ПРОБЛЕМА ВЫПОЛНИМОСТИ
(современные алгоритмы решения)

Дьяконов А.Г.

Это учебное пособие разработано для поддержки курса, который читается для магистров факультета ВМК МГУ. Материал актуален по состоянию на дату составления (2006г.), после этого в текст вносились лишь поправки редакционного характера. По стилю – это конспект лекций, который не представляет особой ценности без прослушивания всего курса. Все замечания и предложения по улучшению текста можно слать автору на почту djakonov (a) mail (point) ru. С другими работами автора можно познакомиться на сайте http://alexanderdyakonov.narod.ru/ (в разделе «Скачать»).

Москва, 2006 г.

Терминология
Проблема выполнимости (ВЫП или SAT) –
для данной булевой формулы определить, существует ли набор, на котором она обращается в единицу. Такой набор называется *выполняющим*. Если он существует, то формула называется *выполнимой*[1].

Далее проблема выполнимости будет рассматриваться только для формул специального вида: конъюнктивных нормальных форм (КНФ).

Вообще, если говорить о терминологии, то в современной литературе выделяют более широкую проблему: CSP (Constraint Satisfaction Problem) – *выполнимость при ограничениях* (могут ли выполняться одновременно ограничения, накладываемые на переменные). Однако, именно задача выполнимости является центральной в этой проблеме, кроме того имеет много хорошо разработанных и исследованных методов решения (достаточно общих и специализированных).

Приложения

- Автоматическая генерация тестов (ATPG - Automatic Test Patten Generation)

[Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6{22, January 1992],
[P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design*, 15(9):1167-1176, September 1996]

- Проверка эквивалентности схем (combinational circuit equivalence checking)

[E. I. Goldberg, M. R. Prasad, and R. K. Brayton. Using SAT for combinational equivalence checking. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe (DATE)*, 2001],
[Joao P. Marques-Silva and Thomas Glass. Combinational equivalence checkingusing satisfiability and recursive learning. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe (DATE)*, 1999]

- Проверка последовательности (sequential property checking)

[Mary Sheeran, Satnam Singh, and Gunnar Stalmark. Checking safety properties using induction and a SAT-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, 2000]

- Проверка микропроцессоров (microprocessor verification)

[M.N. Velev and R.E. Bryant. Efective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Proceedings of the Design Automation Conference (DAC)*, June 2001]

[1] Часто проблему выполнимости формулируют в «более практической» форме: для данной булевой формулы *найти* выполняющий набор или доказать, что его не существует.

- Проверка моделей (model checking)

[Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, 1999],
[Ken L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proceedings of 14th Conference on Computer-Aided Verification (CAV2002)*. Springer Verlag, 2002]

- Анализ достижимости (reachability analysis)

[Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. SAT-based image computation with application in reachability analysis. In *Proceedings of Third International Conference Formal Methods in Computer-Aided Design (FMCAD 2000)*, 2000],
[Parosh Aziz Abdulla, Per Bjesse, and Niklas Een. Symbolic reachability analysis based on SAT-solvers. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*, 2000]

- Устранение избыточности (redundancy removal)

[Joonyoung Kim, Joao P. Marques Silva, Hamid Savoj, and Karem A. Sakallah. RID-GRASP: Redundancy identification and removal using GRASP. In *IEEE/ACM International Workshop on Logic Synthesis*, 1997]

- Временной анализ (timing analysis)

[Joao P. Marques-Silva and Karem A. Sakallah. Eficient and robust test generation-based timing analysis. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, May 1994]

- Маршрутизация (routing)

[G. Nam, K. A. Sakallah, and R. A. Rutenbar. Satisfiability-based layout revisited: Routing complex FPGAs via search-based Boolean SAT. In *Proceedings of International Symposium on FPGAs*, Feburary 1999]

Покажем на модельных примерах принцип приложения задач о выполнимости. Допустим мы строим программную систему, которая должна уметь проверять функциональную эквивалентность программ, записанных на каком-то языке программирования. В частности, решать вопрос об эквивалентности следующих двух фрагментов кода.

<table>
<tr><td><pre>if(!a && !b) h();
else if(!a) g();
else f();</pre></td><td><pre>if(a) f();
else if(b) g();
else h();</pre></td></tr>
</table>

Произведём преобразование этих фрагментов по правилу

$$\texttt{compile(if x then y else z)} = (x\wedge y)\vee(\neg x\wedge z).$$

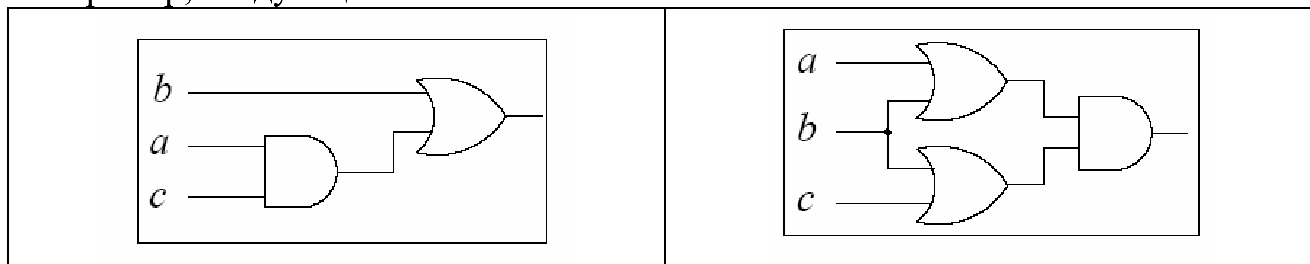
$(\neg a\wedge\neg b)\wedge h \vee \neg(\neg a\wedge\neg b)\wedge (\neg a\wedge g \vee a\wedge f)$	$a\wedge f \vee \neg a\wedge(b\wedge g \vee \neg b\wedge h)$

Тогда задача сводится к проверке выполнимости формулы

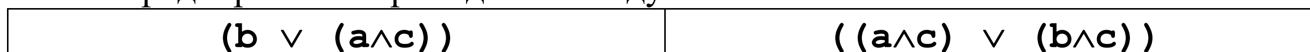
(¬a∧¬b)∧h∨¬(¬a∧¬b)∧(¬a∧g∨a∧ f) != a∧ f∨¬a∧(b∧g∨¬b∧h),
которую можно представить и в виде КНФ.

Задание. Представьте эту формулу в виде КНФ.

Аналогично можно решать вопрос о функциональной эквивалентности схем. Например, следующие схемы



после предобработки приводятся к виду

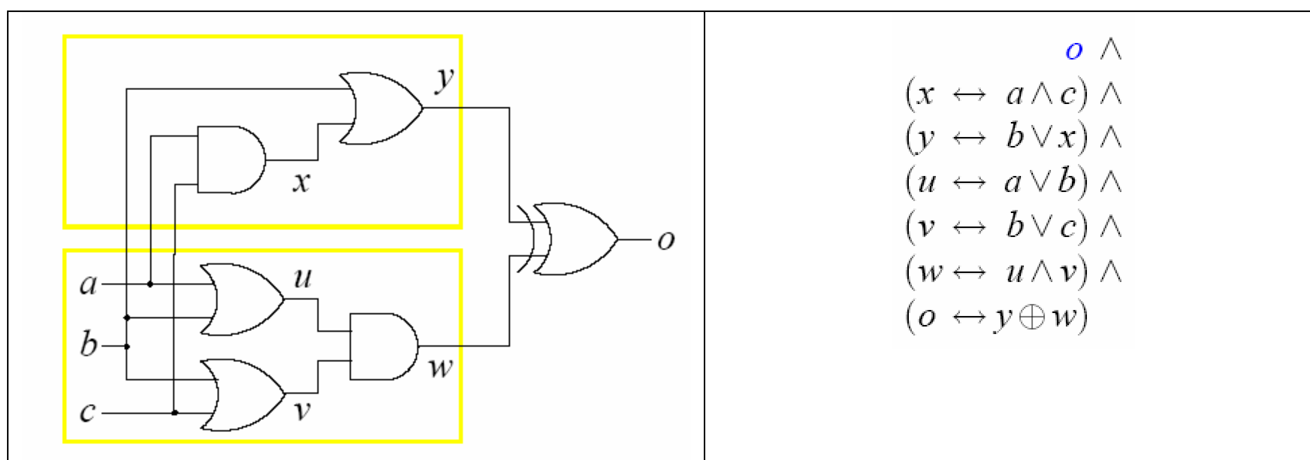


(b ∨ (a∧c))	((a∧c) ∨ (b∧c))

Задача свелась к проверки на выполнимость формулы

(b ∨ (a∧c)) != ((a∧c) ∨ (b∧c)).

Алгоритмы для представления сложных систем в виде булевых формул специального вида – отдельная большая тема. Приведём лишь иллюстрации подобного представления для схем из функциональных элементов.



$$
\begin{aligned}
x \leftrightarrow \overline{y} \;&\Leftrightarrow\; (x \rightarrow \overline{y}) \wedge (\overline{y} \rightarrow x) \\
&\Leftrightarrow\; (\overline{x} \vee \overline{y}) \wedge (y \vee x)
\end{aligned}
$$

$$
\begin{aligned}
x \leftrightarrow (y \vee z) \;&\Leftrightarrow\; (y \rightarrow x) \wedge (z \rightarrow x) \wedge (x \rightarrow (y \vee z)) \\
&\Leftrightarrow\; (\overline{y} \vee x) \wedge (\overline{z} \vee x) \wedge (\overline{x} \vee y \vee z)
\end{aligned}
$$

$$
\begin{aligned}
x \leftrightarrow (y \wedge z) \;&\Leftrightarrow\; (x \rightarrow y) \wedge (x \rightarrow z) \wedge ((y \wedge z) \rightarrow x) \\
&\Leftrightarrow\; (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{(y \wedge z)} \vee x) \\
&\Leftrightarrow\; (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z} \vee x)
\end{aligned}
$$

$$
\begin{aligned}
x \leftrightarrow (y \leftrightarrow z) \;&\Leftrightarrow\; (x \rightarrow (y \leftrightarrow z)) \wedge ((y \leftrightarrow z) \rightarrow x) \\
&\Leftrightarrow\; (x \rightarrow ((y \rightarrow z) \wedge (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x) \\
&\Leftrightarrow\; (x \rightarrow (y \rightarrow z)) \wedge (x \rightarrow (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x) \\
&\Leftrightarrow\; (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \leftrightarrow z) \rightarrow x) \\
&\Leftrightarrow\; (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (((y \wedge z) \vee (\overline{y} \wedge \overline{z})) \rightarrow x) \\
&\Leftrightarrow\; (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \wedge z) \rightarrow x) \wedge ((\overline{y} \wedge \overline{z}) \rightarrow x) \\
&\Leftrightarrow\; (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (\overline{y} \vee \overline{z} \vee x) \wedge (y \vee z \vee x)
\end{aligned}
$$

a, b — AND — c: $(a \vee \neg c)(b \vee \neg c)(\neg a \vee \neg b \vee c)$

a, b — OR — c: $(\neg a \vee c)(\neg b \vee c)(a \vee b \vee \neg c)$

a, b — XOR — c: $(\neg a \vee \neg b \vee \neg c)(\neg a \vee b \vee c)(a \vee \neg b \vee c)(a \vee b \vee \neg c)$

Проблема выполнимости является NP-полной, потому не существует (пока?) полиномиального алгоритма для её решения. Тем не менее, существует достаточно много алгоритмов для решения проблемы выполнимости за приемлемое время на реальных данных.

Современные SAT-солверы[2] могут определять выполнимость КНФ от 1000 переменных за несколько минут. КНФ, выполнимость которых надо определять на практике (их часто называют *индустриальными*: industrial) имеют специальный вид. Случайные КНФ (сгенерированные с помощью специальных программ, использующих датчики псевдослучайных чисел) имеют более сложную структуру. Для них 3SAT-проблему[3] не удаётся решить при числе переменных более 700.

---

[2] Так мы будем называть программные реализации алгоритмов, предназначенные для решения прикладных задач и оценки эффективности алгоритмов путём сравнения с другими программами.

[3] Проблема выполнимости для КНФ, в каждой скобке которых содержится не более трёх литералов (см. дальше).

Классификация алгоритмов:

1. *Неполные* (incomplete).

Алгоритмы, которые осуществляют поиск выполняющего набора неполным перебором пространства возможных решений. Как правило, используют локальный поиск (local search) для нахождения решения. Если алгоритм не находит решения, то это не означает, что его не существует. Таким образом, неполные алгоритмы могут доказать только выполнимость (и то не всегда) и не могут доказать невыполнимость[4]. Основное достоинство таких алгоритмов – быстрая скорость работы.

```
GSAT_Solver()
{
for i = 1 to MAX_TRIES
{
T = a randomly generated assignment
for j = 1 to MAX_FLIPS
{
if (no unsatisfied clauses exist)
return T
else
x = find variable in T with max score
if (score of x is smaller than 0)
break;
else
flip the assignment of x
}
}
return No Satisfying assignment found
}
}
```

GSAT предложен Селманом, Левескью и Мичелом в

[Bart Selman, Hector Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 459-465, 1992].

GSAT стал первым солвером, основанном на стохастическом локальном поиске (Stochastic Local Search), который решал сложные задачи с большим числом переменных и клауз.

WalkSAT предложен МакАлестором, Селманом и Каутсом в [David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 321-326, Providence,

---

[4] В последнее время разрабатывают также полные алгоритмы, основанные на локальном поиске (точнее: алгоритмы, доказывающие невыполнимость). Часто для этих алгоритмов исходная задача записывается в других терминах («переформулируется»). При этом основная проблема – компактная запись новой переформулированной задачи. Подобные алгоритмы пока применимы только для «очень маленьких» задач.

Rhode Island, 1997.]. Отличие от GSAT в выборе переменной. В WalkSAT рейтинг переменной равен числу клауз, которые перестанут быть выполнимы при её изменении. Если текущий набор не является искомым, выбирается невыполнимая клауза (случайно). Меняется значение одной из её переменных с учётом рейтинга (например, если в ней есть переменная с нулевым рейтингом, то меняется её значение).

В настоящее время существует множество модернизаций этих алгоритмов. Основная их цель – избежать «застревания» алгоритма в локальных минимумах[5]. Например, в локальном минимуме увеличивать веса невыполнимых клауз, «вносить шум» в функционал, более того, регулировать уровень шума в процессе перебора.

Вообще, неполные алгоритмы это отдельная большая тема, которой, впрочем, здесь не будет уделено достойного внимания. Основная причина – их эвристическая природа, подробное описание в литературе по смежным областям, отсутствие «универсальных подходов». Это «очень знакомые методы»: метод отжига, метод случайных блужданий, генетические алгоритмы и т.д. См. [Дьяконов А.Г. Анализ данных, обучение по прецедентам, логические игры, системы WEKA, RapidMiner и MatLab// учебное пособие http://www.machinelearning.ru/wiki/images/7/7e/Dj2010up.pdf]

Основное достоинство неполных методов – их приложение и к другим задачам, тесно связанным с SAT. Например, MAXSAT (найти набор, который выполняет максимальное число клауз в заданной КНФ). Вообще, почти все эти методы пытаются минимизировать некоторый функционал (часто даже не учитывая его структуру).

