# SAT Solver



## Was ist SAT?

- Erfüllbarkeit von aussagenlogischen Formeln
  - gegeben in konjunktiver Normalform
  - $f(x_1, x_2, x_3) = (x_1 + x_2) \cdot (\neg x_1 + \neg x_3) \cdot (\neg x_1 + x_3)$
  - Erfüllende Belegung:
    $x_1 = 0,\ x_2 = 1,\ x_3 = 0.$



## Komplexität

- SAT ist **NP-vollständig**
  - Sogar 3SAT ist NP-vollständig
  - Es kann nicht erwartet werden, dass es für alle Probleminstanzen schnell eine Antwort gibt
  - In der Praxis sind die Instanzen oft „einfach“, >100.000 Klauseln sind handhabbar
  - Es gibt Funktionen, deren Erfüllbarkeit nur schwer entscheidbar ist.



## Anwendungen

- **Equivalence Checking**
- **Property Checking**
- Logik-Synthese
- Testvektor-Generierung
- Künstliche Intelligenz
- ... und vieles mehr

## Literatur


- J.P. Marques-Silva, K.A. Sakallah: Grasp – a search algorithm for propositional satisfiability. IEEE Trans. on Computers, 48(5), 506-521, 1999.
- M. Moskewicz, *et al.*: Chaff: Engineering an efficient SAT solver. Proc. of the 39th Design Automation Conf., 2001.
- H. Chen, C. Gomes, B. Selman: Formal models of heavy-tailed behavior in combinational search. Principles and Practice of Constraint Programming, LNCS 2239, pp. 408-421.

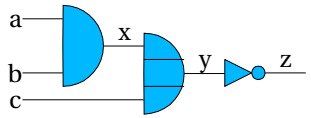



## Schaltkreise als SAT-Formel

- SAT basiert auf konjunktiver Normalform
- Produkt der charakteristischen Funktion der Gatter
  - neue Variablen für alle internen Signale
- Beispiel: AND-Gatter ($x \equiv a \cdot b$):
  $(x \equiv a \cdot b) = (x \rightarrow a) \cdot (x \rightarrow b) \cdot (\neg x \rightarrow (\neg a + \neg b))$
  $= (\neg x + a) \cdot (\neg x + b) \cdot (x + \neg a + \neg b)$
- Beispiel: OR-Gatter ($x \equiv a + b$):
  $(x \equiv a + b) = (x + \neg a) \cdot (x + \neg b) \cdot (\neg x + a + b)$



## Schaltkreise als SAT-Formel (2)




- $(x \equiv a \cdot b) \cdot (y \equiv x + c) \cdot (z \equiv \neg y)$
  - $(x \equiv a \cdot b) = (\neg x + a) \cdot (\neg x + b) \cdot (x + \neg a + \neg b)$
  - $(y \equiv x + c) = (y + \neg x) \cdot (y + \neg c) \cdot (\neg y + x + c)$
  - $(z \equiv \neg y) = (z + y) \cdot (\neg z + \neg y)$
- Größe der SAT-Formel ist linear zur Größe des Schaltkreises



## Satisfiability Checking

- Finde eine Belegung der Variablen, die die Formel erfüllt, oder beweise, dass es keine solche Belegung gibt
  - Erfüllen bedeutet, dass alle Klauseln *gleichzeitig* erfüllt sein müssen!
- Typische Größen (lösbarer) SAT-Instanzen:
  - > 100 Variablen
  - > 10 Millionen Klauseln

## Entwicklung von SAT-Solvern

| Instance | Posit' 94 | Grasp' 96 | Sato' 98 | Chaff' 01 |
|---|---|---|---|---|
| ssa2670-136 | 40,66s | 1,2s | 0,95s | 0,02s |
| bf1355-638 | 1805,21s | 0,11s | 0,04s | 0,01s |
| pret150_25 | >3000s | 0,21s | 0,09s | 0,01s |
| dubois100 | >3000s | 11,85s | 0,08s | 0,01s |
| aim200-2_0-no-1 | >3000s | 0,01s | 0s | 0s |
| 2dlx_..._bug005 | >3000s | >3000s | >3000s | 2,9s |
| c6288 | >3000s | >3000s | >3000s | >3000s |

