35 | 0.42 |
| bf1355-075.cnf | 0.31 | 0.07 | 0.12 | 0.1 |
| bf1355-638.cnf | 0.18 | 0.07 | 0.11 | 0.06 |
| bf2670-001.cnf | 0.16 | 0.07 | 0.05 | 0.06 |
| bw_large.a.cnf | 0.14 | 0.03 | 0.07 | 0 |
| bw_large.a.cnf | 0.14 | 0.03 | 0.06 | 0.05 |
| bw_large.b.cnf | 0.99 | 0.24 | 0.27 | 0.22 |
| bw_large.b.cnf | 1 | 0.25 | 0.19 | 0.15 |
| bw_large.c.cnf | 81.97 | 6.75 | 3.93 | 2.39 |
| bw_large.c.cnf | 81.49 | 6.93 | 4.02 | 3.95 |
| bw_large.d.cnf | 1427.06 | 775.4 | 36.92 | 41.72 |
| bw_large.d.cnf | 1401.15 | 784.14 | 38.04 | 21.64 |
| cnf-r1-b4-k1.1-comp.cnf | 0.62 | 0.05 | 0.1 | 0.07 |
| cnf-r1-b4-k1.2-comp.cnf | 0.61 | 0.07 | 0.1 | 0.05 |
| cnf-r2-b1-k1.1-comp.cnf | 0.14 | 0.03 | 0.06 | 0 |
| cnf-r2-b1-k1.2-comp.cnf | 0.52 | 0.05 | 0.04 | 0.07 |
| cnf-r3-b1-k1.1-comp.cnf | 10000 | 494.92 | 0.93 | 2.12 |
| cnf-r3-b1-k1.2-comp.cnf | 10000 | 10000 | 4.06 | 3.7 |
| cnf-r3-b2-k1.1-comp.cnf | 309.99 | 91.96 | 0.64 | 0.54 |

| | | | | |
|---|---|---|---|---|
| cnf-r3-b2-k1.2-comp.cnf | 3195.13 | 633.55 | 0.54 | 0.44 |
| cnf-r3-b3-k1.1-comp.cnf | 479.91 | 953.27 | 0.57 | 0.23 |
| cnf-r3-b3-k1.2-comp.cnf | 121.84 | 1488.47 | 0.6 | 2.3 |
| dubois100.cnf | 7.9 | 0.07 | 0.08 | 1.02 |
| dubois27.cnf | 0.12 | 0.01 | 0.01 | 0.01 |
| dubois28.cnf | 0.13 | 0.02 | 0 | 0 |
| dubois29.cnf | 0.15 | 0 | 0.01 | 0.01 |
| dubois30.cnf | 0.13 | 0.01 | 0.01 | 0.01 |
| dubois50.cnf | 0.72 | 0.02 | 0.02 | 0.01 |
| e0ddr2-10-by-5-1.cnf | 102.13 | 73.67 | 8.08 | 6.78 |
| e0ddr2-10-by-5-4.cnf | 48.65 | 1.01 | 4.08 | 1.21 |
| enddr2-10-by-5-1.cnf | 52.84 | 2.81 | 11.92 | 1.63 |
| enddr2-10-by-5-8.cnf | 48.45 | 1.1 | 2.57 | 0.89 |
| ewddr2-10-by-5-1.cnf | 55.17 | 1.19 | 12.22 | 1.13 |