Пример алгоритма RANGER (RANdomised GEneral Resolution), доказывающего невыполнимость и основанного на локальном поиске [Steven Prestwich and Ines Lynce Local Search for Unsatisfiability].

```
RANGER(F, p, p1, p2, p3, w, k)
{
i = 1;
F[1] = (any k clauses from F);
while F[i] does not contain the empty clause
  with probability p
    replace a random F[i] clause by a random F
clause
  otherwise
    resolve random F[i] clauses c, c0 giving r
    if r is non-tautologous and |r|<=w
       with probability p2
```

[5] В локальных минимумах функционала качества решения. Его можно определить как число невыполненных клауз на данном наборе.

```
        if |r|<=max(|c|,|c0|) replace the longer of
c,c0 by r
        otherwise
        replace a random F[i] clause by r
        with probability p3
          apply    any    satisfiability-preserving
transformation to F, F[i]
        i = i + 1; F[i+1]<-the new formula from F
return UNSATISFIABLE
}
```

Современные солверы, основанные на локальном поиске: AdaptNovelty+, R+ AdaptNovelty+, unitwalk.

2. *Полные* (complete).
Алгоритмы, которые выполняют полный перебор. Как правило, базируются на DPLL-методе. Часто выделяют две основные категории полных алгоритмов: алгоритмы, основанные на резолюциях, алгоритмы, основанные на «переборе дерева».

Заметим, что именно доказательство невыполнимости[6] (которое осуществляют полные алгоритмы) имеет важное прикладное значение при автоматическом доказательстве теорем в ИИ (automated theorem proving) и верификации схем (circuit verification).

Примеры: rel_sat, GRASP, SATO, Chaff, BerkMin.

Дэвис (Davis) и Путнэм (Putnam) предложили алгоритм для решения проблемы выполнимости в 1960 году [Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201-215, July 1960]. Однако у него имелся существенный недостаток: переполнение памяти. Модифицированная версия Дэвиса, Логемана (Logemann) и Ловелэнда (Loveland) появилась в 1962 году [Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394-397, July 1962]. В этой версии вместо резолюций использовался поиск. Этот алгоритм часто называют DLL или DPLL, а первый – DP. Подробнее о резолюциях и поиске будет сказано ниже. Заметим, что несмотря на недостатки резолюционных алгоритмов как таковых, минимальные резолюционные решения проблемы экспоненциально короче DPLL-решений (см. [E. Ben-Sasson, R. Impagliazzo, A. Wigderson. Near-Optimal Separation of Treelike and General Resolution. Combinatorica vol. 24 no. 4, 2004, pp. 585-603.]).

Часто солверы, в которых реализован алгоритм DPLL, делят на две группы: Conflict-driven (minisat, vallst, zChaff) и look-ahead (kcnfs, march, OKsolver).

[6] Не всегда это доказательство эффективно представляется в аналитическом виде.

Подробнее о них дальше. Отметим, что солверы, основанные на локальном поиске, показывают хорошие результаты на случайных выполнимых 3-КНФ, Conict-driven-солверы – на индустриальных КНФ, look-ahead-солверы – на невыполнимых КНФ.

Переборные алгоритмы, основанные на других идеях, значительно уступают этим. Их описания можно найти в

- Binary Decision Diagrams [Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions in Computers*, 8(35):677{691, 1986],
- Stalmarck's algorithm [G. Stalmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Technical report, European Patent N 0403 454 (1995), US Patent N 5 276 897, Swedish Patent N 467 076 (1989), 198] , [Mary Sheeran and Gunnar Stºalmark. A tutorial on Stºalmark's proof procedure for propositional logic. *Formal Methods in System Design*, 16:23{58, January 2000]
- local search and random walk [Jun Gu. Local search for satisfiability SAT problem. *IEEE Transactions on Systems and Cybernetics*, 23(3):1108{1129, 1993], [Bart Selman, Hector Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 459{465, 1992].

Определения и обозначения

Литерал – переменная или её отрицание: x_i или $\bar{x}_i$.
Клауза – дизъюнкция конечного числа литералов.

При решении практических задач, естественно допускаются клаузы вида: $x_1 \vee x_1$ (повторение литералов), $x_1 \vee \bar{x}_1$ (взаимообратные литералы). В первом случае один из литералов можно удалить (это не влияет на выполнимость/ невыполнимость КНФ), во втором – клауза тождественно равна единице и её можно удалить из КНФ. Поэтому для удобства будем считать, что клаузы являются элементарными дизъюнкциями[7]:

$$x_{i_1}^{\sigma_1} \vee \ldots \vee x_{i_N}^{\sigma_N},\ 1 \le i_1, \ldots, i_N \le n,\ |\{i_1, \ldots, i_N\}| = N.$$

КНФ – конъюнкция конечного числа клауз. Будем считать, что заданная КНФ – $D_1 \vee \ldots \vee D_k$. При этом $|D_j|$ - число литералов в клаузе D_j, $y \in D_j$ – предикат вхождения литерала y в клаузу D_j.

Пример

$$K_1 = (x_1 \vee x_2)\,\&\,(\bar{x}_2 \vee x_3)\,\&\,(\bar{x}_1 \vee \bar{x}_3)\,\&\,(x_4 \vee x_5)\,\&\,(\bar{x}_4).$$

Удобно эту КНФ обозначать следующим образом:

---

[7] Допускается случай пустой клаузы – когда N=0. Она тождественно равна нулю.

$$(1 \vee 2)(\overline{2} \vee 3)(\overline{1} \vee \overline{3})(4 \vee 5)(\overline{4}).$$

Выполняющий набор (модель) КНФ K – вектор $\widetilde{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, на котором КНФ обращается в единицу: $K(\widetilde{\alpha}) = 1$.

Частичное приписание – множество $\bigcup_{i \in X} \{(x_i, \sigma_i)\}$ для некоторого подмножества $X \subseteq \{1,2,\ldots,n\}$. Частичное приписание служит для отражения факта, что в процессе работы алгоритма мы полагаем $x_i = \sigma_i$ для некоторых i. В упрощённых обозначениях это множество $\bigcup_{i \in X} \{(i, \sigma_i)\}$.

Приписание называется *полным*, если $X = \{1,2,\ldots,n\}$. При заданном приписании формула может обращаться в единицу (приписание тогда называется *выполняющим*), обращаться в ноль (*невыполняющим*) или быть *неопределённой* (значение формулы существенно зависит от переменных, которые не входят в приписание). Переменные (и соответствующие им литералы), которые не входят в приписание называют *свободными*. Также часто приписанием называют саму пару (x_i, σ_i) или (неформально) принятие решения, что $x_i = \sigma_i$.

Пример

Для КНФ K_1 неполное приписание $\{(x_1,1),(x_2,0),(x_3,0)\}$ оставляет формулу неопределённой, а полное приписание $\{(x_1,1),(x_2,0),(x_3,0),(x_4,0),(x_5,1)\}$ выполняет формулу.

DP-алгоритм

Применение к КНФ правила удаления юнитов, чистых литералов и резолюции.

1. Если в КНФ входит пустая клауза, вернуть НЕВЫПОЛНИМА.
2. Найти переменную, которая входит в клаузу КНФ длины 1 (только она и входит). Удалить все клаузы, которые содержат эту переменную. Удалить все отрицания этой переменной.
3. Найти переменную, которая входит в КНФ либо только с отрицанием, либо без отрицания. Удалить все клаузы, которые содержат эту переменную.
4. Если в КНФ нет клауз, вернуть ВЫПОЛНИМА.
5. Выбрать переменную, которая входит в КНФ с отрицанием и без. Добавить в КНФ клаузы, которые получаются всевозможными резолюциями относительно этой переменной. Удалить все клаузы, в которые входит эта переменная.
6. Повторить.

Пример

$$(x_1 \vee x_2) \,\&\, (\overline{x}_2 \vee x_3) \,\&\, (\overline{x}_1 \vee \overline{x}_3) \,\&\, (x_4 \vee x_5) \,\&\, (\overline{x}_4) \text{ юнит } 4$$

$$(x_1 \vee x_2) \,\&\, (\overline{x}_2 \vee x_3) \,\&\, (\overline{x}_1 \vee \overline{x}_3) \,\&\, (x_5) \text{ чистый литерал } 5$$

$$(x_1 \vee x_2)\,\&\,(\bar{x}_2 \vee x_3)\,\&\,(\bar{x}_1 \vee \bar{x}_3) \text{ резолюция 1}$$

$$(x_2 \vee \bar{x}_3)\,\&\,(\bar{x}_2 \vee x_3)\overline{\overline{\&\,(x_1 \vee x_2)\,\&\,(\bar{x}_1 \vee \bar{x}_3)}}$$

$$(x_2 \vee \bar{x}_3)\,\&\,(\bar{x}_2 \vee x_3) \text{ резолюция 2}$$

$$(x_3 \vee \bar{x}_3)\,\&\,\overline{\overline{(x_2 \vee \bar{x}_3)\,\&\,(\bar{x}_2 \vee x_3)}}$$

$$(x_3 \vee \bar{x}_3)$$

Описание DPLL

1. Применить юнит-резолюцию и удалить все поглощаемые клаузы (ВСР).
2. Если найдена пустая клауза, то вернуть НЕВЫПОЛНИМА.
3. Найти переменную, которая входит в КНФ только с отрицанием или без отрицания.
4. Удалить все клаузы, в которые входит эта переменная (удаление чистых литералов).
5. Если нет клауз, то вернуть ВЫПОЛНИМА.
6. Выбрать переменную x (она входит как с отрицанием, так и без).
7. Рекурсивно вызвать DPLL на текущей КНФ с добавленной клаузой (x).
8. Рекурсивно вызвать DPLL на текущей КНФ с добавленной клаузой $(\bar{x})$.
9. Если один из рекурсивных вызовов вернул ВЫПОЛНИМА, то вернуть ВЫПОЛНИМА.
10. Вернуть НЕВЫПОЛНИМА.

Пространство поиска алгоритма изображают в виде дерева. Каждой вершине дерева соответствует переменная x_i, а исходящим из неё дугам соответствующие приписания: (x_i, σ_i) - если исходит только одна дуга, (x_i, σ_i), $(x_i, \bar{\sigma}_i)$ – если исходят две дуги. Также каждой вершине приписан уровень. Уровень любой вершины равен числу вершин степени выше 1 в цепи, которая начинается в корне и заканчивается в этой вершине. Например, уровень корня равен 1. Уровень соответствует числу принятых решений о значениях переменных в текущем приписании. Остальные значения получились автоматически (см. дальше).

DPLL-поиск осуществляет просмотр дерева с целью обнаружения выполняющего набора. Дерево просматривается до нахождения выполняющего набора (в этом случае просмотр прекращается) или до нахождения конфликта (в этом случае происходит подъём по дереву до предыдущего уровня).

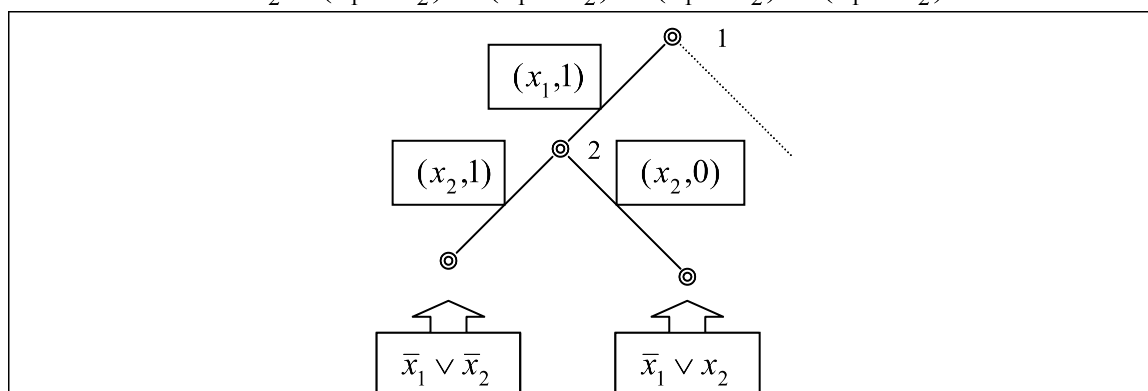
Конфликт – появление невыполняющейся клаузы (т.е. выяснение, что текущее приписание является невыполнимым). При этом найденная невыполнимая клауза называется *конфликтующей.*

Традиционно алгоритм записывают в виде рекурсивной процедуры. Хотя возможны и нерекурсивные реализации [Joao P. Marques-Silva and Karem A. Sakallah. GRASP - a search algorithm for propositional satisfiability. *IEEE Transactions in Computers*, 48(5):506-521,May 1999]. Более того, при практической реализации алгоритма предпочтение отдаётся нерекурсивной форме.

Пример

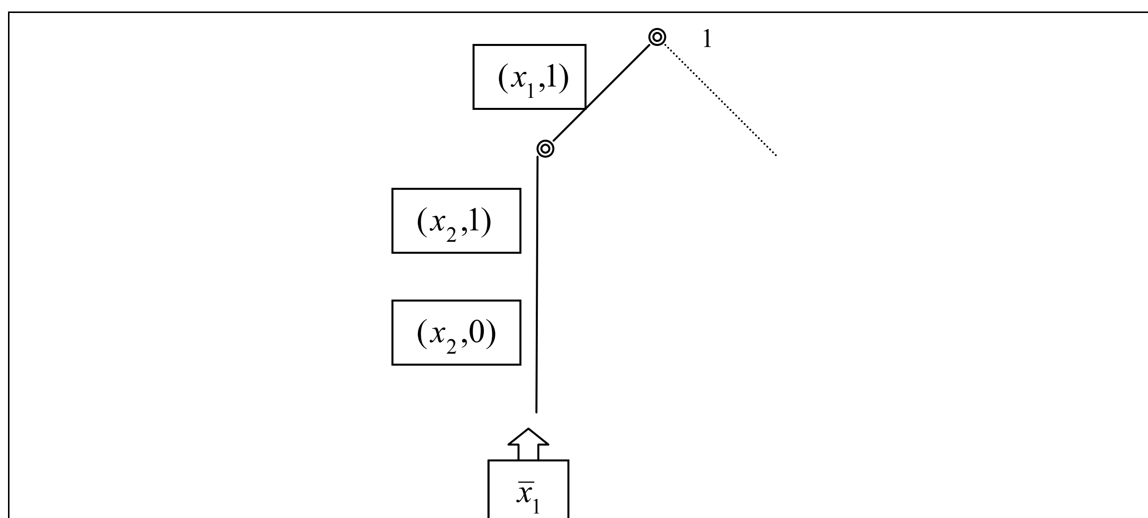
Рассмотрим самый простой пример:

$$K_2 = (x_1 \vee x_2)\,\&\,(x_1 \vee \bar{x}_2)\,\&\,(\bar{x}_1 \vee x_2)\,\&\,(\bar{x}_1 \vee \bar{x}_2)$$



На дереве перебора пространства возможных решений показано появление двух конфликтующих клауз. В реальности дерево перебора значительно отличается от приведённого.

1. Из-за необходимых приписаний нет ветвлений в некоторых вершинах. Число уровней сокращается.
2. Часто никаких конфликтующих клауз в явном виде не возникает. Конфликт (здесь: возможность появления невыполнимой клаузы) отлавливается по необходимым приписаниям (появляются два взаимоисключающих друг друга необходимых приписания).
3. Появляется конфликтная клауза (причина конфликта; см. ниже).



В этом дереве уже один уровень. Это тоже некорректное дерево: два взаимоисключающих приписания в процессе работы алгоритма в явном виде не

возникает[8]. Однако, это дерево отражает реальную ситуацию. Дело в том, что необходимые приписания сразу не осуществляются, а заносятся в стек необходимых приписаний. Появление в таком стеке двух противоречивых приписаний и есть свидетельство конфликта. Почему внизу записана клауза $\bar{x}_1$ (она и называется конфликтной) мы поясним ниже.