Quelle: Joao Marques-Silva, Technical University of Lisbon



## Satisfiability Checking (2)

- Durch eine Belegung (eines Teils der) Variablen kann eine Klausel
  - erfüllt sein (= 1)
  - unerfüllt sein (= 0)
  - offen sein („unresolved“)
- Beispiel: $(a+b)(\neg a+c)(\neg a+d)$ $a=1, b=1, c=0$



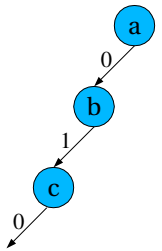

## SAT Solving

- Naives Verfahren: Aufzählen aller Belegungen
- Besseres Verfahren:
  - Aufzählen der Belegungen in Form eines „*Entscheidungsbaumes*“
  - Backtracking, sobald es einen Konflikt gibt
    - Konfliktanalyse
  - (Randomisierte) Neustarts
  - Effiziente Implementierungen



## Entscheidungsbaum

- Drei Teile
  - Entscheidung
    - welche Variable, welcher Wert
  - Deduktion
    - Auflösen der Implikationen
  - Diagnose
    - der Konflikte



## Begriffe

- Literal
  - Eine Variable oder deren Komplement
- Freie Literale
  - Literale, die durch eine Variablenbelegung nicht festgelegt sind
- Unit Clause
  - Klausel, die nur aus einem (freien) Literal besteht
- Null Clause
  - Eine leere Klausel (unerfüllbar!)



## Davis-Putnam – Algorithmus

```
bool Satisfiable(clause list S) {
    // unit propagation
    do {
        for each unit clause L ∈ S do {
            delete every clause from S containing L
            delete ¬L from every clause S containing ¬L
        }
        if S is empty then return true;
        if S contains a null clause then return FALSE;
    } while (there was a change);
    ...
```



## Davis-Putnam – Algorithmus (2)

```
    ...
    // splitting
    choose a literal L occurring in S
    if Satisfiable(S ∪ { L }) then return TRUE;
    if Satisfiable(S ∪ { ¬L }) then return TRUE;
    return FALSE;
}
```



## Beispiel

$k_1 = (\neg x_1 + \neg x_3)$, $k_2 = (x_1 + \neg x_2)$, $k_3 = (x_1 + \neg x_2 + \neg x_3)$

| Action | Decision | Implic. | $k_1$ | $k_2$ | $k_3$ |
|---|---|---|---|---|---|
| Start | Ø | Ø | $k_1$ | $k_2$ | $k_3$ |
| Decision | $x_2$=1 | Ø | $k_1$ | $x_1$ | $x_1+\neg x_3$ |
| Decision | $x_2$=1, $x_3$=1 | Ø | $\neg x_1$ | $x_1$ | $x_1$ |
| Implicat. | $x_2$=1, $x_3$=1 | $x_1$=0 | 1 | C | Conflict |
| Back | $x_2$=1 | Ø | $k_1$ | $x_1$ | $x_1+\neg x_3$ |
| Decision | $x_2$=1, $x_3$=0 | Ø | 1 | $x_1$ | 1 |
| Implicat. | $x_2$=1, $x_3$=0 | $x_1$=1 | 1 | 1 | 1 |

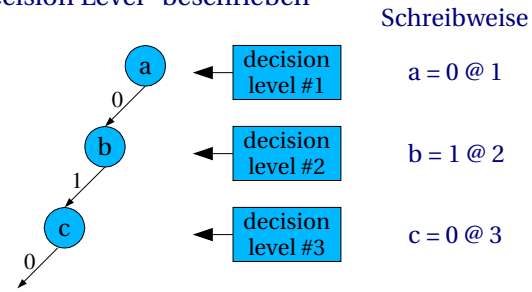
## Neuere Verbesserungen

- Suchstrategien
  - Welche Variable wird ausgewählt
  - Wann / wie soll Backtracking ausgeführt werden?
- Suchtechniken
  - Hinzunahme neuer Klauseln
  - Vereinfachen der Formel
- Bessere Datenstrukturen
  - Schnellere Branch-Strategien
  - Schnellere Implikationen
  - Schnellere Konfliktanalyse



