3. Satisfiability Checking

Content

3.1 Satisfiability-(SAT)-Checking
3.2 Variants of SAT-Checking

3. Satisfiability Checking

3.1 SAT-Checking Procedures

- Satisfiability (SAT):
  - A Boolean function \( f \) is **satisfiable**, if there exists an assignment \( A \) of Boolean constants to variables so that \( f = 1 \)
    - Example: \( f = xy + xz + r \) is satisfied by the assignment \( A: (x = 1, y = 0) \)
- Boolean functions are tautologies (\( = 1 \)), are satisfiable (SAT), or are unsatisfiable (UNSAT, \( = 0 \))
- SAT checker
  - Rather than to demonstrate the tautology \( f = 1 \), show that \( f \) is unsatisfiable
    \[
    f = a + b + \overline{a} \cdot \overline{b} = 1 \\
    f = (a + b + \overline{a} \cdot \overline{b}) = \\
    = \overline{a} \cdot \overline{b} \cdot (a + b)
    \]

3. Satisfiability Checking

3.1 SAT-checking procedures

- SAT checkers **work on conjunctive normal forms** (cnf's)
  - Example: \( (a + \overline{b} + c) \cdot (b + \overline{c} + d) \cdot (\overline{a} + d) \)
- The sum-terms are called (or)-**clauses**
- Satisfying a cnf means that all clauses have to be satisfied!
3. Satisfiability Checking

3.1 SAT-checking procedures

- Resolution (Davis/Putnam 1960)
  - Idea:
    - Assume that there are two clauses with opposing values of some variable \( x \), e.g., \( (x + a)(\bar{x} + b) \)
    - We have generally \( (x + a)(\bar{x} + b) = (x + a)(\bar{x} + b)(a + b) \)
    - Selecting \( x \) as a decision variable, we will delete \( (x + a) \) and \( (\bar{x} + b) \)
    - from our set of clauses and will continue with \( (a + b) \)
    - \((a + b)=0\) will also make \((x + a)(\bar{x} + b)=0\)
  - Systematic application to all variables
  - Unsatisfiability: 0 can be derived

- A different explanation of the resolution step:
  - In order to prove the unsatisfiability of \( f \), we apply Boole's expansion theorem to \( f \) for some variable \( x \):
    - \( f = \bar{x}f(0) + xf(1) \)
  - \( f = 0 \) iff \( f(0)+f(1) = 0 \)
  - Let \( f \) be in the form \( f = Z'(x + a)(\bar{x} + b) \) where \( Z \) stands for the rest of the cnf. Then \( f(0) = Z'a \) and \( f(1) = Z'b \). Thus, \( f(0) + f(1) = Z'(a + b) \).

- Example:
  - \( \bar{a} + c \)
  - \( b + c \)
  - \( a + c \)

- Given an assignment \( A \) of Boolean constants to variables, each clause is either:
  - Satisfied (\( = 1 \))
  - Unsatisfied (\( = 0 \))
  - Unresolved (can not be reduced to a constant)

- Satisfiability checking of a cnf \( g \):
  - Find an assignment so that all clauses of \( g \) are satisfied
  - If this is infeasible then \( g \) is equal 0

- Problem: how to organize "finding an assignment"?
3. Satisfiability Checking

3.1 SAT-checking procedures

Finding an assignment can be organized as a decision tree
(Davis/Logemann/Loveland 1962, DLL algorithm)

Example: \((\overline{b} + e) \cdot (b + \overline{c} + d) \cdot (\overline{a} + d)\)

- With the new assignment \(A: (a = 1, b = 1, d = 1)\) the first
  clause remains unresolved
- The unresolved clause \((\overline{b} + e)\) is a unit clause
  - A unit clause is an unresolved clause which has exactly
    one unassigned literal \((e\) in the example)
  - Unit clauses should be used to determine the next
    variable to be assigned

In the example, a satisfying assignment
\(A: (a = 1, b = 1, d = 1, e = 1)\) was found

- In the example, all branches terminate in conflicting assignments then
  the function is unsatisfiable
- In the worst case, an exponential number of steps has to be executed
- Practically, the selection of decision variables on the basis of detected unit clauses results in an important improvement of efficiency
3. Satisfiability Checking