Далее приведём различные формализации DPLL-метода.

```
loop
propagate() //propagate unit clauses
 if not conflict then
   if all variables assigned then
     return Satisfiable
   else
     decide() //pick a new variable and assign it
 else
analyze() //analyze con and add a con clause
if top-level con found then
  return Unsatisfiable
else
  backtrack() //undo assignments until con clause is unit
```

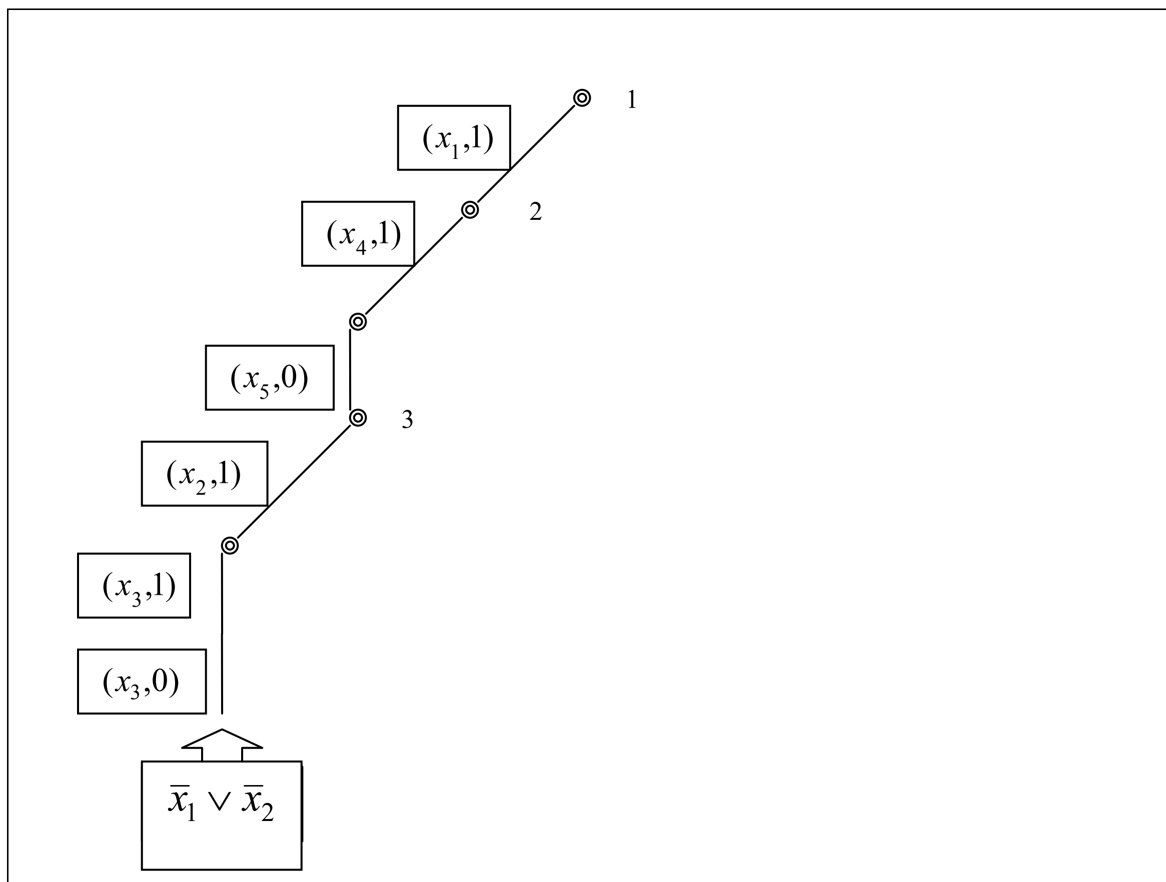
```
while (true)
      {
      if (!decide()) // if no unassigned vars
          return(satisifiable);
      while (!bcp())
            {
            if (!resolveConflict())
               return(not satisfiable);
            }
       }

bool resolveConflict()
     {
     d = most recent decision not 'tried both ways';
     if (d == NULL) // no such d was found
        return false;
     flip the value of d;
     mark d as tried both ways;
     undo any invalidated implications;
     return true;
     }
```

Классический алгоритм.

```
DLL(formula, assignment)
{
necessary = deduction(formula, assignment);
```

[8] Это, впрочем, зависит от реализации алгоритма.

```
new_asgnmnt = union(necessary, assignment);
if (is_satisfied(formula, new_asgnmnt))
   return SATISFIABLE;
else {
   if (is_conflicting(formula, new_asgnmnt))
       return CONFLICT;
}
branching_var = choose_free_variable(formula,
new_asgnmnt);
asgn1 = union(new_asgnmnt, assign(branching_var, 1));
result1 = DLL(formula, asgn1);
if (result1 == SATISFIABLE)
    return SATISFIABLE;
asgn2 = union (new_asgnmnt, assign(branching_var, 0));
return DLL(formula, asgn2);
}
```



На рисунке показана часть дерева разбора для примера $K_1 =$

$$(x_1 \vee x_2 \vee x_3)\,\&\,(x_1 \vee x_2 \vee \bar{x}_3)\,\&\,(x_1 \vee \bar{x}_2 \vee x_3)\,\&\,(x_1 \vee \bar{x}_2 \vee \bar{x}_3)\,\&$$
$$\&\,(\bar{x}_1 \vee x_2 \vee x_3)\,\&\,(\bar{x}_1 \vee x_2 \vee \bar{x}_3)\,\&\,(\bar{x}_1 \vee \bar{x}_2 \vee x_3)\,\&\,(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)\,\&$$
$$\&\,(x_5 \vee x_4)\,\&\,(\bar{x}_5 \vee \bar{x}_4)$$

Сначала делаем предположение, что $x_1 = 1$, затем, что $x_4 = 1$, откуда следует необходимое приписание (см. дальше) $x_5 = 0$ и т.д.

Необходимые приписания (necessary assignments)
Если в процессе поиска решения задачи выполнимости становится известным, что если выполняющий набор существует, то существует выполняющий набор, в котором $x_i = \sigma_i$ при $i \in I$, то приписания (x_i, σ_i), $i \in I$, называются *необходимыми*[9].

Поиск необходимых приписаний играет ключевую роль при реализации SAT-солверов. *Boolean Constraint Propagation (BCP)* – процесс нахождения необходимых приписаний.

1. Юниты (units)
Юнитами называются клаузы длины 1. Если в КНФ входит хотя бы одна такая клауза: (x_i^σ), то в выполняющем наборе $x_i = \sigma$ и соответствующее приписание можно сразу сделать, поскольку оно может вызвать другие необходимые приписания[10].

Например, в КНФ K_1 имеем необходимое приписание $x_4 = 0$, которое вызывает приписание $x_5 = 1$.

В современных алгоритмах обязательно реализуется процедура нахождения юнитов! Это очень важная процедура. Юниты находятся до тех пор, пока возможно. Затем следует расщепление: разветвление в дереве перебора (принудительное назначение какой-то переменной значения).

Разрабатывались аналогичные процедуры, которые «достраивали» выполняющий набор не только по клаузам длины 1, но и по клаузам длины 2 и клаузам специального вида. Однако анализ таких клауз занимает очень много машинного времени.

Процесс BCP, в котором идёт поиск только юнитов, часто называют *unit propagation*.

2. Чистые литералы (pure literals)
Литерал называется *чистым*, если в КНФ не входит его отрицание. В этом случае переменной можно приписать сразу соответствующее значение.

---

[9] Это, естественно, скорее понятие, чем точное определение. Заметим, что каждый выполняющий набор задаёт соответствующие необходимые приписания. Здесь идёт речь о необходимых приписаниях, наличие которых можно «быстро установить» (не решая непосредственно задачу выполнимости).

[10] Это и показывает затруднения при точном определении понятия «необходимое приписание». Одно приписание влечёт за собой другое. Для этой цели, собственно, они и находятся! Происходит существенное упрощение дерева поиска: уменьшения числа уровней, т.е. ветвлений.

Пусть, например, $K_3 = (x_1 \vee x_2)\,\&\,(x_1 \vee x_3)\,\&\,(x_2 \vee \bar{x}_3)\,\&\,(\bar{x}_2 \vee x_3)$, тогда имеем чистый литерал x_1, необходимое приписание $x_1 = 1$, после которого КНФ упрощается: $(x_2 \vee \bar{x}_3)\,\&\,(\bar{x}_2 \vee x_3)$. Заметим, что КНФ K_3 имеет выполняющий набор (011), который не соответствует этому приписанную, однако цель SAT-солвера найти хотя бы один выполняющий набор.

На практике нет процедур поиска чистых литералов, так как

1. Значительное время их поиска не окупается выгодой их применения.
2. Хорошие эвристики расщепления автоматически выбирают такие литералы.
3. Есть также и иные необходимые приписания, которые, однако, трудно идентифицировать. Вообще, если формула имеет единственный выполняющий набор, то любое приписание, которое ему соответствует, является необходимым. Однако, их поиск эквивалентен в этом случае решению проблемы выполнимости.

Пусть в КНФ входят клаузы $(x_1 \vee x_2)$ и $(x_1 \vee \bar{x}_2)$, тогда в выполняющем наборе $x_1 = 1$, так как при $x_1 = 0$ в КНФ входят клаузы x_2 и $\bar{x}_2$ и она тождественно равна нулю.

Если в КНФ входят клаузы $(x_1 \vee \bar{x}_2)$ и $(\bar{x}_1 \vee x_2)$, тогда в выполняющем наборе $x_1 = x_2$.

Эти примеры показывают, что есть очень много методов поиска необходимых приписаний. В зависимости от того, какие приписания мы ищем. Например, вхождение в КНФ подКНФ вида $(x_1 \vee \bar{x}_2)\,\&\,(x_2 \vee \bar{x}_3)\,\&\,(x_3 \vee \bar{x}_1)\,\&\,(x_1 \vee x_2 \vee x_3)$ говорит о необходимости приписаний из $\{(x_1 = 1), (x_2 = 1), (x_3 = 1)\}$. Более того, любая подКНФ «с малым числом единиц» задаёт небольшое множество необоdимых приписаний. Однако необходимость их поиска весьма сомнительна.

Пример. В КНФ $(x_1 \vee x_2)\,\&\,(\bar{x}_1 \vee x_3)\,\&\,(\bar{x}_2 \vee x_3)$ если выполняется первая клауза, то, чтобы выполнялись остальные необходимо, чтобы $x_3 = 1$. Заметим, что клауза (x_3) получается с помощью резолюций из клауз КНФ.

Пример. Нетрудно видеть, что для КНФ

$$(x_1 \vee x_2)\,\&\,(\bar{x}_1 \vee x_3)\,\&\,(x_4 \vee \bar{x}_2)\,\&\,(x_4 \vee \bar{x}_3)$$

справедливо

$$(x_1, 0) \Rightarrow (x_2, 1) \Rightarrow (x_4, 1),$$
$$(x_1, 1) \Rightarrow (x_3, 1) \Rightarrow (x_4, 1).$$

Отсюда следует, что приписание $(x_4,1)$ является необходимым. Из этого примера понятна суть метода Сталмарка[11] для поиска необходимых приписаний: получить приписания, которые следуют из приписания $(x_i,0)$, затем получить приписания, которые следуют из приписания $(x_i,1)$ и взять их пересечение. По причине своей трудоёмкости этот метод редко используется в SAT-солверах.

Задание. Получить все необходимые приписания в рассмотренных примерах с помощью метода резолюций.

Пример. Покажем «неформальность» понятия «необходимое приписание». В КНФ $(x_1 \vee \bar{x}_2)\,\&\,(x_2 \vee \bar{x}_3)\,\&\ldots\&\,(x_{n-1} \vee \bar{x}_n)\,\&\,(x_n \vee \bar{x}_1)$ два выполняющих набора: $(0\ldots0)$ и $(1\ldots1)$. Поэтому любое(!) приписание вида (x_i,σ) является необходимым.

3. Специальные приписания (autarkies)
Это приписание, которое выполняет все клаузы, в которые входят переменные из этого приписания, т.е. это расширение понятия «чистый литерал».
Например, в последнем примере есть два таких приписания.

<u>Расширения DPLL</u>

Далее рассмотрим современные нововведения в алгоритм DPLL.

Запись клауз (Clause Recording Schemes)
После каждого конфликта определяется множество литералов, которого достаточно для его получения. На основе этого множества создаётся клауза (*конфликтная* клауза – conflict clause), которая записывается (добавляется к КНФ).

Идея записи клауз появилась в
[R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence,* p. 203-208, 1997],
а применительно к SAT в
[J.P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design,* pages 220-227, November 1996].

Вообще, это очень популярная идея: обучение на ошибках, которая применяется во многих областях прикладной математики. Находится причина неудачи, а затем она «учитывается» (чтобы в дальнейшем не совершать подобные ошибки).

---

[11] Этот метод защищён патентом для программных продуктов.

В классическом алгоритме DLL каждое приписание в узле дерева имеет флаг – исследовалось ли обратное приписание. При конфликте ищется верхний уровень, для которого не просматривалось обратное приписание. При реализации алгоритма с конфликтными клаузами необходимость во флаге отпадает: по дереву поднимаемся пока конфликтная клауза равна нулю, затем выполняем приписание, обращающее её в единицу[12].

Получение по графу конфликтных клауз

$\omega_1 = (\neg x_1 + x_2)$

$\omega_2 = (\neg x_1 + x_3 + x_9)$

$\omega_3 = (\neg x_2 + \neg x_3 + x_4)$

$\omega_4 = (\neg x_4 + x_5 + x_{10})$

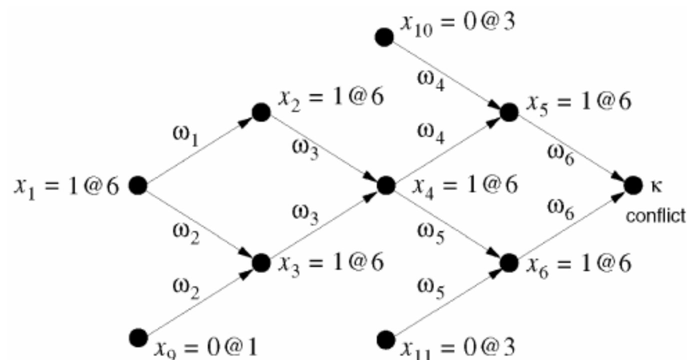
$\omega_5 = (\neg x_4 + x_6 + x_{11})$

$\omega_6 = (\neg x_5 + \neg x_6)$

$\omega_7 = (x_1 + x_7 + \neg x_{12})$

$\omega_8 = (x_1 + x_8)$

$\omega_9 = (\neg x_7 + \neg x_8 + \neg x_{13})$



Рассмотрим ориентированный ациклический граф, на котором показаны приписания[13]. Выражение $x_i = \sigma @ k$ обозначает, что приписание $x_i = \sigma$ было сделано на уровне k. Дуга, ведущая в вершину, помечается клаузой, которая рассматривалась при соответствующем приписании.

Нетрудно видеть, что к конфликту привела ***часть текущего приписания*** $\{(x_1 = 1), (x_9 = 0), (x_{10} = 0), (x_{11} = 0)\}$. КНФ может быть пополнена конфликтной клаузой $\bar{x}_1 \vee x_9 \vee x_{10} \vee x_{11}$ (чтобы избежать подобных конфликтов в будущем). При возврате на 6й уровень эта клауза является юнитом и получается автоматическое приписание $x_1 = 0$.