## Decision Level

- Die Reihenfolge der Entscheidungen wird durch das „Decision Level“ beschrieben






## Implikationen

- Betrachte $(a+b)(\neg a+c)(\neg a+d+e)$
  - Wenn $a = 0$ ist, dann muss $b = 1$ sein, d.h. aus $a = 0 @ i$ folgt $b = 1 @ i$ als direkte Implikation
  - Wenn $a = 1 @ i$ und später $d = 0 @ j$, dann impliziert dies $e = 1 @ j$.
- Das Erkennen *aller* Implikationen wird als **Boolean Constraint Propagation** (BCP) bezeichnet.



## Implikationsgraph

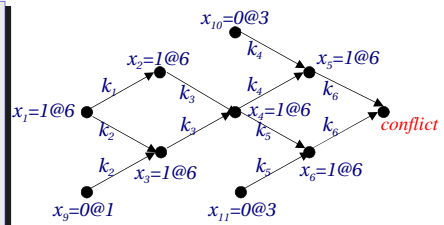
- Die Implikationen einer Entscheidung können als Implikationsgraph gespeichert werden
  - Ein Knoten für jede Variablenzuweisung
  - Kanten sind mit Klauseln markiert und bedeuten „Implikation“
- Beispiel: Klausel $k_i = (a + b)$

$$a = 0 @ j \xrightarrow{k_i} b = 1 @ j$$

# Beispiel

- Zuweisungen: $x_9$=0@1,$x_{10}$=0@3, $x_{11}$=0@3, $x_{12}$=1@2, $x_{13}$=1@2
- Momentan: $x_1$=1@6

$k_1 = (\neg x_1 \vee x_2)$
$k_2 = (\neg x_1 \vee x_3 \vee x_9)$
$k_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$
$k_4 = (\neg x_4 \vee x_5 \vee x_{10})$
$k_5 = (\neg x_4 \vee x_6 \vee x_{11})$
$k_6 = (\neg x_5 \vee x_6)$
$k_7 = (x_1 \vee x_7 \vee \neg x_{12})$
$k_8 = (x_1 \vee x_8)$
$k_9 = (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$






# Konfliktanalyse

- Zuweisungen: $x_9$=0@1,$x_{10}$=0@3, $x_{11}$=0@3, $x_{12}$=1@2, $x_{13}$=1@2
- Vorhin: Zuweisung $x_1$=1@6 führt zu Konflikt ➔ backtrack
- Konflikt hing ab von $x_1$=1, $x_9$=0, $x_{10}$=0, $x_{11}$=0.
- Wenn diese 4 Variablen später wieder diese Belegung haben, muss Implikationsgraph wieder ausgewertet werden
- Es ist möglich, die **Ursache des Konflikts** zu **lernen**, indem die Klausel $(x_1 + \neg x_9 + \neg x_{10} + \neg x_{11})$ zur Klauselmenge hinzugenommen wird
  - Konflikt ist schon durch Auswerten der zusätzlichen Klausel sichtbar



# Konfliktanalyse (2)

- Problem: Hinzunahme neuer Klauseln führt zu einer Vergrößerung der Klauselmenge
- Wie lässt sich ein zu starkes Wachsen verhindern?
  - Es dürfen nur Klauseln, die „wenige“ Literale enthalten, aufgenommen werden
  - Klauseln können später wieder gelöscht werden, wenn sie nichts (mehr) nutzen