| ewddr2-10-by-5-8.cnf | 57.93 | 1.17 | 2.64 | 1.21 |
| f7hh.14.cnf | 44.62 | 81.46 | 1.52 | 2.78 |
| f7hh.15.cnf | 76.92 | 77.67 | 2.48 | 1.21 |
| f8h_10.cnf | 4.52 | 0.2 | 0.28 | 0.22 |
| f8h_11.cnf | 6.34 | 0.22 | 0.43 | 0.37 |
| facts7h.10.cnf | 3.12 | 0.16 | 0.22 | 0.18 |
| facts7h.cnf | 5.72 | 0.19 | 0.41 | 0.21 |
| facts7hh.12.cnf | 152.95 | 13.76 | 1.5 | 4.63 |
| facts7hh.13.cnf | 38.21 | 2.52 | 1.37 | 0.85 |
| facts7hh.13.simple.cnf | 20.14 | 2.35 | 0.82 | 1.26 |
| facts7hh.13-simp.cnf | 20.04 | 12.12 | 0.7 | 0.65 |
| facts7hh.13-simp.simple.cnf | 18.06 | 11.83 | 0.69 | 0.72 |
| facts7hha.12.cnf | 47.23 | 5.89 | 0.93 | 0.45 |
| facts7hha.13.cnf | 12.82 | 20.34 | 0.79 | 2.31 |
| facts8.13.cnf | 23.87 | 0.39 | 0.77 | 0.15 |
| facts8h.12.cnf | 10.42 | 0.28 | 0.54 | 0.26 |
| hfo3l.011.0.cnf | 21.55 | 0.15 | 0.34 | 0.37 |
| hfo3l.012.0.cnf | 5.54 | 0.17 | 0.24 | 0.19 |
| hfo4.001.0.cnf | 10000 | 59.48 | 204.5 | 219.32 |
| hfo4.002.1.cnf | 10000 | 2.16 | 61.9 | 46.37 |
| hfo4.034.1.cnf | 3640.54 | 37.74 | 43.13 | 32.6 |
| hfo4.035.1.cnf | 10000 | 28.79 | 90.82 | 32.7 |
| hfo4.036.0.cnf | 10000 | 86.51 | 144.97 | 128.4 |
| hfo4.037.0.cnf | 10000 | 102.44 | 136.64 | 118.76 |
| hfo4l.001.1.cnf | 2.17 | 0.05 | 0.14 | 0.1 |
| hfo4l.002.1.cnf | 3.86 | 0.07 | 0.28 | 0.05 |
| hfo4l.039.1.cnf | 0.9 | 0.01 | 0.14 | 0.2 |
| hfo4l.040.1.cnf | 0.35 | 0.05 | 0.07 | 0.12 |
| hfo5.001.0.cnf | 10000 | 79.06 | 122.04 | 250.05 |
| hfo5.002.0.cnf | 10000 | 78.2 | 166.31 | 149.3 |
| hfo5l.001.0.cnf | 6.41 | 0.18 | 0.21 | 0.15 |
| hfo5l.002.0.cnf | 5.12 | 0.15 | 0.19 | 0.2 |
| hfo5l.039.1.cnf | 0.28 | 0.04 | 0.03 | 0.01 |
| hfo5l.040.0.cnf | 5.18 | 0.24 | 0.23 | 0.3 |
| hfo6l.039.1.cnf | 0.14 | 0.11 | 0.07 | 0.05 |