Заметим, что при обнаружении конфликта и конфликтного приписания $\{(x_{i_1} = \sigma_1 @ k_1), \ldots, (x_{i_m} = \sigma_m @ k_m)\}$ возвращаться надо на уровень $\max[\{k_1, \ldots, k_m\} \setminus \max[k_1, \ldots, k_m]]$ (см. дальше).

Вообще, конфликт – это появление в графе двух взаимоисключающих друг друга приписаний. Переменную в этом приписании будем называть *конфликтной.* Для анализа конфликта достаточно рассматривать компоненту связности, содержащую эти приписания.

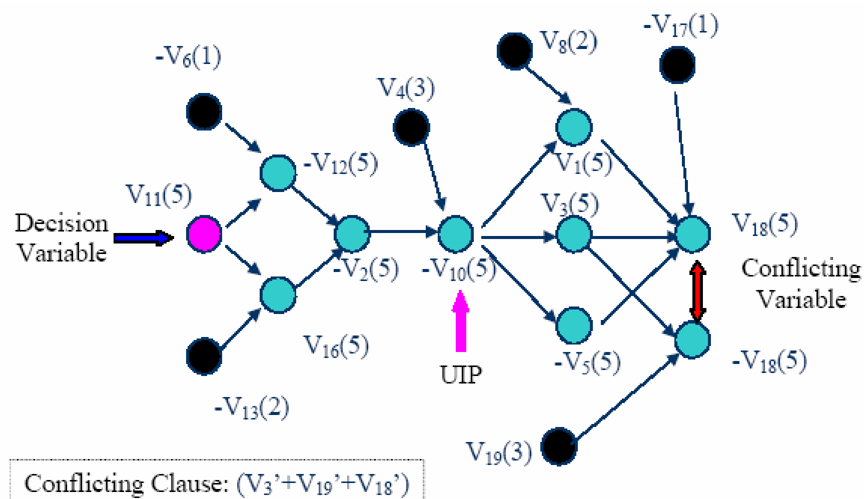
---

[12] Пример будет приведён в разделе «Нехронологический перебор».

[13] Можно сказать, что показана «история приписаний».

Анализ конфликта – процедура, которая находит причину конфликта (часть текущего приписания, которая ответственна за конфликт) и «указывает новое направление для поиска».

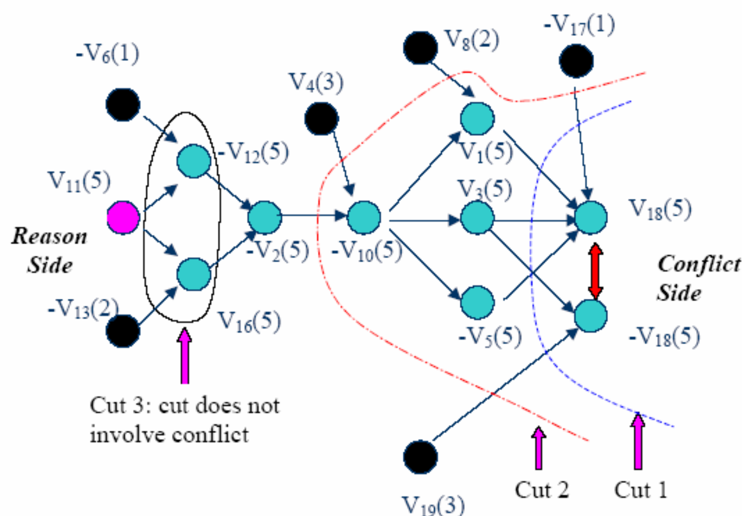
В стандартном DPLL методе при возникновении конфликта происходит возврат по дереву до вершины ветвления, которая соответствует переменной, которую мы рассмативали «в одной фазе».



Современный анализ конфликта включает в себя
1) возврат на более высокие уровни (нехронологический),
2) пополнение списка клауз (обучение).

Клаузы, которыми пополняют список, называют learned clauses или conflict clauses (конфликтная клауза), а клаузы, которые привели к конфликту - conflicting clauses (конфликтующие).

Конфликтную клаузу можно породить с помощью бипарации графа, т.е. разделения графа на две доли: в первой должны быть все переменные ветвления, во второй – вершины с конфликтной переменной. Все вершины первой доли должны иметь путь в вершины конфликтной переменной.



Различные схемы обучения получают разные бипарации.

Один из первых SAT-солверов, в которых было реализовано обучение и нехронологический возврат, Rel_sat строит по графу клаузу, «развёртывая»[14] клаузу $x_t \vee \bar{x}_t$, где x_t – конфликтная переменная, до тех пор, пока она не будет содержать только одну переменную текущего уровня.

Есть схемы обучения, при которых строится кратчайшая конфликтная клауза (или близкая к ней), клауза, содержащая только одну переменную каждого уровня приписания и т.д. Хороший подробный обзор этих схем содержится в [Zhang, L., Madigan, C., Moskewicz, M., And Malik, S. Efficient conflict driven learning in a boolean satisfiability solver. Proceedings of ICCAD 2001 (2001)].

Важное замечание. Конфликтная клауза, которую мы заносим в список, меняет пространство поиска. Надо заносить такую клаузу, которая даёт сочетание переменных «потенциально встречаемое в будущем».

Relevance-Based Learning (так называется простое и эффективное улучшение механизма сохранения клауз)
Не обязательно хранить все конфликтные клаузы в течение всего времени работы алгоритма. Более того, их число растёт экспоненциально, поэтому такое хранение практически неосуществимо. Поэтому их удаляют.
Некоторые критерии удаления:

1. Число неприписаных литералов больше некоторого порога.
2. Клаузы с длиной больше, чем k оставляют, а остальные удаляют, если есть хотя бы два неприписанных литерала (*k-bounded learning)* [GRASP].
3. При решении об удалении учитывается число конфликтов, в которых участвовали литералы клаузы, и возраст клаузы[15].

---

[14] т.е. осуществляется спуск по графу «против стрелок».
[15] Существуют также обобщения перечисленных методов и их комбинации. Например, хранятся короткие клаузы и длинные, в которых число неприписанных литералов меньше некоторого порога.

См. [R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence,* p. 203-208, 1997.] и [J.P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design,* pages 220-227, November 1996.]

Нехронологический перебор (Non-Chronological Backtracking[16])

[R.M. Stallman and G.J. Sussman, “Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis” , *Artificial Intelligence,* vol. 9, pp. 135-196, October 1977.]
[P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268{299, 1993.]

В современных SAT-солверах после конфликта управление передаётся сразу на несколько уровней выше по дереву разбора. Кстати, после такой передачи, алгоритм может принимать решение о приписании переменных не совпадающие с теми, которыми он принимал раньше, т.е. фактически перестраивается целое поддерево перебора!

Пример

Пусть рассматриваемая КНФ

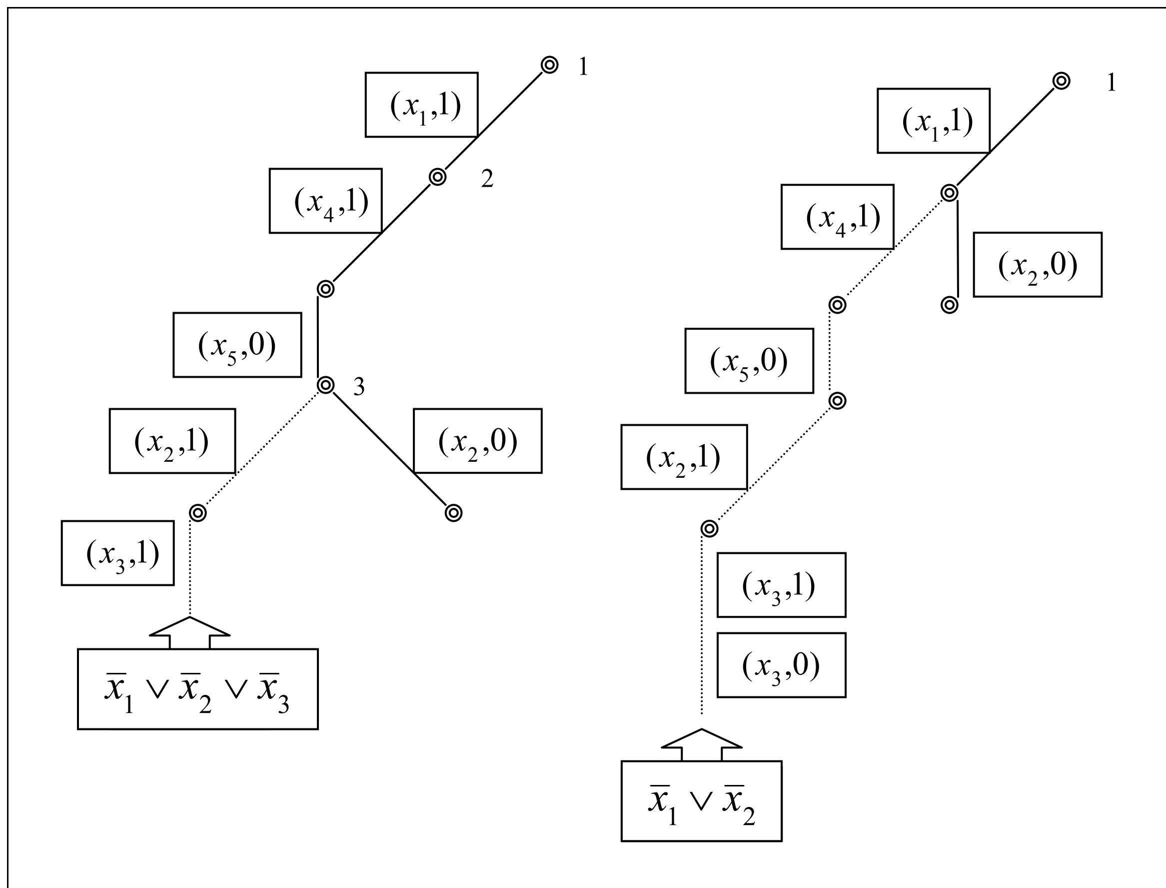
$$(\overline{1} \vee 2 \vee 3)(\overline{1} \vee \overline{2} \vee 3)(\overline{1} \vee 2 \vee \overline{3})(\overline{1} \vee \overline{2} \vee \overline{3})(4 \vee 5)(\overline{4} \vee \overline{5}).$$

Сделаны приписания из $\{(x_1 = 1@1), (x_4 = 1@2), (x_5 = 0@2)\}$ (заметим, что на втором уровне было необходимое приписание). На третьем уровне при приписании $(x_2 = 1@3)$ получаем конфликт. При стандартном переборе дерева (хронологическом) мы возвращаемся на третий уровень и применяем приписание $(x_2 = 0@3)$. При нехронологическом переборе дерева мы возвращаемся на первый уровень и применяем необходимое приписание $(x_2 = 0@1)$, которое следует из конфликтной клаузы $\bar{x}_1 \vee \bar{x}_2$ и приписания первого уровня $(x_1 = 1@1)$. Заметим, что «в создании» конфликтной клаузы не участвовали переменные x_4, x_5. Поэтому при возврате по дереву перебора мы поднимаемся так высоко – их значения не важны (пока)! Гораздо важнее выполнить конфликтную клаузу[17].

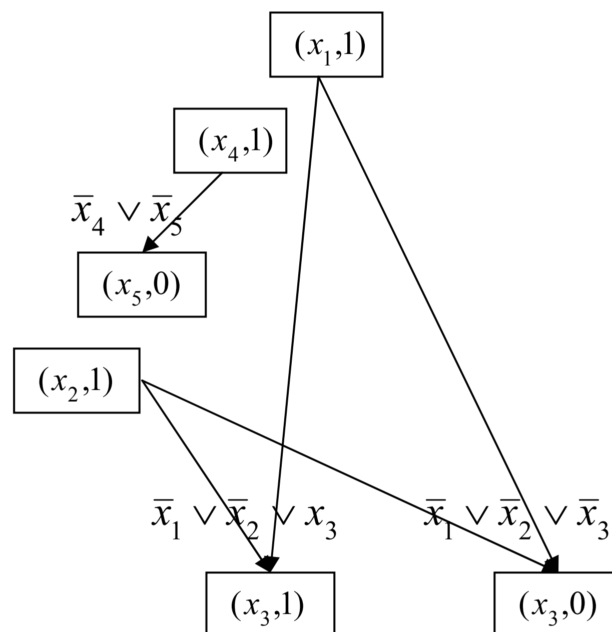
---

[16] Буквально: нехронологический возврат, т.е. возврат по дереву перебора не в хронологическом порядке.

[17] Кстати, выполнение конфликтной клаузы может привести к новым необходимым приписаниям.



Следующая иллюстрация поясняет происхождение конфликтной клаузы.



Отметим, что в солвере satz реализован хронологический перебор. Связано это с наличием look-ahead процедуры для выбора приписаний. Реализация алгоритма становится «неудобной» для получения конфликтных клауз, зато выбирается «более хорошая» переменная для ветвления.

Перезапуски (Search Restarts)

Перезапуском (рестартом) называется останов перебора, обнуление приписаний и запуск перебора заново с новым порядком приписаний.

[C. P. Gomes, B. Selman and H. Kautz. Boosting Combinatorial Search Through Randomization, in. *Proceedings of the National Conference on Artificial Intelligence,*July 1998.]

При перезапуске сохраняется список конфликтных клауз (быть может, частично). Часто перезапуски делают после фиксированного числа шагов (это число постоянно увеличивают). Цель перезапуска – избежать зависания в локальном минимуме[18]. Кроме того, часто необходимо несколько запусков алгоритма для накопления статистики.

Эвристики расщепления (Branching Heuristics)

Эвристики для выбора переменной нужны для приписания в том случае, когда нет необходимых приписаний[19]. Это одна из наиболее важных частей алгоритма: именно от неё зависит время счёта. Переменные, в идеале, надо выбрать именно так, чтобы дерево разбора было минимальным (по числу вершин). Сначала мы рассмотрим только «статические эвристики»: которые не используют накопленный в процессе перебора опыт.

Как правило, выбираются переменные, которые часто входят в короткие клаузы. Это делается для получения юнитов на следующих шагах. Вообще, есть два подхода к выбору переменной: выбирать, чтобы максимально упростить КНФ; выбирать, чтобы получить КНФ «похожую» на выполнимую. Первый считается эффективнее.

Статические эвристики

MOM – Maximum Occurrence on Clauses of Minimal Size

– это общий принцип статических эвристик. Предпочтение отдаётся литералам, которые вместе со своими отрицаниями входят в короткие клаузы. Это можно формализовать максимизацией функционала

$$J(x) = \left[\sum_{j:x\in D_j} 1 + \sum_{j:\bar{x}\in D_j} 1 \right] 2^k + \sum_{j:x\in D_j} 1 \cdot \sum_{j:\bar{x}\in D_j} 1$$

при достаточно большом k.

Literal Count Heuristics

Эвристики, которые используют только информацию о числе переменных в невыполнимых клаузах. Например, максимизируется функционал

[18] Для этого они часто применяются в комбинаторной оптимизации.

[19] Как правило, просто когда нет юнитов.

$$J(x)=\left[\sum_{j:x\in D_j}1+\sum_{j:\bar{x}\in D_j}1\right] \text{ или } J(x)=\sum_{j:x\in D_j}1$$

и выбирается приписание

$$\begin{cases}(x,1), & \sum_{j:x\in D_j}1\geq\sum_{j:\bar{x}\in D_j}1,\\ (x,0), & \sum_{j:x\in D_j}1<\sum_{j:\bar{x}\in D_j}1.\end{cases}$$

Jeroslaw-Wang Heuristics

Делается попытка оценить «рейтинг» каждого литерала (с точки зрения выполнимости всех клауз). Это можно формализовать максимизацией функционала

$$J(y)=\sum_{j:y\in D_j}2^{-|D_j|}.$$

Такую формализацию называют *односторонней эвристикой*. При максимизации функционала $J(y)+J(\bar{y})$ эвристику называют *двусторонней.* При этом делают приписание

$$\left[\begin{array}{ll}(x,\sigma), & J(y)\geq J(\bar{y}),\\ (x,\bar{\sigma}), & J(y)<J(\bar{y}).\end{array}\right.$$

Считается, что двусторонняя эвристика лучше.