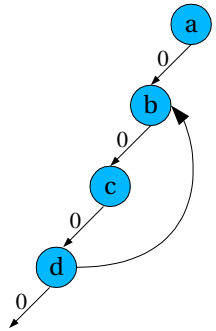


# Non-chronological Backtracking

- Die hinzugefügte Klausel hängt von den Variablen ab, von denen der Implikationsgraph abhängt
- Es ist möglich, dass der Konflikt nicht auf die letzte Variablenzuweisung zurückgeht
  - Beispiel:
    a = 0 @ 1, b = 0 @ 2
    Konflikt wird bei c = 1 @ 5 sichtbar
- Dann kann zu decision level 2 zurückgesprungen werden („non-chronological backtracking“)

## Beispiel

- Angenommen, bei d = 0 @ 4 wird die Klausel (a + b) gelernt
- Dann ist ein Backtracking zu Level 2 möglich






## Beobachtung

- 90% der Rechenzeit wird für Boolean Constraint Propagation (BCP) verbraucht
  - Effiziente Implementierung sehr wichtig
- „Schließen“ ist nur möglich, wenn eine Klausel genau eine freie Variable hat (und alle anderen Literale mit 0 belegt sind)
  - Verfahren 1: Betrachte jede Klausel und zähle die freien Variablen
- Wie kann ich dies bei einer Klausel schnell bestimmen?



## Optimiertes BCP (Chaff)

- Betrachte zwei Literale $l_1$ und $l_2$ einer Klausel
  - Solange beide Literale frei sind, kann noch nichts impliziert werden
  - Wenn eines der Literale $l_b = 0$ ist (etwa $b = 1$), dann
    - gibt es entweder noch ein weiteres freie Literal. Ersetze dann $l_1$ durch dieses Literal
    - oder es gibt kein weiteres freies Literal. Dann ist $l_2$ das einzige freie Literal und es kann muss 1 sein.
  - Invariante: beide Literale einer offenen Klausel sind frei
  - Beim Backtracking muss kein Update erfolgen



## Beispiel

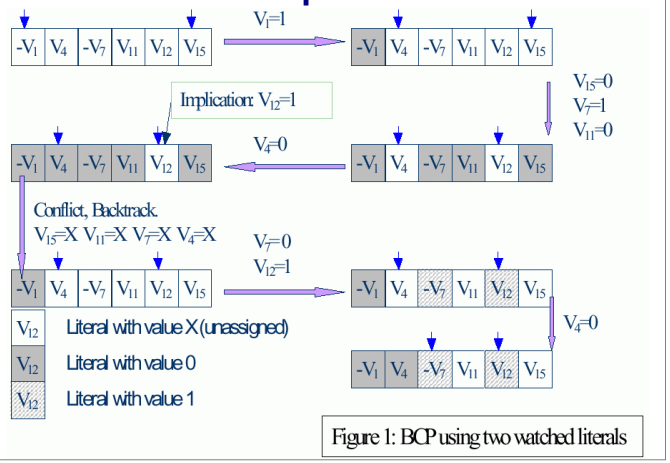



Figure 1: BCP using two watched literals

## Optimiertes BCP (2)

- Vorteile:
  - Beim Backtracking muss kein teures Update erfolgen
  - Freigeben von Variablen kann in konstanter Zeit erfolgen
    - bei Verwendung von Zählern müssten diese angepasst werden
  - Erneutes Zuweisen einer Variablen mit dem gleichen Wert benötigt tendentiell sehr wenig Zeit
    - Geringere Zahl an „Cache misses“



## Entscheidungsverfahren

- Welche Variablen sollte als nächstes ausgewählt werden?
  - Zufällige Bestimmung einer freien Variablen
  - Dynamic Largest Individual Sum (DLIS) Heuristik: Auswahl derjenigen Variable, die am häufigsten in den offenen Klauseln vorkommt (in Grasp verwendet)
    - Berechnung: O(# Literale)
  - Auswahl der Variablen, die am meisten Klauseln entscheidet, d.h. diejenige Variable, die am häufigsten in Unit Clauses vorkommt.