| | | | | |
|---|---|---|---|---|
| hfo6l.040.1.cnf | 0.53 | 0.04 | 0.04 | 0.05 |
| hole10.cnf | 10000 | 69.65 | 36.04 | 36.5 |
| hole6.cnf | 0.3 | 0.04 | 0.01 | 0.01 |
| hole7.cnf | 3.21 | 0.11 | 0.32 | 0.24 |
| hole8.cnf | 18.4 | 5.4 | 0.95 | 1.05 |
| hole9.cnf | 1498.34 | 6.44 | 5.49 | 4.23 |
| huge.cnf | 0.23 | 0.05 | 0.09 | 0.01 |
| ii16a1.cnf | 11.4 | 10000 | 0.33 | 2.56 |
| ii16a2.cnf | 4.41 | 0.32 | 0.35 | 0.45 |
| ii16b1.cnf | 10000 | 10000 | 59.92 | 320.56 |
| ii16b2.cnf | 62.69 | 6.85 | 6.59 | 7.6 |
| ii16c1.cnf | 6.17 | 0.18 | 0.36 | 0.15 |
| ii16c2.cnf | 1.94 | 0.17 | 0.36 | 0.25 |
| ii16d1.cnf | 2.83 | 10000 | 0.25 | 3.21 |
| ii16d2.cnf | 0.46 | 0.2 | 0.21 | 0.15 |
| ii16e1.cnf | 10.71 | 0.17 | 0.55 | 0.42 |
| ii16e2.cnf | 2.07 | 0.09 | 0.31 | 0.12 |
| ii32a1.cnf | 0.2 | 0.11 | 0.23 | 0.07 |
| ii32b1.cnf | 0.04 | 0.02 | 0.02 | 0.01 |
| ii32b2.cnf | 0.08 | 0.05 | 0.09 | 0.05 |
| ii32b3.cnf | 0.15 | 0.09 | 0.2 | 0.1 |
| ii32b4.cnf | 0.3 | 0.09 | 0.28 | 0.1 |
| ii32c1.cnf | 0.03 | 0.02 | 0.03 | 0.01 |
| ii32c2.cnf | 0.09 | 0.04 | 0.06 | 0 |
| ii32c3.cnf | 0.09 | 0.09 | 0.11 | 0 |
| ii32c4.cnf | 0.79 | 0.38 | 0.76 | 0.43 |
| ii32d1.cnf | 0.06 | 0.02 | 0.06 | 0.04 |
| ii32d2.cnf | 0.28 | 0.9 | 0.37 | 1.2 |
| ii32d3.cnf | 0.52 | 0.87 | 0.41 | 0.5 |
| ii32e1.cnf | 0.04 | 0.01 | 0.04 | 0 |
| ii32e2.cnf | 0.07 | 0.04 | 0.05 | 0.01 |
| ii32e3.cnf | 0.12 | 0.15 | 0.1 | 0.2 |
| ii32e4.cnf | 0.17 | 0.1 | 0.14 | 0.07 |
| ii32e5.cnf | 0.17 | 0.14 | 0.46 | 0.12 |
| jnh1.cnf | 0.06 | 0 | 0.03 | 0.01 |
| jnh2.cnf | 0.05 | 0 | 0.02 | 0.03 |
| jnh3.cnf | 0.18 | 0.03 | 0.04 | 0.01 |
| jnh4.cnf | 0.06 | 0.02 | 0.03 | 0 |
| jnh5.cnf | 0.04 | 0.02 | 0.02 | 0.23 |
| jnh6.cnf | 0.12 | 0.02 | 0.03 | 0.02 |
| jnh7.cnf | 0.05 | 0.02 | 0.02 | 0.03 |
| jnh8.cnf | 0.04 | 0.01 | 0.02 | 0.05 |
| jnh9.cnf | 0.04 | 0.02 | 0.03 | 0.07 |
| logistics.a.cnf | 7.89 | 0.82 | 0.14 | 0.23 |
| logistics.b.cnf | 10.53 | 226.21 | 0.19 | 0.12 |
| logistics.c.cnf | 45.33 | 93.2 | 0.39 | 0.48 |
| longmult0.cnf | 0.02 | 0.01 | 0.02 | 0.01 |
| longmult1.cnf | 0.04 | 0.03 | 0.04 | 0 |
| longmult10.cnf | 10000 | 528.3 | 477.62 | 482 |