В процессе счёта при вычислении $J(y)$ сумма вычисляется по всем свободным клаузам, здесь[20] $|D_j|$ – число свободных литералов в клаузе D_j. Аналогично следует понимать все формулы в этом разделе.

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

Bohm

Выбирается переменная с максимальным вектором $(H_1(x),\ldots,H_n(x))$ в смысле лексикографического порядка,

$$H_i(x)=\alpha\max[h_i(x),h_i(\bar{x})]+\beta\min[h_i(x),h_i(\bar{x})],$$

где $h_i(x)$ - число невыполнимых клауз с i литералами, содержащих переменную x. Обычно полагают $\alpha=1$, $\beta=2$.

Во многих современных SAT-солверах, например в MiniSate, периодически (или случайно[21]) для приписания выбирается случайная переменная.

[20] Обозначение становится корректным, если считается, что КНФ изменяется в процессе перебора.
[21] На каждом шаге это делается с определённой вероятностью.

look-ahead-эвристики

UP-heuristics

Также есть эвристики, которые подсчитывают сколько (необходимых) приписаний вызовет искомое приписание. Вообще подобные эвристики (*BCP-based probing или UP-heuristics или look-ahead*) очень трудоёмки, хотя позволяют выбирать действительно «нужную переменную» (реализованы в rel_sat). Заметим, что в эвристиках, подобных MOM, переменная выбирается, чтобы попытаться (!) максимально эксплуатировать BCP() для последующих выборов переменных. Здесь же эвристика реально оценивает (!) сколько именно необходимых приписаний вызовет данное, т.е. идея MOM реализуется буквально. Естественно, плата за это – время работы процедуры Assign(). Такие эвристики хорошо зарекомендовали себя на случайных 3-КНФ (реализованы в POSIT , Tableau, satz).

Один из возможных подходов следующий.

Перебираем все свободные литералы.
Для текущего литерала x делаем соответствующее приписание.
Для этого приписания проводим BCP().
Если есть конфликт обрабатываем его стандартным методом (*возврат*).
Оцениваем «простоту» множества свободных клауз $C(x)$.
Выбираем литерал, максимизирующий ф-л

$$J(x)=C(x)+C(\bar{x})+C(x)\cdot C(\bar{x})2^k.$$

Очень часто подобными эвристиками анализируют не все переменные, а только часть: те переменные, которые «подозрительно хороши» для расщепления.

Динамические эвристики

Следующие эвристики учитывают историю конфликтов. Основное их преимущество – «мгновенная» оценка рейтинга переменных. Дело в том, что изменение рейтинга происходит только при конфликте. Процедура выбора переменной ветвления просто выбирает литерал с максимальным рейтингом, не проводя оценки как таковой (реализованы в Chaff).

VSIDS (Variable State Independent Decaying Sum Decision) Heuristics

Каждому литералу приписан счётчик (начальное значение 0 или число вхождений литерала в КНФ). При нахождении конфликтной клаузы счётчики всех её литералов увеличиваются на 1. Периодически значения всех счётчиков делятся на константу. Выбирается литерал, счётчик которого имеет максимальное значение.

Впервые реализована в Chaff. Подобные эвристики, которые используют историю поиска, называют эвристиками второго порядка.

В одной из версий солвера Minisat выбрана такая стратегия: после каждого конфликта активность переменных (счётчики) уменьшается на 5%. Переменные хранятся в упорядоченном по активности виде.

BerkMin Heuristics
Эвристика следует идеи VSIDS, но при конфликте увеличиваются счётчики всех(!) литералов, ответственных за конфликт, т.е. которые в графе импликаций имеют путь к конфликтным литералам. А самое главное: клаузы хранятся упорядоченно. При выборе переменной находим первую невыполнимую клаузу – из её литералов и выбираем кандидата.

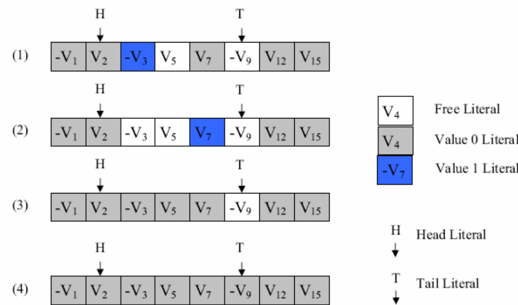
Эффективная реализация некоторых эвристик требует специальных типов данных! Поэтому совместное использование нескольких эвристик (на одних уровнях дерева перебора - одних, на других - других) бывает затруднительным.

Структура данных
сильно влияет на эффективность солвера.

SATO
Это первая «ленивая» структура данных, т.е. организация данных, при которой лишний раз стараются не заносить изменения в клаузы. Первоначально была использована в солвере SATO.

Литералы клаузы записаны в массиве. На два элемента массива указывают специальные ссылки: голова и хвост. Изначально, голова указывает на первый элемент, хвост – на последний. Во время работы алгоритма, все литералы слева от головы и справа от хвоста равны нулю (это постоянно поддерживается). Пусть, например, голова указывает на первый литерал x_i^σ, тогда при приписании $x_i = \overline{\sigma}$ клауза просматривается слева направо от головы. Если клауза выполнима, то она не изменяется и объявляется выполнимой, если находим неприписанный литерал, отличный от хвоста, то он становится головой. При возвращении по дереву голова и хвост возвращаются на свои позиции.

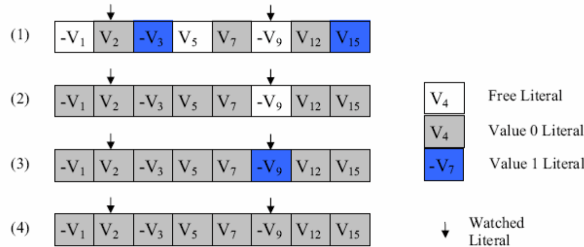
Для каждой переменной X есть четыре списка: клаузы с головой/хвостом равной X/неX.



Chaff's Watched Literals(WL)

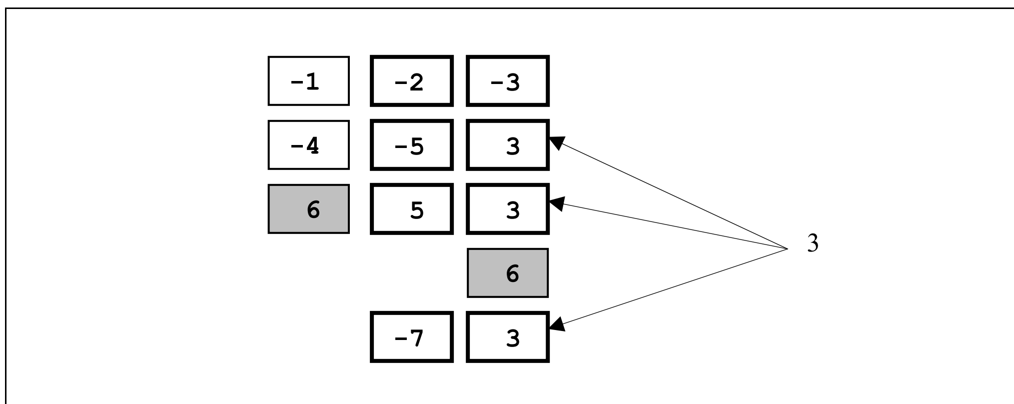
Пока клауза содержит 2 литерала, которым не приписаны значения, она не может быть нулевой или юнитом.

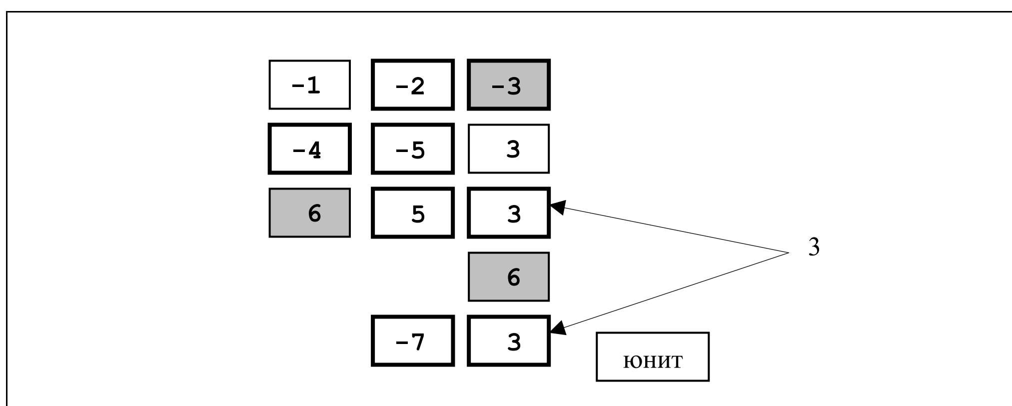
На каждую клаузу ровно 2 ссылки (они не упорядочены). Каждая переменная X имеет два списка: клаузы с отмеченной X/неX.



Преимущество: при возврате по дереву не надо менять ссылки. Недостаток: определение юнит/не юнит происходит при просматривании всех литералов клаузы.

На рисунке показаны изменения отмеченных литералов (ссылок) при приписании $x_3 = 0$ (считается, что до этого выполнено приписание $x_6 = 1$).





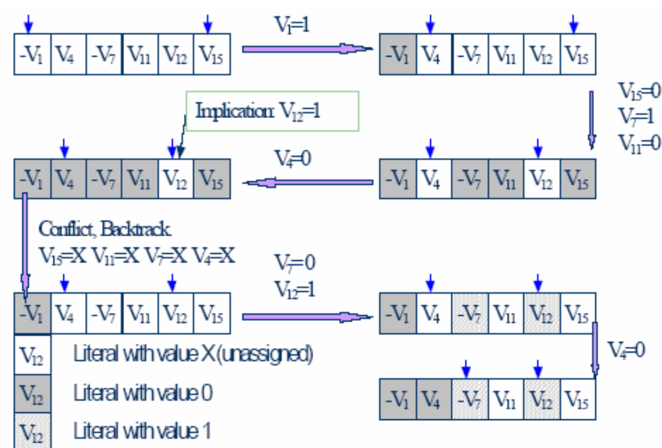


Figure 1: BCP using two watched literals

Эффективность солвера.

На что же следует обращать внимание при создании солвера в первую очередь? Что непосредственно влияет на его эффективность (в смысле среднего времени решения прикладных задач выполнимости)?

1. Эффективная реализация процедуры BCP().

Основную часть времени «забирает» именно эта процедура. КНФ, которые приходится обрабатывать, хорошие в смысле структуры: даже случайные приписания вызывают много необходимых. Следовательно, надо быстро их обнаруживать и применять. Кроме того, под эффективной реализацией BCP() понимается и эффективный возврат по дереву перебора. Хотя это и осуществляют разные процедуры, но они сильно взаимосвязаны: от того как написана процедура BCP() зависит написание BACKTRACK()[22].

2. Выбор приписания (переменной для ветвления).

Здесь главное «угадать» с переменной. Не так важна скорость работы процедуры. Важнее результат. Огромное количество статей посвящено просто процедуре выбора очередного приписания. Собственно, в современных SAT-солверах больше улучшать почти нечего.

3. Сокращение пространства поиска.

[22] Часто её называют ERASE().

Частично это покрывается пунктом 2. Но только частично. По сокращением понимается и предобработка КНФ и её упрощение в процессе счёта. Часто сюда включают обучение алгоритма: запись клауз и их эффективное использование. На самом деле, это понятие включает и в себя частично пункт 1! Точнее, организацию использования необходимых приписаний (какие необходимые приписания отлавливаются).

Предобработка КНФ
Предобработка применяется до запуска основного алгоритма решения задачи выполнимости и служит для упрощения КНФ, её «сжатия» или «подгона под алгоритм» (преобразования КНФ в специальную форму, на которой алгоритм наиболее эффективен).

Предобработка может значительно уменьшить время решения задачи ([37]).
Может включать: сокращения литералов и клауз, добавление новых клауз.
1. Поглощение клауз (clause subsumption)
Клауза D_j поглощает клаузу D_i, если
$D_j = 1 \Rightarrow D_i = 1$. При этом клауза D_i может быть удалена из КНФ.
Пример. В КНФ
$(x_1 \vee \bar{x}_2)\,\&\,(x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4)\,\&\ldots$
вторая клауза может быть удалена.

Поглощение клауз можно применять также при записи клауз для упрощения КНФ. Кроме того, возможны применения к текущей свободной КНФ (учитывая только свободные литералы в конъюнкциях). Однако подобное применение поглощений очень трудоёмко.

2. Разделение формулы (formula partitioning)
КНФ
$(\bar{x}_1 \vee x_2)\,\&\,(\bar{x}_1 \vee x_3)\,\&\,(x_1 \vee \bar{x}_2 \vee \bar{x}_3)\,\&\,(x_4 \vee x_5 \vee \bar{x}_6)\,\&\,(\bar{x}_4 \vee x_6)\,\&\,(\bar{x}_5 \vee x_6)$
распадается на две подКНФ. Первая зависит только от переменных x_1, x_2, x_3, а вторая – только от x_4, x_5, x_6. КНФ выполнима тогда и только тогда, когда выполнимы эти две подКНФ.

Часто разделение формулы удаётся после предварительной обработки. Кроме того, это разделение может быть «завуалировано»:

$$(\bar{x}_1 \vee x_2)\,\&\,(\bar{x}_1 \vee x_3)\,\&\,(x_1 \vee \bar{x}_2 \vee \bar{x}_3)\,\&$$
$$\&\,(x_4 \vee x_5 \vee \bar{x}_6)\,\&\,(\bar{x}_4 \vee x_6)\,\&\,(\bar{x}_5 \vee x_6)\,\&$$
$$\&\,(x_1 \vee \bar{x}_5 \vee x_6)\,\&\,(\bar{x}_1 \vee \bar{x}_5 \vee x_6)$$

3. Резолюция

Пусть дана КНФ $(x_i \vee D_1)\,\&\,(\bar{x}_i \vee D_2)\,\&\ldots$

Очевидно, что она выполнима тогда и только тогда, когда выполнима КНФ $(D_1 \vee D_2)\,\&\,(x_i \vee D_1)\,\&\,(\bar{x}_i \vee D_2)\,\&\ldots$

Клауза $(D_1 \vee D_2)$ называется резольвентой, а операция её добавления в КНФ резолюцией.

Пример. В солвере Satz реализована следующая предобработка. Выполняется перебор по всем парам клауз длины не больше трёх. Для каждой пары получаем резольвенту. Если длина резольвенты не больше трёх, то заносим её в КНФ. Эта нехитрая предобработка хорошо зарекомендовала себя на случайных 3-КНФ, а также индустриальных КНФ с «хорошей структурой».