## Entscheidungsverfahren (2)

- Welches Qualitätsmaß sollte für Entscheidungsverfahren verwendet werden?
  - Zahl der notwendigen Entscheidungen
    - Ein Entscheidungsverfahren, bei dem wenige Entscheidungen notwendig sind, kann sehr teure BCP-Schritte erfordern
  - Zahl der beobachteten Konflikte
    - gleiches Argument
- Kosten zur Berechnung der Entscheidung können signifikant sein



## Variable State Independent Decaying Sum (VSIDS) Decision Heuristic

- Jede Variable hat für jede Polarität einen Zähler mit Anfangswert 0
- Wenn eine Klausel hinzugefügt wird, so wird der Zähler für jedes darin vorkommende Literal inkrementiert
- Die (freie) Variable und Polarität mit dem höchsten Zählerwert wird als nächstes ausgewählt
  - Ist dies nicht eindeutig, so entscheidet der Zufall
- Nach einer bestimmten Zeit werden alle Zählerwerte durch eine Konstante geteilt

## VSIDS (2)

- Eine Liste der freien Variablen, sortiert nach Zählerwerten, wird in einer STL set gespeichert
- Ziel der Strategie: Erfüllen von kürzlich hinzugefügten Konfliktklauseln
  - Konfliktklauseln geben der Suche eine Richtung
- Geringer Overhead
- Wird in *Chaff* verwendet



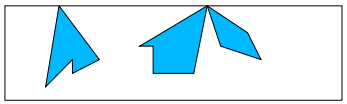
## Löschen von Klauseln

- Der Speicherverbrauch steigt enorm, wenn Klauseln immer nur hinzugefügt werden
- Unwichtig gewordene Klauseln sind zu löschen
  - Klauseln der ursprünglichen Klauselmenge dürfen nicht gelöscht werden!!!
- Strategie in Chaff:
  - Beim Anfügen einer Klausel wird bestimmt, wann sie zu löschen ist
  - Sie wird markiert, wenn zum ersten Mal mehr als N (typischerweise 100 .. 200) ihrer Literale frei werden
  - Lazy Deletion: Kompaktion geschieht später



## Randomisierung

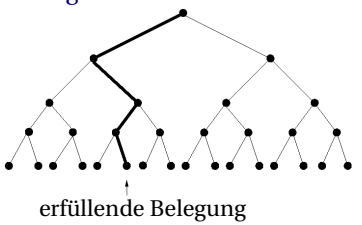
- Die Laufzeit von Backtrack-basierten SAT-Solvern hängt stark von Branching-Heuristik ab
- Geringe Änderungen des Problems können einen großen Einfluss auf die Laufzeit haben
- Randomisierung der Heuristiken führt zu starken Varianzen der Laufzeiten
  - Es werden andere Teile des Suchraumes betrachtet





## Szenario 1

- Welche Wahrscheinlichkeitsverteilung hat ein randomisiertes Verfahren?
- Annahmen:
  - Chronological Backtracking
  - Kein BCP-Solving
  - alle Pfade gleich wahrscheinlich
  - genau eine erfüllende Belegung



## Szenario 1 (2)

- Die Wahrscheinlichkeit, dass die erfüllende Belegung in polynomieller Zeit gefunden wird, ist exponentiell klein in $n$,
  - so wie $f(n) / b^{n-1}$ exponentiell klein ist für jedes Polynom $f$.
- Macht es Sinn, die Suche zwischendurch abzubrechen und neu zu beginnen, wenn zu lange nichts gefunden wurde?
  - Nein.

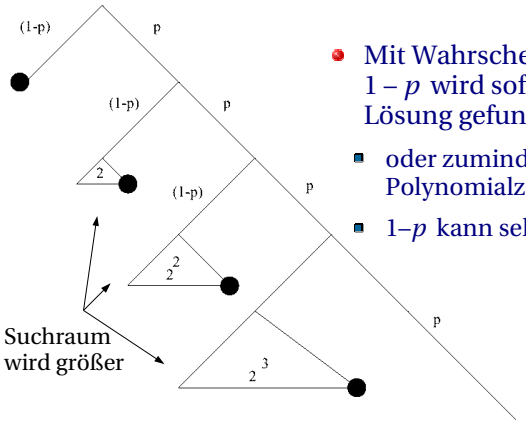


## Was wurde nicht betrachtet?

- Moderne SAT-Solver sind nicht „so dumm“
  - non-chronological backtracking
  - dynamic variable ordering
  - BCP, pruning
  - Lernen von Klauseln
- Dadurch werden viele „schlechte“ Teilbäume schnell verlassen
- Der resultierende Suchbaum ist sehr unregelmäßig