| | | | | |
|---|---|---|---|---|
| longmult11.cnf | 10000 | 907 | 688.37 | 702.4 |
| longmult12.cnf | 10000 | 973.64 | 602.05 | 10000 |
| longmult13.cnf | 10000 | 1255.16 | 803.06 | 624.35 |
| longmult14.cnf | 10000 | 1224.12 | 682.91 | 824.15 |
| longmult15.cnf | 10000 | 1614.31 | 632.7 | 700.5 |
| longmult2.cnf | 0.16 | 0.07 | 0.05 | 0.12 |
| longmult3.cnf | 0.79 | 11.05 | 0.08 | 2.05 |
| longmult4.cnf | 4.29 | 38.89 | 0.18 | 0.24 |
| longmult5.cnf | 23.41 | 60.76 | 0.85 | 1.4 |
| longmult6.cnf | 151.45 | 86.33 | 4.76 | 6.42 |
| longmult7.cnf | 2129.03 | 180.9 | 37.52 | 25.34 |
| longmult8.cnf | 10000 | 218.71 | 212.54 | 198.64 |
| longmult9.cnf | 10000 | 490.97 | 359.74 | 671.75 |
| medium.cnf | 0.01 | 0.01 | 0.02 | 0.01 |
| misg.boehm.cnf | 0.03 | 35.04 | 0.02 | 0.01 |
| mul04.boehm.cnf | 0.03 | 0.01 | 0.01 | 0 |
| mul05.boehm.cnf | 0.12 | 0.05 | 0.04 | 0.01 |
| mul06.boehm.cnf | 0.79 | 0.17 | 0.08 | 0.05 |
| mul07.boehm.cnf | 5.27 | 0.68 | 0.27 | 0.14 |
| mul08.boehm.cnf | 24.31 | 2.65 | 0.93 | 0.54 |
| par16-1.cnf | 340.02 | 9.6 | 1.46 | 0.72 |
| par16-1-c.cnf | 184.93 | 2.44 | 0.66 | 0.23 |
| par16-2.cnf | 195.38 | 15.81 | 2.74 | 0.46 |
| par16-2-c.cnf | 3251.1 | 34.66 | 4.16 | 2.98 |
| par16-3.cnf | 45.46 | 12.18 | 12.22 | 6.23 |
| par16-3-c.cnf | 7.84 | 35.93 | 2.48 | 1.24 |
| par16-4.cnf | 70.02 | 1.45 | 2.53 | 3.21 |
| par16-4-c.cnf | 216.24 | 44.99 | 0.62 | 7.64 |
| par16-5.cnf | 8385.03 | 2.42 | 1.25 | 3.14 |
| par16-5-c.cnf | 6014.45 | 71.4 | 0.88 | 0.69 |
| par8-1.cnf | 0.04 | 0.02 | 0.01 | 0.01 |
| par8-1-c.cnf | 0.01 | 0 | 0.02 | 0 |
| par8-2.cnf | 0.02 | 0 | 0.02 | 0.01 |
| par8-2-c.cnf | 0 | 0 | 0.01 | 0.01 |
| par8-3.cnf | 0.03 | 0.01 | 0.02 | 0.01 |
| par8-3-c.cnf | 0 | 0 | 0 | 0 |
| par8-4.cnf | 0.02 | 0.02 | 0.02 | 0.01 |
| par8-5-c.cnf | 0.02 | 0.01 | 0.01 | 0.01 |
| pitch.boehm.cnf | 0.12 | 0.08 | 0.1 | 0.05 |
| pret150_25.cnf | 0.66 | 855.31 | 0.05 | 2.87 |
| pret150_40.cnf | 0.66 | 2.66 | 0.03 | 0.05 |
| pret150_60.cnf | 0.66 | 74.3 | 0.03 | 0.02 |
| pret150_75.cnf | 0.65 | 1.26 | 0.04 | 0.01 |
| pret60_25.cnf | 0.05 | 0.03 | 0.01 | 0.01 |
| pret60_40.cnf | 0.06 | 0.02 | 0.01 | 0.01 |
| pret60_60.cnf | 0.06 | 0.02 | 0.01 | 0.05 |
| qg1-07.cnf | 3.79 | 0.42 | 1.04 | 2.05 |
| qg1-08.cnf | 10000 | 43.69 | 128.12 | 74.92 |
| qg2-07.cnf | 7.28 | 0.41 | 1.04 | 0.52 |