Например, в КНФ

$$(x_1 \vee x_2 \vee x_3 \vee x_4)\,\&\,(\bar{x}_4 \vee \bar{x}_5)\,\&\,(\bar{x}_1 \vee x_4)\,\&\,(x_1 \vee \bar{x}_2 \vee \bar{x}_3)\,\&\,(x_3 \vee x_4 \vee x_5)$$

осуществляются следующие резолюции

$$(\bar{x}_4 \vee \bar{x}_5)\,\&\,(\bar{x}_1 \vee x_4) \to (\bar{x}_5 \vee \bar{x}_1),$$

$$(\bar{x}_1 \vee x_4)\,\&\,(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \to (x_4 \vee \bar{x}_2 \vee \bar{x}_3),$$

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3)\,\&\,(x_3 \vee x_4 \vee x_5) \to (x_1 \vee \bar{x}_2 \vee x_4 \vee x_5) \text{ (большая)}.$$

Таким образом, ДНФ пополняется двумя клаузами.

4. Трансляция в 3-КНФ

Перевод КНФ в КНФ, состоящую из клауз длины 3. Получается введением фиктивных переменных и операцией

$$D_1 \vee D_2 \to (D_1 \vee t)\,\&\,(D_2 \vee \bar{t}).$$

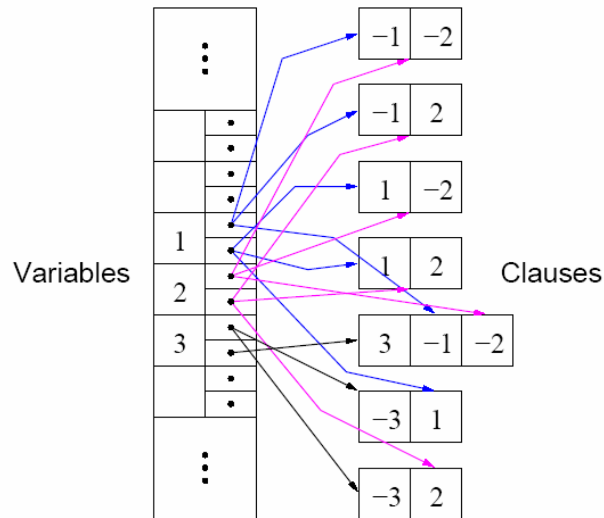
Обзоры алгоритмов предобработки: [Joao P. Marques-Silva. Algebraic simplification techniques for propositional satisfiability. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'2000)*, September 2000., Daniel Le Berre. Exploiting the real power of unit propagation lookahead. In *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*, June 2001]. Однако общий вывод при использовании «хитрых» алгоритмов предобработки такой. Даже если есть видимое значительное упрощение формулы, это не означает, что время работы основного алгоритма уменьшится. Как правило, алгоритм предобработки не реализуют эффективно (не подгоняют под него структуры данных и т.д.), так как он вызывается лишь один раз перед запуском основного алгоритма.

Хранение данных

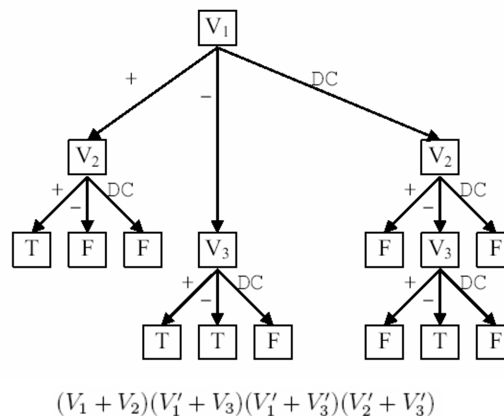
В современных прикладных задачах необходимо обрабатывать КНФ очень больших размеров. При размещении их в памяти необходимо учитывать специфику кэширования: соседние клаузы КНФ должны размещаться «рядом». Данные должны храниться «компактно»!

Способы хранения

1. Разреженная матрица. Хранение в «естественном виде»: sparse matrix.
2. С помощью указателей (GRASP, relsat). Удобно манипулировать с данными, но не эффективно используется кэширование.



3. В отдельном массиве (chaff). Не так удобно работать как со списками, проблемы с удалением клауз[23], но эффективное использование памяти.
4. Дерево (SATO).



$$(V_1 + V_2)(V_1' + V_3)(V_1' + V_3')(V_2' + V_3')$$

[23] В процессе удаления образуется «мусор»: участки памяти, которые не используются и располагаются между «полезными данными».

Формат DIMACS

Формат для хранения КНФ в текстовом файле. В первой строке файла указывается число переменных КНФ (натуральное число и переход строки), во второй – число клауз, в последующих – сами клаузы. В каждой строке записывается ровно одна клауза, т.е. число строк в файле превышает число клауз на 2. Каждая строка, описывающая клаузу, содержит индексы переменных, которые в неё входят. Для переменных с отрицанием перед индексом ставится знак «минус». Заканчивается такая строка нулём (цифра ноль и перевод строки). Обратите внимание, что переменные нумеруются с единицы, а в каждой строке число целых чисел превышает число литералов в клаузе на единицу.

3 4 1 -2 0 2 -3 0 3 -1 0 1 2 3 0	$(x_1 \vee \bar{x}_2)\,\&\,(x_2 \vee \bar{x}_3)\,\&\,(x_3 \vee \bar{x}_1)\,\&\,(x_1 \vee x_2 \vee x_3)$

Пишем SAT-солвер…

Написание SAT-солвера, т.е. программы, которая на вход получает КНФ (лучше всего в формате DIMACS), а на выход выдаёт «SAT», «UNSAT», «UNKNOWN», в зависимости от результата своей работы, это лишь написание «движка». Следующий шаг – настройка параметров алгоритма, навешивание различных эвристических процедур. Дело в том, что практически невозможно написать универсальный солвер, т.е. программу, которая одинаково хорошо будет работать на всех видах КНФ. Поэтому основная часть программистского времени уходит на настойку программы на те данные, на которых её планируется в дальнейшем использовать. Здесь решающую роль играет выбор тестовых КНФ.

Дальше разберём код типичного SAT-солвера.

```
#define   SAT 1
#define UNSAT 2
#define CONFLICT 3

unsigned int N; // число переменных
unsigned int K; // число клауз

vector< vector<unsigned int> >C; // клаузы...
                                 // [переменная][отрицание]
                                 // -----x Посл.бит x=1 => неX
                                 //        Посл.бит x=0 =>   X

vector< vector<unsigned int> >W; // списки watched-переменных
                           // обращение по [переменная][отрицание]
```

Пользуемся библиотекой стандартных шаблонов. Правда при этом сильно уменьшается скорость работы солвера. Клаузы хранятся в векторе векторов. Каждый элемент этой структуры данных – беззнаковое целое, первый бит которого – «знак переменной» (сама переменная или её отрицание).

Аналогично хранятся данные в списке watched-переменных. Только здесь хранятся номера клауз, в которых содержатся соответствующие переменные, т.е. если `W[3][5]==4`, то переменная $\bar{x}_1$ (`3=1*2+1`) является нулевой watched-переменной во 2й клаузе (`4=2*2+0`). Все ссылки на клаузы, содержащие $\bar{x}_1$-watched-переменную, находятся в `W[3][.]`.

Важно! В каждой клаузе первые две позиции (нулевую и первую) занимают watched-литералы. На какую именно позицию ссылается элемент списка watched, определяется правым битом (остатком от деления на 2). Предполагаем, что каждая клауза содержит более одного литерала. Исходные юниты сразу заносятся в стек юнитов, watched-ссылок на них нет.

```
vector<unsigned int> U;  // стек юнитов
                         // [переменная][отрицание]


vector<unsigned int> Ux; // это их вектор ----xy
                         // --x- - знак во втором бите! X - неX
                         // ---y - =1 если это ПРОЧЕРК
```

Стек юнитов нужен для обнаружения конфликтов. Сюда заносятся необходимые приписания перед их непосредственным приписанием. Основное их применение показано в функции

```
//inline
int PushUnit(unsigned int c)
{
          if ((Ux[c>>1])&1)
             {
             Ux[c>>1] = ((c&1)<<1);
             U.push_back(c);
             }
          else
             if (Ux[c>>1] != ((c&1)<<1))
                {
                U.push_back(c); // потом для эффективного удаления
истории болезни!
                return UNSAT; // КОНФЛИКТ!
                }
return SAT; // просто успешный возврат
}
```

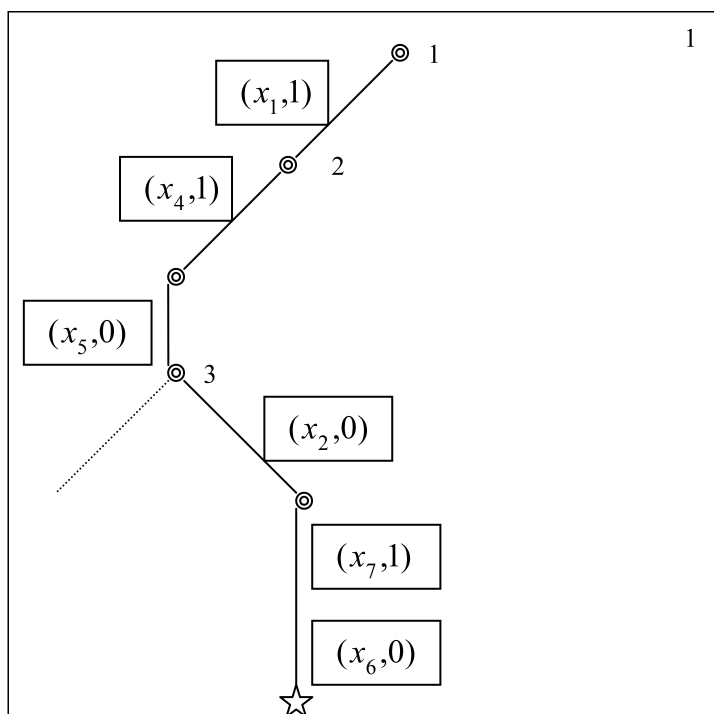
Если раньше не было занесения в стек, то заносим очередной юнит. Если было (это контролирует вектор `Ux`), то смотрим наличие конфликта (одновременное занесение переменной и её отрицания).

```
vector< int> Uc;                                 // для анализа конфликтов!
                                                  // ссылки на юниты
                                                  // адресация от 2 до 2N+1
```

Хранятся номера клауз, которые являются текущими юнитами. Например, если `U[6]==5`, то приписание $x_3 = 0$ (`6 = 3*2+0`) вызвала пятая клауза (она содержит переменную $\bar{x}_3$).

```
vector<unsigned int> Levels;    // для каждой переменной - уровень
                              // пусть пока нумерация от 2 до 2N+1

unsigned int Level;                         // текущий уровень
unsigned int Level2;                             // куда возвращаться
после обработки конфликта

vector<int> X;                                           // набор текущих
значений! SIGNED!!!

vector< vector<unsigned int> >Xh;
 // история присвоений!
 // [переменная][отрицание]
 //
 //  [баз.пер.(0,0) ]  [баз.пер.(1,0) ]  [баз.пер.(2,0) ]
 //  [необх.пр.(0,1)]  [необх.пр.(1,1)]
 //                    [необх.пр.(1,2)]
```

Эти структуры данных тесно взаимосвязаны в процессе работы солвера. Пусть текущее состояние дерева разбора следующее (в задаче восемь переменных)



тогда `X==[-1, 0, 1, -1, 0, 1, 1, 0, -1]` (нулевая позиция ничему не соответствует, 0 соответствует переменной, 1 – её отрицанию), `Levels==[0, 1, 3, 0, 2, 2, 3, 3, 0]`, `Xh==[[1*2], [4*2, 5*2+1], [2*2+1, 7*2, 6*2+1]]`. Обратите внимание, что переменные нумеруются от единицы (это сделано для удобства, хотя можно нумеровать и с нуля).

Например, в следующем фрагменте кода представлена функция, которая проверяет, выполняется ли на данном наборе КНФ (если выполняется – возвращает 1).

```
int CheckSolution()
{
for (int i=K; --i>=0; ) // по всем клаузам
    {
    int j=C[i].size();
    for (; --j>=0;)     // по литералам в клаузе
        {
        if (X[((C[i][j])>>1)]==(((C[i][j])&1))) // если = 1
           break;
        }
    if (j<0) return 0; // если клауза = 0 или - (завершили проход)
    }
return 1;
}
```

Обратите внимание, на то, что выполнение условия `X[((C[i][j])>>1)]==(((C[i][j])&1))` означает, что `j`-й литерал `i`-й клаузы обращается на наборе `X` в 1.

```
vector <unsigned int> ClauseZ; // клауза, которой обучились!
vector <int> XClauseZ;
// это для её заполнения... "обращение до N+1"
```

Это типы для хранения конфликтной клаузы. Используется и для её создания.

```
vector<double> Activity;        // активность клауз в конфликтах
                                // [переменная][отрицание]

vector<unsigned int> Lengths;   // статистика длин клауз

unsigned long NMconflict;
unsigned long NMassign;
unsigned long NMpropagate;
unsigned long NMunit;
```

Различные переменные для сбора статистики: вектор активности переменных (участие в конфликтах), сколько клауз фиксированной длины входит, число конфликтов, приписаний, вызовов функции `propagate`, число юнитов. Естественно, это факультативная часть в коде солвера.

Разберём основную функцию Solve…

```
int Solve()
{

Xh.resize(1); // подготовить историю

for (;;)
    {

    if (Propagate()==CONFLICT)
       {
       // Обработка конфликта

       // СТАТИСТИКА
       Lengths[ClauseZ.size()]++;

       for (int i=U.size(); --i>=0;) Uc[(U[i])] = -1;
       // удаление графа "истории болезни"

       U.resize(0);
       Ux.resize(0);
       Ux.resize(N,1); // везде прочерки

       int basevar = Xh[Level-1][0];  // = ClauseZ[0];

       if (Level==1) return UNSAT;

```

```
        // понижение уровня...
        // удаление истории конфликтов
        while (Xh.size()>Level2)
              {
              if (Level==1) return UNSAT;
              Level--;
              for (int i=Xh[Level].size(); --i>=0;)
                  {
                  int c = (Xh[Level][i]);
                  Uc[c] = -1;
                  // обнулить историю юнитов!
                  X[c>>1] = -1;
                  Levels[c] = 0;
                  }
              Xh.pop_back();
              }


        basevar = basevar ^ 1;


        // заносим в список конфликтную клаузу
        if (ClauseZ.size()==1) Uc[basevar] = -1;
        else
           {
           if (ClauseZ.size()<=3) // если короткая
              {
              // ДОБАВЛЕНИЕ В ОБЩИЙ СПИСОК...
              W[ClauseZ[0]].push_back((C.size()<<1));
              W[ClauseZ[1]].push_back((C.size()<<1)|1);
              }

           Uc[basevar] = C.size();
           C.push_back(ClauseZ);
           }

        // Адаптировать активность...
        for (int i=ClauseZ.size(); --i>=0;)
             Activity[ClauseZ[i]]+=1;
        if (NMconflict%N==0) for (int i=(N<<1); --i>=0;)
                                   Activity[i]/=2;

        PushUnit(basevar);
        }
     else
        {
        if (Assign()==SAT) return SAT;
        }

     } // for(;;)
}
```

В бесконечном цикле мы запускаем сначала функцию Propagate(), а затем Assign(). В функции Propagate() обрабатываются необходимые приписания (если они есть). В функции Assign() происходит очередное приписание (ветвление дерева), если приписание нельзя осуществить (все переменные имеют значения), то рассматриваемая КНФ выполнима. Между прочим, приписание, которое порождает Assign() заносится в стек юнитов! Вообще, функция делает следующее:

1. Увеличивает текущий уровень (`Level++`).
2. Расширяет историю (`Xh.resize(Xh.size()+1)`).
3. Заносит приписание в стек юнитов (`PushUnit(i)`).
4. Ведёт статистику (`NMassign++`).