## Szenario 2




- Mit Wahrscheinlichkeit $1 - p$ wird sofort eine Lösung gefunden
  - oder zumindest in Polynomialzeit)
  - $1-p$ kann sehr klein sein



## Szenario 2: Wahrscheinlichkeitsverteilung

- Sei $0 \leq p \leq 1$ eine Wahrscheinlichkeit.
- Sei $X$ eine Zufallsvariable, die den Wert $2^i$ mit Wahrscheinlichkeit $(1\text{-}p)p^i$ annimmt, $i \geq 0$.
- Wahrscheinlichkeitsverteilung, da $\sum_{i \geq 0} (1-p)p^i = 1$
- Eigenschaften:
  - Wenn $p \geq ½$, dann ist der Erwartungswert $\infty$.
    - $EX = \sum_{i \geq 0} P[X = 2^i]2^i = \sum_{i \geq 0} (1-p)p^i 2^i = (1-p)\sum_{i \geq 0} (2p)^i$
  - Wenn $p > ¼$, dann ist die Varianz $\infty$.

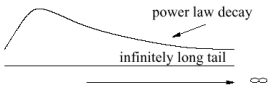
## Szenario 2: Eigenschaften

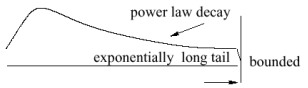
- Es kostet viel Zeit, wenn man am Anfang nicht die richtigen Entscheidungen trifft
- Macht es Sinn, die Suche zwischendurch abzubrechen und neu zu beginnen, wenn zu lange nichts gefunden wurde?
  - Ja!
- Es lässt sich beweisen, dass die erwartete Suchzeit dadurch endlich bleibt



## Szenario 3

- Suchraum hat endliche Größe
  - Erwartungswert kann nicht unendlich sein




- Unendliche Momente
- Endliche erwartete Laufzeit bei Restarts
- exponentielle Momente
- polynomielle erwartete Laufzeit bei Restarts



## Szenario 3: (2)

- Macht es Sinn, die Suche zwischendurch abzubrechen und neu zu beginnen, wenn zu lange nichts gefunden wurde?
  - Ja.
- Wie verhalten sich SAT-Solver?



## Search Restarts

- An einem bestimmten Punkt Neustart (Löschen aller Belegungen)
- Verschiedene Strategien können verwendet werden
  - Parallel auf verschiedenen CPUs
  - Nach jedem Restart kann die Strategie verändert werden

## GSAT

- Starten mit zufälliger Belegung der Variablen
- Ändern der Belegung einer Variablen, so dass dadurch am meisten Klauseln zusätzlich erfüllt werden
- Dies wird wiederholt, bis entweder
  - eine erfüllende Belegung gefunden wurde, oder
  - eine maximale Zahl an Belegungen geändert wurden
- Im zweiten Fall wird mit einer neuen Belegung weitergemacht



## GSAT (2)

```
bool GSAT(clause list S) {
    for (i = 1; i <= MAX_TRIES; ++i) {
        T = a randomly generated truth assignment
        for (j = 1; j <= MAX_FLIPS; ++j) {
            if T satisfies S, return true;
            p = a propositional variable such that a
                change in its truth assignment gives the
                largest increase in the total number of
                clauses of S that are satisfied by T.
            T = T with the truth assignment of p reversed
    }   }
    return false;
}
```