| | | | | |
|---|---|---|---|---|
| qg2-08.cnf | 10000 | 10.61 | 82.09 | 9.32 |
| qg3-08.cnf | 30.63 | 0.12 | 0.19 | 0.24 |
| qg3-09.cnf | 10000 | 8.24 | 321.42 | 112.64 |
| qg4-08.cnf | 69.04 | 0.23 | 0.63 | 0.34 |
| qg4-09.cnf | 72.18 | 0.1 | 0.36 | 0.26 |
| qg5-09.cnf | 30.31 | 0.19 | 0.39 | 0.22 |
| qg5-10.cnf | 689.23 | 0.33 | 0.91 | 0.37 |
| qg5-11.cnf | 3501.46 | 0.38 | 0.93 | 0.35 |
| qg5-12.cnf | 10000 | 1.58 | 3.17 | 2.85 |
| qg5-13.cnf | 10000 | 50.08 | 235.32 | 36.19 |
| qg6-09.cnf | 4.23 | 0.13 | 0.3 | 0.2 |
| qg6-10.cnf | 44.34 | 0.26 | 0.55 | 0.29 |
| qg6-11.cnf | 1148.6 | 1 | 1.81 | 1.24 |
| qg6-12.cnf | 10000 | 13.49 | 38.52 | 14.6 |
| qg7-09.cnf | 0.93 | 0.12 | 0.28 | 0.12 |
| qg7-10.cnf | 22.51 | 0.24 | 0.56 | 0.26 |
| qg7-11.cnf | 920 | 0.72 | 2.35 | 1.01 |
| qg7-12.cnf | 10000 | 4.02 | 15.75 | 5.24 |
| qg7-13.cnf | 10000 | 201.25 | 10.16 | 7.64 |
| queueinvar10.cnf | 10.47 | 12.19 | 0.51 | 0.42 |
| queueinvar12.cnf | 24.42 | 63.72 | 0.92 | 0.45 |
| queueinvar14.cnf | 42.41 | 566.55 | 1.32 | 0.4 |
| queueinvar16.cnf | 52.08 | 941.52 | 0.98 | 0.5 |
| queueinvar18.cnf | 234.26 | 10000 | 5.33 | 1.25 |
| queueinvar2.cnf | 0.02 | 0.01 | 0.01 | 0.01 |
| queueinvar20.cnf | 397.75 | 10000 | 6.01 | 3.38 |
| queueinvar4.cnf | 0.11 | 0.06 | 0.03 | 0.02 |
| queueinvar6.cnf | 0.95 | 0.56 | 0.1 | 0.5 |
| queueinvar8.cnf | 1.76 | 1.57 | 0.18 | 0.12 |
| rocket_ext.a.cnf | 0.93 | 0.09 | 0.07 | 0.05 |
| rocket_ext.b.cnf | 0.68 | 0.12 | 0.07 | 0.11 |
| ssa0432-003.cnf | 0.04 | 0.02 | 0.03 | 0 |
| ssa2670-130.cnf | 0.46 | 4556.57 | 0.08 | 0.05 |
| ssa2670-141.cnf | 1.5 | 3951.07 | 0.13 | 0.07 |
| ssa6288-047.cnf | 0.35 | 0.24 | 0.49 | 0.21 |
| ssa7552-038.cnf | 0.09 | 0.04 | 0.05 | 0.02 |
| ssa7552-158.cnf | 0.08 | 0.03 | 0.03 | 0.01 |
| ssa7552-159.cnf | 0.07 | 0.03 | 0.03 | 0.01 |
| ssa7552-160.cnf | 0.09 | 0.05 | 0.05 | 0.02 |
| sw100-100-8-0.cnf | 0.21 | 0.01 | 0.03 | 0 |
| sw100-10-8-0.cnf | 0.25 | 0.02 | 0.04 | 0.01 |
| sw100-1-8-0.cnf | 0.22 | 0.01 | 0.03 | 0 |
| sw100-84-8-1.cnf | 0.22 | 0.02 | 0.04 | 0.01 |
| sw100-85-8-1.cnf | 0.24 | 0.02 | 0.03 | 0.01 |
| sw100-86-8-1.cnf | 0.22 | 0.02 | 0.05 | 0.01 |
| sw100-87-8-1.cnf | 0.24 | 0.02 | 0.03 | 0 |
| ucsc-bf0432-015.cnf | 1.24 | 5.25 | 0.23 | 0.02 |
| ucsc-bf0432-017.cnf | 1.09 | 15.07 | 0.24 | 0.21 |
| ucsc-bf2670-532.cnf | 0.33 | 1718.95 | 0.06 | 0.23 |