Напомним, что `PushUnit(9)` означает занесение приписания $\bar{x}_4$.

В функции Propagate() может случиться конфликт (два взаимоисключающих друг друга приписания занесены в стек юнитов), тогда функция возвращает UNSAT. В теле функции Solve происходит возврат на нужный уровень (Level2) по дереву (для наглядности возврат стоило бы оформить в виде отдельной функции). Если возвращаться «уже некуда», то наша КНФ невыполнима. При возврате:

1. Обнуляется информация о графе импликаций (`Uc[c] = -1`).
2. Текущий набор заполняется прочерками (`X[c>>1] = -1`).
3. Обнуляются уровни у приписаний (`Levels[c] = 0`).
4. Удаляются слои из истории приписаний: т.е. все приписания этого уровня (`Xh.pop_back()`).

После всех этих действий заносим конфликтную клаузу в общий список. Если её длина небольшая (`ClauseZ.size()<=3`), то оформляем на неё watched-ссылки (см. также функцию Propagate). Отметим, что в представленной программе совсем нет механизма удаления! Хотя, конечно, его несложно написать. Самый простой вариант (если пишем солвер без обучения), не оформлять watched-ссылки, а при возврате по дереву добавить пункт

5. Удалить конфликтные клаузы-юниты, порождавшие соответствующие необходимые приписания.

Разберём последовательно функцию Propagate…

```
int Propagate()
{
NMpropagate++;  //счётчик

while (!U.empty()) // по списку юнитов
      {

      NMunit++;   //счётчик
```

```
      // ИЗ СТЕКА ЮНИТОВ ДОСТАТЬ ОЧЕРЕДНОЙ ЛИТЕРАЛ
      int x = U[U.size()-1];
      U.pop_back();                  // x <= юнит
      X[x>>1] = (x&1);               // занести в вектор X
      Xh[Xh.size()-1].push_back(x); // занести в историю Xh
      Levels[x] = Level;             // X приписать уровень
```

Из списка юнитов извлекаются элементы (пока список не станет пустым). После извлечения литерала надо «зарегистрировать» соответствующее необходимое приписание, т.е.

1. Внести изменения в текущий набор `X`.
2. Внести изменения в историю `Xh`.
3. Приписать литералу текущий уровень (разбора дерева).

Ясно, что в принципе солвер может быть упрощён. Например, информация об уровне может быть получена из истории приписаний `Xh`.

```
int Wsize = W[x^1].size(); // где есть не_x

      // ОБРАБОТКА СПИСКА WATCHED
      for (int i=0; i<Wsize; ++i)    // по списку Watched
          {

          int w = W[x^1][i];
          int w1 =   w>>1;           // в какой клаузе w-литерал
          int w2 =  (w&1);           // какой по счёту...
          int j=C[w1].size();

          for (; --j>=2;)               // по очередной конъюнкции
              {                         // кроме 2х крайних эл-ов

              if (X[C[w1][j]>>1]<0)     // нашли др. watched
                  {
                  int xch = C[w1][j];   // обмен
                  C[w1][j] = C[w1][w2];
                  C[w1][w2] = xch;

                  W[xch].push_back(w);    // в новый список watched
                // удалить из старого Watched (само удаление потом)
                  Wsize--;
                  W[x^1][i] = W[x^1][Wsize];
                  i--;                   // ещё раз пройтись...

                  break;
                  }

              if (X[C[w1][j]>>1]==((C[w1][j])&1)) {break;} // =1

              } // for j
```

Следующий шаг – обработка списка watched для нового приписания. Нужно сменить watched-литералы в клаузах на неприписанные. При этом, если какой-то литерал равен 1, то ничего не меняем, а если находим неприписанный литерал, то делаем его watched-литералом. Чтобы сделать литерал watched-литералом надо

1. Перенести его на одну из первых двух позиций в клаузе.
2. Внести изменения в списки watched (удалить элемент из старого списка, корректно внести информацию в новый).

```
          if  (j<2) // если (!) всё заполнено...
              {
              int c  = C[w1][w2^1];

              if (X[c>>1]<0)         // да! это - юнит.
                 {
```

Если мы не нашли неприписаннный литерал, то наша клауза порождает необходимое приписание, которое мы заносим в стек юнитов. Тонкий момент, связанный с рассматриваемой реализацией: проверяем второй watched-литерал (он точно не обращается в ноль, но он может обращаться в единицу, и клауза выполнима).

```
if (PushUnit(c)==UNSAT) // тут можно и с нулевым флагом занести
                    {
                    // произошёл конфликт!

                    NMconflict++; // счётчик

                    Level2 = 1; // нижняя оценка уровня

                    // подготовить конфликтную клаузу
                    ClauseZ.resize(0);          // удалить!
                    XClauseZ.resize(0);
                    XClauseZ.resize(N,0);

                //факультативно для дальнейшего внесения в историю
                    ClauseZ.push_back((Xh[Xh.size()-1][0])^1);
                    // точно есть тут
                    // Хотя есть тонкость, когда вверху одни юниты

                    XClauseZ[(Xh[Xh.size()-1][0]>>1)]++;
                    // больше и не заносить (1)
    ClauseZ.insert(ClauseZ.end(),C[Uc[c]].begin(),C[Uc[c]].end());
ClauseZ.insert(ClauseZ.end(),C[Uc[c^1]].begin(),C[Uc[c^1]].end());
                    XClauseZ[(c>>1)]++;
                    // тоже не заносить! (2)

                    unsigned int iw2 = 0;
                   // где вторая watched
                   // чтобы потом поставить на позицию 1
```

```
                    // на нуле уже стоит...

                   // счётчик с 1 из-за (1)
                   // вот (2)... точно не учитывается?
                for (unsigned int i1 = 1; i1<ClauseZ.size(); i1++)
                        {
                        int x = ClauseZ[i1];

                        if (XClauseZ[x>>1]!=0)
                           {
                           // уже добавляли!
                          // просто пропустить
                          ClauseZ[i1] = ClauseZ[ClauseZ.size()-1];
                          i1--;
                          ClauseZ.pop_back();
                           }
                        else
                           {
                           XClauseZ[x>>1]++; // = 1

                           if (Uc[x ^ 1]>=0)
                                       // здесь везде(!) x (+) 1
                              {
                              XClauseZ[x>>1] = -1;
                  // она не входит в клаузу! но её смотрели...
ClauseZ.insert(ClauseZ.end(),C[Uc[x    ^    1]].begin(),C[Uc[x    ^
1]].end());
                              //прикрепить историю
                          ClauseZ[i1] = ClauseZ[ClauseZ.size()-1];
                              // удалить эту (есть история!)
                              i1--;
                              ClauseZ.pop_back();
                              }

                     // смотрим уровень возврата для этой клаузы...
               if ((Levels[x ^ 1]>Level2)&&(Levels[x ^ 1]!=Level))
                              {
                              Level2 = Levels[x ^ 1];
                              }

        if ((Level2 == Levels[x ^ 1]) && (Xh[Level2-1][0]==(x^1)))
                               iw2 = i1;// потом в позицию 1
                           }
                        }

                    if (iw2>1)
                           // поместить вторую watched в позицию 1
                       {
                       unsigned int tmp = ClauseZ[iw2];
                       ClauseZ[iw2] = ClauseZ[1];
                       ClauseZ[1]   = tmp;
                       }
```

```
                    W[x^1].resize(Wsize); // изм. размер!

                    return CONFLICT;
                    } // конфликт
```

Если при внесении необходимого приписания в список юнитов произошёл конфликт (т.е. внесены два взаимоисключающих друг друга приписания), то его надо обработать. Вообще, обработка конфликта это отдельная процедура, кроме того, часть обработки проходит в основной функции Solve (вызывается из неё). Здесь представлен код для анализа графа импликаций (для наглядности это надо оформить в виде отдельной функции). Основная задача – «оформить конфликтную клаузу». Это клауза представлена двумя векторами: `ClauseZ` и `XClauseZ`. Во втором помечается, «какие литералы уже занесены» (для отсеивания повторных занесений). В первом в итоге будет конфликтная клауза. Изначально заносим все литералы, ответственные за конфликт («последний слой» в графе импликаций), на первую позицию можно сразу поставить первый литерал текущего уровня (его назначила функция Assign[24]). Причём этот литерал ставим на нулевую позицию (он соответствует юниту, который порождается конфликтной клаузой). На первую позицию ставится литерал, который при возврате по дереву перебора будет вторым watched-литералом, т.е. литерал второго по величине уровня в конфликтной клаузе.

```
                         }
                    }
               }
      // ! НЕЛЬЗЯ СТИРАТЬ ВЕСЬ СПИСОК... ТАМ ЕСТЬ ЮНИТЫ!
      W[x^1].resize(Wsize);
      }

return 0;
}
```

Последний фрагмент кода функции Propagate можно оставить без комментариев… Отметим лишь, что перед выходом надо закончить удаление лишнего в старом watched-списке.

Мы разобрали основные функции очень простого и даже примитивного SAT-солвера. Отметим, что «самая первая» функция – функция загрузки данных. Она корректно оформляет списки клауз, watched-литералов и т.д. Кроме того, перед её запуском надо обнулить все счётчики и подготовить структуры данных для записи информации. Например, возможен подобный фрагмент кода

```
NMconflict  = 0;
NMassign    = 0;
NMpropagate = 0;
```

[24] Внимательный анализ кода показывает, что это не всегда так. Но при исключениях это не сказывается на функциональности.

```
NMunit      = 0;
NMDunit     = 0;
N = N+1; // нумерация с единицы!
Activity.resize(N<<1,0);
W.resize(N<<1);
Uc.resize(N<<1,-1);
X.resize(N,-1); //!
Ux.resize(N,-1);
C.resize(K);
Levels.resize(N<<1,0);
Lengths.resize(N,0);
```

Что надо для написания более мощного солвера?

1. Более тщательно обдумать структуры данных. Сделать их эффективными (учесть локализацию). Наверное, отказаться от библиотеки стандартных шаблонов или «подстроить её» под задачу.
2. Написать более эффективную обработку графа импликаций.
3. Продумать механизм обучения солвера. Сделать эксперименты.

Пример работы DPLL-алгоритма

Разберём log-файлы нашего солвера, в которых отражены приписания и конфликты, возникающие в процессе DPLL-перебора.

<table>
<tr><td>
<pre>
p cnf 7 10
 1 -2   0
    2 -3 0
-1   3 0
 1 2 3 0
-1 -2 -3 0
          5 -6 0
          6 -7 0
          7 -5 0
         5 6 7 0
      -5 -6 -7 0
</pre>
</td><td>

Пример 1.

Невыполнимая КНФ с 7 переменными и 10 клаузами.
</td></tr>
<tr><td>
<pre>
Xh = [] [7]
X = (------1)

Xh = [] [7 6]
X = (-----11)
ConfClause = [-7]
U = [-5 +5]

Xh = [-7]
X = (------0)

Xh = [-7 -5]
X = (----0-0)
ConfClause = [+7]
U = [-6 +6]
</pre>
</td><td>

Первое приписание $x_7 = 1$. В первой строке показано содержимое истории приписаний `Xh`. Первый уровень пустой (здесь должны быть необходимые приписания). Он был бы заполнен, если в исходной КНФ были бы юниты. Во второй строке – текущий вектор `X`.

При конфликте показана конфликтная клауза `ConfClause` и текущее содержимое стека юнитов (в нём обязательно есть два взаимоисключающих друг друга приписания).
</td></tr>
</table>

UNSAT 2 конфликта	

В следующем примере показаны содержимое двух log-файлов для одной невыполнимой КНФ при разном порядке приписаний. Это иллюстрация того, насколько важен «правильный» порядок приписаний.

Пример 2.