## GSAT (3)

- Schwierige 3CNF-Formeln

| formulae | | GSAT | | | DP | | |
|---|---|---|---|---|---|---|---|
| vars | clauses | flips | tries | time | choices | depth | time |
| 50 | 215 | 250 | 6,4 | 0,4s | 77 | 11 | 1,4s |
| 100 | 430 | 500 | 42,5 | 6s | 8,4E+04 | 19 | 2,8m |
| 140 | 602 | 700 | 52,6 | 14s | 2,2E+06 | 27 | 4,7h |
| 150 | 645 | 1500 | 100,5 | 45s | -- | -- | -- |
| 300 | 1275 | 6000 | 231,8 | 12m | -- | -- | -- |
| 500 | 2150 | 10000 | 995,8 | 1,6 h | -- | -- | -- |

Eine SAT-Formel ist schwierig, wenn das Verhältnis Klauseln zu Variablen etwa 4,24 ist.



## Randomisierung bei Davis-Putnam

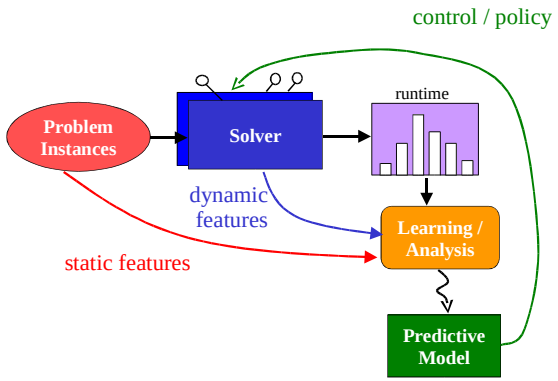
- Branching-Heuristik kann randomisiert bestimmt werden
  - Die Laufzeit von Backtrack-basierten SAT-Solvern hängt stark von Branching-Heuristik ab
  - Addieren von Zufallswerten auf die Entscheidungswerte, um diese zu beeinflussen
- Random backtracking
  - Backtracking wird auch ausgeführt, wenn dies nicht zwingend notwendig ist

## Randomisierte Restarts

- Wird von allen state-of-the-art SAT-Solvern verwendet
  - Grasp, Chaff, ...
- Gelernte Klauseln brauchen nicht immer gelöscht zu werden!



## Gesteuerte Restarts



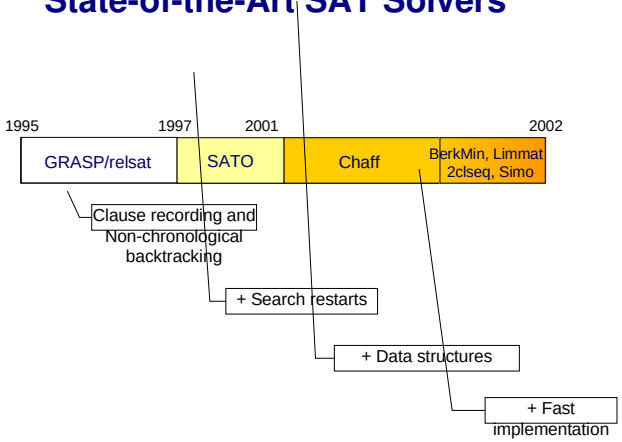



## Search Restarts: Vollständigkeit

- Restarts können zu einem unvollständigen Verfahren führen
  - Nicht-Erfüllbarkeit kann nicht mehr gezeigt werden!
- Es ist meist sehr wichtig, ein vollständiges Verfahren zu haben!
- Vollständigkeit kann erzielt werden, wenn
  - Gelernte Klauseln behalten werden
  - Die Restarts immer seltener werden



## State-of-the-Art SAT Solvers



## Chaff

- Entstand nach genauer Analyse
  - Was dominiert die Laufzeit für eine gegebene Probleminstanz?
  - Beschleunige diesen Teil
  - Iteriere, bis alles in Balance ist
- Leitendes Prinzip: Laziness
  - Verzögere Berechnungen so lange wie möglich
- Source Code ist verfügbar unter http://www.ee.princeton.edu/~chaff/zchaff.php