| | | | | |
|---|---|---|---|---|
| ucsc-bf2670-533.cnf | 0.27 | 65.03 | 0.07 | 0.42 |
| ucsc-bf2670-538.cnf | 0.16 | 0.05 | 0.06 | 0.07 |
| ucsc-bf2670-546.cnf | 0.3 | 103.27 | 0.08 | 32 |
| ucsc-bf2670-547.cnf | 0.26 | 8132.58 | 0.08 | 14.21 |
| ucsc-ssa0432-001.cnf | 0.04 | 0.04 | 0.01 | 0.05 |
| ucsc-ssa2670-127.cnf | 0.46 | 4535.71 | 0.09 | 0.12 |
| ucsc-ssa2670-128.cnf | 0.45 | 10000 | 0.1 | 0.02 |
| z9sym.boehm.cnf | 0.35 | 10000 | 0.01 | 0.7 |
| ztwaalf1.boehm.cnf | 0.03 | 0 | 0.01 | 0.02 |
| ztwaalf2.boehm.cnf | 0.02 | 0 | 0.01 | 0.02 |

**表 3 最终生成的 QuickSAT 与现成的 GRASP、SATO、zChaff 之间的比较**

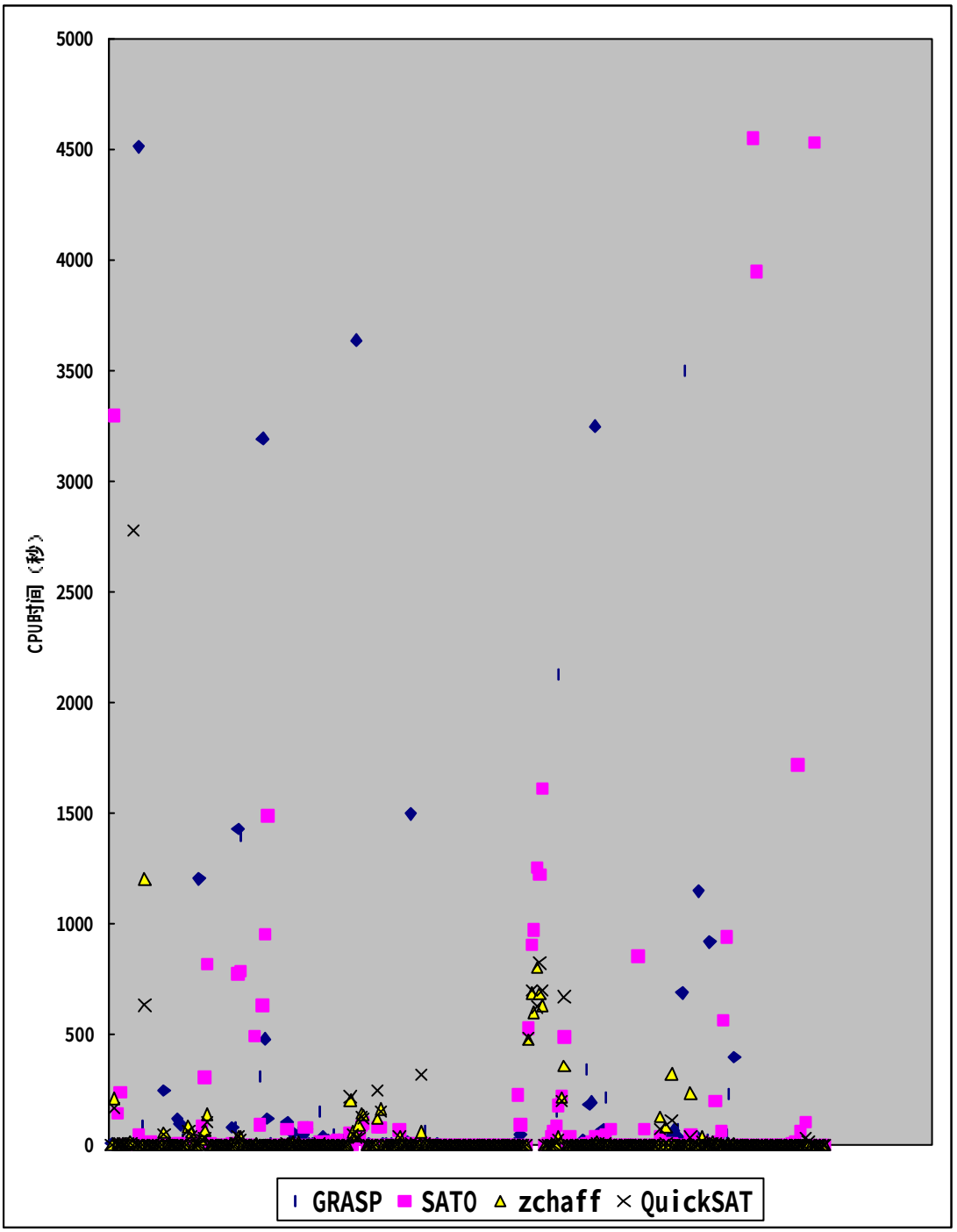



图 3 表 3 的数据的散点示意图

# 致 谢

本文得以顺利完成，除了我自己的努力外，离不开许多人的指导、关心和帮助。在此，我要向他们表示衷心的感谢。

首先要感谢的是我的导师徐良贤教授。没有徐老师对我的悉心指导和孜孜教诲，我不可能完成这篇论文。正是徐老师高屋建瓴的点拨和对我充满信任的支持，才使我得以在遇到困难的时候站稳脚跟，坚持了过来，最终把问题一一解决。此外，徐老师不但是一位良师，同时也是一位充满爱心的亲切的长辈，使我在生活上也能感受到他的温暖。

其次要感谢的是我们系的另外一位德高望重的教授，那就是陆汝占老师。他曾经一针见血地指出我的工作的不足，让我受益无穷。

要感谢的人还有参加过我开题答辩的陆鑫达教授、在我研究中给予我极大帮助的博士师兄肖正光、曹大军和吴尉林，还有师姐毛家菊；另外我的同学韩晓峰、郭荣等也在我的工作中给过我不少帮助和支持。此外，对那些在我学习、生活上给予过帮助的同学，包括江济、王波、戴欣、张辉等，我也要说一声：谢谢！

最后，我再次向所有曾经给予过我帮助和支持的人们表示衷心的感谢。

# 攻读硕士学位期间发表的学术论文

[1] 焦加麟，韩晓峰，徐良贤，算法工程的思想、方法与工具——以 SAT 算法为研究实例，计算机科学，已录用（2002 年 12 月）

[2] 焦加麟，徐良贤，戴克昌，人工智能在智能教学系统中的应用，计算机仿真，已录用（2002 年 11 月）