p cnf 8 10 1 -2 5 0 1 -2 -5 0 2 -3 6 0 2 -3 -6 0 3 -4 7 0 3 -4 -7 0 -1 4 8 0 -1 4 -8 0 1 2 3 4 0 -1 -2 -3 -4 0	
Xh = [] [8] X = (-------1) Xh = [] [8] [7] X = (------11) Xh = [] [8] [7] [6] X = (-----111) Xh = [] [8] [7] [6] [5] X = (----1111) Xh = [] [8] [7] [6] [5] [4] X = (---11111) Xh = [] [8] [7] [6] [5] [4 3] X = (--111111) Xh = [] [8] [7] [6] [5] [4 3 2] X = (-1111111) ConfClause = [-4 -5 -6 -7] U = [+1 -1] Xh = [] [8] [7] [6] [5 -4] X = (---01111) Xh = [] [8] [7] [6] [5 -4 -1] X = (0--01111) Xh = [] [8] [7] [6] [5 -4 -1 -2]	Xh = [] [1] X = (1-------) Xh = [] [1] [-4] X = (1--0----) ConfClause = [+4 -1] U = [-8 +8] Xh = [] [1 4] X = (1--1----) Xh = [] [1 4] [7] X = (1--1--1-) Xh = [] [1 4] [7 3] X = (1-11--1-) Xh = [] [1 4] [7 3 -2] X = (1011--1-) ConfClause = [-7 -1] U = [-6 +6] Xh = [] [1 4 -7] X = (1--1--0-) Xh = [] [1 4 -7 3] X = (1-11--0-) Xh = [] [1 4 -7 3 -2] X = (1011--0-)

```
X = (00-01111)
ConfClause = [-5 -6 -7 -8 ]
U = [-3 +3 ]

Xh = [] [8 ] [7 ] [6 -5 ]
X = (----0111)

Xh = [] [8 ] [7 ] [6 -5 ] [4 ]
X = (---10111)

Xh = [] [8 ] [7 ] [6 -5 ] [4 3 ]
X = (--110111)

Xh = [] [8 ] [7 ] [6 -5 ] [4 3 2 ]
X = (-1110111)
ConfClause = [-4 -6 -7 -8 ]
U = [-1 +1 ]

Xh = [] [8 ] [7 ] [6 -5 -4 ]
X = (---00111)

Xh = [] [8 ] [7 ] [6 -5 -4 -1 ]
X = (0--00111)

Xh = [] [8 ] [7 ] [6 -5 -4 -1 -2 ]
X = (00-00111)
ConfClause = [-6 -7 -8 ]
U = [-3 +3 ]

Xh = [] [8 ] [7 -6 ]
X = (-----011)

Xh = [] [8 ] [7 -6 ] [5 ]
X = (----1011)

Xh = [] [8 ] [7 -6 ] [5 ] [4 ]
X = (---11011)

Xh = [] [8 ] [7 -6 ] [5 ] [4 3 ]
X = (--111011)

Xh = [] [8 ] [7 -6 ] [5 ] [4 3 2 ]
X = (-1111011)
ConfClause = [-4 -5 -8 -7 ]
U = [+1 -1 ]

Xh = [] [8 ] [7 -6 ] [5 -4 ]
X = (---01011)

Xh = [] [8 ] [7 -6 ] [5 -4 -1 ]
X = (0--01011)

Xh = [] [8 ] [7 -6 ] [5 -4 -1 -2 ]
X = (00-01011)
ConfClause = [-5 -7 -8 ]
U = [+3 -3 ]

Xh = [] [8 ] [7 -6 -5 ]
X = (----0011)

Xh = [] [8 ] [7 -6 -5 ] [4 ]
X = (---10011)

Xh = [] [8 ] [7 -6 -5 ] [4 3 ]
X = (--110011)

Xh = [] [8 ] [7 -6 -5 ] [4 3 2 ]
X = (-1110011)
ConfClause = [-4 -7 -8 ]
U = [-1 +1 ]

Xh = [] [8 ] [7 -6 -5 -4 ]
X = (---00011)

Xh = [] [8 ] [7 -6 -5 -4 -1 ]
X = (0--00011)

Xh = [] [8 ] [7 -6 -5 -4 -1 -2 ]
X = (00-00011)
ConfClause = [-1 ]
U = [-6 +6 ]

Xh = [-1 ]
X = (0-------)

Xh = [-1 ] [2 ]
X = (01------)
ConfClause = [-2 +1 ]
U = [-5 +5 ]

Xh = [-1 -2 ]
X = (00------)

Xh = [-1 -2 ] [6 ]
X = (00---1--)

Xh = [-1 -2 ] [6 -3 ]
X = (000--1--)

Xh = [-1 -2 ] [6 -3 4 ]
X = (0001-1--)
ConfClause = [-6 +1 ]
U = [+7 -7 ]

Xh = [-1 -2 -6 ]
X = (00---0--)

Xh = [-1 -2 -6 -3 ]
X = (000--0--)

Xh = [-1 -2 -6 -3 4 ]
X = (0001-0--)
ConfClause = [+1 ]
U = [+7 -7 ]

UNSAT
6 конфликтов
```

```
ConfClause = [-7 -8 ]
U = [+3 -3 ]

Xh = [] [8 -7 ]
X = (------01)

Xh = [] [8 -7 ] [6 ]
X = (-----101)

Xh = [] [8 -7 ] [6 ] [5 ]
X = (----1101)

Xh = [] [8 -7 ] [6 ] [5 ] [4 ]
X = (---11101)

Xh = [] [8 -7 ] [6 ] [5 ] [4 3 ]
X = (--111101)

Xh = [] [8 -7 ] [6 ] [5 ] [4 3 2 ]
X = (-1111101)
ConfClause = [-4 -5 -6 -8 ]
U = [+1 -1 ]

Xh = [] [8 -7 ] [6 ] [5 -4 ]
X = (---01101)

Xh = [] [8 -7 ] [6 ] [5 -4 -1 ]
X = (0--01101)

Xh = [] [8 -7 ] [6 ] [5 -4 -1 -2 ]
X = (00-01101)
ConfClause = [-5 -6 -8 ]
U = [-3 +3 ]

Xh = [] [8 -7 ] [6 -5 ]
X = (----0101)

Xh = [] [8 -7 ] [6 -5 ] [4 ]
X = (---10101)

Xh = [] [8 -7 ] [6 -5 ] [4 3 ]
X = (--110101)

Xh = [] [8 -7 ] [6 -5 ] [4 3 2 ]
X = (-1110101)
ConfClause = [-4 -6 -8 ]
U = [-1 +1 ]

Xh = [] [8 -7 ] [6 -5 -4 ]
X = (---00101)

Xh = [] [8 -7 ] [6 -5 -4 -1 ]
X = (0--00101)

Xh = [] [8 -7 ] [6 -5 -4 -1 -2 ]
X = (00-00101)
ConfClause = [-6 -8 ]
U = [-3 +3 ]

Xh = [] [8 -7 -6 ]
X = (-----001)

Xh = [] [8 -7 -6 ] [5 ]
X = (----1001)

Xh = [] [8 -7 -6 ] [5 ] [4 ]
X = (---11001)

Xh = [] [8 -7 -6 ] [5 ] [4 3 ]
X = (--111001)

Xh = [] [8 -7 -6 ] [5 ] [4 3 2 ]
X = (-1111001)
ConfClause = [-4 -5 -8 ]
U = [+1 -1 ]

Xh = [] [8 -7 -6 ] [5 -4 ]
X = (---01001)
```

```
Xh = [] [8 -7 -6 ] [5 -4 -1 ]
X = (0--01001)

Xh = [] [8 -7 -6 ] [5 -4 -1 -2 ]
X = (00-01001)
ConfClause = [-5 -8 ]
U = [+3 -3 ]

Xh = [] [8 -7 -6 -5 ]
X = (----0001)

Xh = [] [8 -7 -6 -5 ] [4 ]
X = (---10001)

Xh = [] [8 -7 -6 -5 ] [4 3 ]
X = (--110001)

Xh = [] [8 -7 -6 -5 ] [4 3 2 ]
X = (-1110001)
ConfClause = [-4 -8 ]
U = [-1 +1 ]

Xh = [] [8 -7 -6 -5 -4 ]
X = (---00001)

Xh = [] [8 -7 -6 -5 -4 -1 ]
X = (0--00001)

Xh = [] [8 -7 -6 -5 -4 -1 -2 ]
X = (00-00001)
ConfClause = Xh = [-8 ]
U = [+3 -3 ]

Xh = [-8 ]
X = (-------0)

Xh = [-8 ] [7 ]
X = (------10)

Xh = [-8 ] [7 ] [6 ]
X = (-----110)

Xh = [-8 ] [7 ] [6 ] [5 ]
X = (----1110)

Xh = [-8 ] [7 ] [6 ] [5 ] [4 ]
X = (---11110)

Xh = [-8 ] [7 ] [6 ] [5 ] [4 3 ]
X = (--111110)

Xh = [-8 ] [7 ] [6 ] [5 ] [4 3 2 ]
X = (-1111110)
ConfClause = [-4 -5 -6 -7 ]
U = [+1 -1 ]

Xh = [-8 ] [7 ] [6 ] [5 -4 ]
X = (---01110)

Xh = [-8 ] [7 ] [6 ] [5 -4 -1 ]
X = (0--01110)

Xh = [-8 ] [7 ] [6 ] [5 -4 -1 -2 ]
X = (00-01110)
ConfClause = [-5 -6 -7 +8 ]
U = [-3 +3 ]

Xh = [-8 ] [7 ] [6 -5 ]
X = (----0110)

Xh = [-8 ] [7 ] [6 -5 ] [4 ]
X = (---10110)

Xh = [-8 ] [7 ] [6 -5 ] [4 3 ]
X = (--110110)

Xh = [-8 ] [7 ] [6 -5 ] [4 3 2 ]
X = (-1110110)
ConfClause = [-4 -6 -7 +8 ]
```

```
U = [-1 +1 ]

Xh = [-8 ] [7 ] [6 -5 -4 ]
X = (---00110)

Xh = [-8 ] [7 ] [6 -5 -4 -1 ]
X = (0--00110)

Xh = [-8 ] [7 ] [6 -5 -4 -1 -2 ]
X = (00-00110)
ConfClause = [-6 -7 +8 ]
U = [-3 +3 ]

Xh = [-8 ] [7 -6 ]
X = (-----010)

Xh = [-8 ] [7 -6 ] [5 ]
X = (----1010)

Xh = [-8 ] [7 -6 ] [5 ] [4 ]
X = (---11010)

Xh = [-8 ] [7 -6 ] [5 ] [4 3 ]
X = (--111010)

Xh = [-8 ] [7 -6 ] [5 ] [4 3 2 ]
X = (-1111010)
ConfClause = [-4 -5 +8 -7 ]
U = [+1 -1 ]

Xh = [-8 ] [7 -6 ] [5 -4 ]
X = (---01010)

Xh = [-8 ] [7 -6 ] [5 -4 -1 ]
X = (0--01010)

Xh = [-8 ] [7 -6 ] [5 -4 -1 -2 ]
X = (00-01010)
ConfClause = [-5 -7 +8 ]
U = [+3 -3 ]

Xh = [-8 ] [7 -6 -5 ]
X = (----0010)

Xh = [-8 ] [7 -6 -5 ] [4 ]
X = (---10010)

Xh = [-8 ] [7 -6 -5 ] [4 3 ]
X = (--110010)

Xh = [-8 ] [7 -6 -5 ] [4 3 2 ]
X = (-1110010)
ConfClause = [-4 -7 +8 ]
U = [-1 +1 ]

Xh = [-8 ] [7 -6 -5 -4 ]
X = (---00010)

Xh = [-8 ] [7 -6 -5 -4 -1 ]
X = (0--00010)

Xh = [-8 ] [7 -6 -5 -4 -1 -2 ]
X = (00-00010)
ConfClause = [-7 +8 ]
U = [+3 -3 ]

[-8 -7 ]
X = (------00)

[-8 -7 ] [6 ]
X = (-----100)

[-8 -7 ] [6 ] [5 ]
X = (----1100)

[-8 -7 ] [6 ] [5 ] [4 ]
X = (---11100)

[-8 -7 ] [6 ] [5 ] [4 3 ]
```

```
X = (--111100)

[-8 -7 ] [6 ] [5 ] [4 3 2 ]
X = (-1111100)
ConfClause = [-4 -5 -6 +8 ]
U = [+1 -1 ]

[-8 -7 ] [6 ] [5 -4 ]
X = (---01100)

[-8 -7 ] [6 ] [5 -4 -1 ]
X = (0--01100)

[-8 -7 ] [6 ] [5 -4 -1 -2 ]
X = (00-01100)
ConfClause = [-5 -6 +8 ]
U = [-3 +3 ]

[-8 -7 ] [6 -5 ]
X = (----0100)

[-8 -7 ] [6 -5 ] [4 ]
X = (---10100)

[-8 -7 ] [6 -5 ] [4 3 ]
X = (--110100)

[-8 -7 ] [6 -5 ] [4 3 2 ]
X = (-1110100)
ConfClause = [-4 -6 +8 ]
U = [-1 +1 ]

[-8 -7 ] [6 -5 -4 ]
X = (---00100)

[-8 -7 ] [6 -5 -4 -1 ]
X = (0--00100)

[-8 -7 ] [6 -5 -4 -1 -2 ]
X = (00-00100)
ConfClause = [-6 +8 ]
U = [-3 +3 ]

[-8 -7 -6 ]
X = (-----000)

[-8 -7 -6 ] [5 ]
X = (----1000)

[-8 -7 -6 ] [5 ] [4 ]
X = (---11000)

[-8 -7 -6 ] [5 ] [4 3 ]
X = (--111000)

[-8 -7 -6 ] [5 ] [4 3 2 ]
X = (-1111000)
ConfClause = [-4 -5 +8 ]
U = [+1 -1 ]

[-8 -7 -6 ] [5 -4 ]
X = (---01000)

[-8 -7 -6 ] [5 -4 -1 ]
X = (0--01000)

[-8 -7 -6 ] [5 -4 -1 -2 ]
X = (00-01000)
ConfClause = [-5 +8 ]
U = [+3 -3 ]

[-8 -7 -6 -5 ]
X = (----0000)

[-8 -7 -6 -5 ] [4 ]
X = (---10000)

[-8 -7 -6 -5 ] [4 3 ]
X = (--110000)
```

[-8 -7 -6 -5] [4 3 2] X = (-1110000) ConfClause = [-4 +8] U = [-1 +1] [-8 -7 -6 -5 -4] X = (---00000) [-8 -7 -6 -5 -4 -1] X = (0--00000) [-8 -7 -6 -5 -4 -1 -2] X = (00-00000) ConfClause = [+8] U = [+3 -3] UNSAT 32 конфликта	

Задание. Построить деревья перебора для этих примеров.

Пример 3.

```
p cnf 12 16
 1 -2        5           9     0
 1 -2       -5                 0
 1 -2                   -9     0
    2 -3       -6          -10 0
    2 -3                    10 0
    2 -3        6              0
       3 -4        7    -9     0
       3 -4       -7           0
       3 -4             9      0
-1        4          -8     10 0
-1        4                -10 0
-1        4           8        0
 1  2  3  4                     11     0
 1  2  3  4                    -11     0
-1 -2 -3 -4                      11 12 0
-1 -2 -3 -4                     -11    0
```

```
Xh = [] [1]
X = (1-----------)

Xh = [] [1] [-4]
X = (1--0--------)

Xh = [] [1] [-4 -10]
X = (1--0-----0--)
ConfClause = [+4 -1]
U = [+8 -8]

Xh = [] [1 4]
X = (1--1--------)

Xh = [] [1 4] [-3]
X = (1-01--------)

Xh = [] [1 4] [-3 9]
X = (1-01----1---)
ConfClause = [+3 -1]
U = [-7 +7]

Xh = [] [1 4 3]
X = (1-11--------)

Xh = [] [1 4 3] [11]
X = (1-11------1-)

Xh = [] [1 4 3] [11 -2]
X = (1011------1-)

Xh = [] [1 4 3] [11 -2 10]
X = (1011-----11-)
ConfClause = [-11 -1]
```

```
U = [+6 -6]

Xh = [] [1 4 3 -11]
X = (1-11------0-)

Xh = [] [1 4 3 -11] [-12]
X = (1-11------00)

Xh = [] [1 4 3 -11] [-12 -2]
X = (1011------00)

Xh = [] [1 4 3 -11] [-12 -2 10]
X = (1011-----100)
ConfClause = [+12 -1]
U = [+6 -6]

Xh = [] [1 4 3 -11 12]
X = (1-11------01)

Xh = [] [1 4 3 -11 12] [-2]
X = (1011------01)

Xh = [] [1 4 3 -11 12] [-2 10]
X = (1011-----101)
ConfClause = [+2 -1]
U = [+6 -6]

Xh = [] [1 4 3 -11 12 2]
X = (1111------01)

Xh = [] [1 4 3 -11 12 2] [5]
X = (11111-----01)

Xh = [] [1 4 3 -11 12 2] [5] [6]
X = (111111----01)

Xh = [] [1 4 3 -11 12 2] [5] [6] [7]
X = (1111111---01)

Xh = [] [1 4 3 -11 12 2] [5] [6] [7] [8]
X = (11111111--01)

Xh = [] [1 4 3 -11 12 2] [5] [6] [7] [8] [9]
X = (111111111-01)

Xh = [] [1 4 3 -11 12 2] [5] [6] [7] [8] [9] [10]
X = (111111111101)

SAT
5 конфликтов
```

Основная литература

Inês Lynce and João P. Marques-Silva An overview of backtrack search satisfiability algorithms Annals of Mathematics and Artificial Intelligence 37: 307–326, 2003.

D. Du, J. Gu, and P,M. Pardalos, editors. *Satisfiability Problem: Theory and Applications,* volume 35. American Mathematical Society, 1997.

M.R. Garry and D.S. Johnson. *Computers and Intractability: A Guide to the theory of NP-Completeness.* W.H. Freeman and Company, San Francisco, 1979.

J. Gu, P.W. Purdom, J. Franco, and B.W. Wah: Algorithms for the Satisfiability(SAT) Problem: A Survey. In *Discrete Mathematics and Theoretical Computer Science: Satisflability (SAT) Problem*, vol. 35, pages 19-152 (1997).

Описания солверов и некоторых особенностей их реализации

R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence,*p. 203-208, 1997.

J.P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design,* pages 220-227, November 1996.

H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deductuion,* p. 272-275, July 1997. 72

M. Moskewicz, C. Madigan, Y.Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference,* 2001.

E. Goldberg and Y. Novikov. BerkMin: a Fast and Robust Sat-Solver. In *Design, Automation, and Test in Europe (DATE ’02)*, March 2002, pp. 142-149.

I. Lynce and J. Marques-Silva. Efficient Data Structures for Fast SAT Solvers. *International Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, May 2002.

J. P. Marques Silva, “An Overview of Backtrack Search Satisfiability Algorithms”, in *Fifth International Symposium on Artificial Intelligence and Mathematics*, January 1998.

Интернет-ресурсы

http://satlive.org/ – сайт, посвящённый проблемы выполнимости (много полезных ссылок на другие ресурсы).

http://www.satcompetition.org/ – сайт чемпионатов по разработке SAT-солверов.

http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/ – вся информация о солвере MiniSat (один из лучших в мире по состоянию на 2006г.)