### 3.1 SAT-checking procedures

- **GRASP** (Silva, Sakallah '95)
  - Systematic investigation of the implications of assignments
  - "Learning"
  - Non-chronological backtracking

- There are **direct** and **indirect** implications of assignments
  - If \( a = 0 \) was decided previously then the decision \( c = 0@i \) at a certain decision level \( i \) implies \( d = 1@i \) at the same level \( i \) for clause \( (a + c + d) \) for satisfiability
  - \( d = 1@i \) has \( e = 0@i \) as an indirect implication if we have also clause \( (d + e) \)

- The detection of all implications is called **Boolean Constraint Propagation (BCP)**

- The ordering of decisions is recorded by means of a **decision level** associated with each decision
  - We refer to a variable assignment at a certain decision level by means of "@", e.g., \( a = 1@1, b = 1@2, \) etc.

- The implications of a decision are represented by means of an **implication graph**
  - Assume \( n \) clauses \( k_1, \ldots, k_n \)
  - "Variable assignment \( v_{a1} \) implies variable assignment \( v_{a2} \) due to clause \( k_i \)"
    is represented by
    \[
    v_{a1} \xrightarrow{k_i} v_{a2} \quad \text{e.g.} \quad c = 0@i \xrightarrow{k_i} d = 1@i
    \]
3. Satisfiability Checking

3.1 SAT-checking procedures

- Example:

$$(c + d)(d + e)(e + c + f + b)(d + a + f)\ldots$$

Assume the following previous decisions:

$a = 0@1, b = 0@2, \ldots @3, \ldots @4$

Now assume the decision $c = 0@5$ at level 5.

The implication graph becomes:

- We may also view this procedure as a resolution step
  after a partial variable assignment.
  
  --- In the example we have with the partial variable
  assignment $a = b = c = 0$:

  $$(c + d)(d + e)(e + c + f + b)(d + a + f)\ldots$$
  
  $d(d + e)(e + f)(d + f)\ldots$

  $d$: $e\bar{e} + ff\ldots$

  $e$: $ff$

- We now know that the assignment $a = 0@1, b = 0@2, c = 0@5$ leads to a conflict resulting in a backtracking step $c = 1$
  
  - If the procedure makes the same decision later in the
    graph then the complex detection of all implications has
    to be repeated

  $$(c + d)(d + e)(\bar{e} + c + f + b)(d + a + f)\ldots$$

  It is possible to "learn" the conflicting assignment by
  adding the clause $(a + b + c)$:

  $$(c + d)(d + e)(\bar{e} + c + f + b)(d + a + f)(a + b + c)\ldots$$

  enforces $c=1$ for $a=0$
  and $b=0$
### 3. Satisfiability Checking

#### 3.1 SAT-checking procedures

- The "learned" clause \((a + b + c)\) is determined by the "input assignments" of the implication graph:
  - \(a = 0\) and \(b = 0\) and \(c = 0\), i.e., \(\bar{a} \cdot \bar{b} \cdot \bar{c}\) must not occur, i.e., \((\bar{a} \cdot \bar{b} \cdot \bar{c}) = (a + b + c)\)

- The detection of the conflicting assignment \(a = 0@1, b = 0@2, h = 1@3\) entails a backtracking step with revised decision \(c = 1@5\)

- Assume now that \(c = 1@5\) results again in a conflict

- Chronologically, we have to go back to level 4

- Depending on the situation, however, non-chronological backtracking can be performed, i.e., we can go back to some earlier decision level \(< 4\)

- We now know that the assignment \(a = 0@1, b = 0@2, h = 1@3\) leads to a conflict

- We non-chronologically backtrack from level 5 to level 3 and revise \(h = 1@3\)
3. Satisfiability Checking

3.1 SAT-checking procedures

- Problems:
  - With each step of backtracking, a learned clause is added
  - Computing the implications is very time-consuming

- Two-Literals Watching Schema (TLWS) in Chaff (Zhang, Malik et al. DAC'01)
  [http://citeseer.ist.psu.edu/moskewicz01chaff.html]

  - Most time is spent for BCP
  - An implication is caused only if n-1 literals are set to 0 in a clause of n literals
  - This situation can easily be detected by two pointers for each clause that point at two arbitrary literals which are not assigned to 0

  - As long as such two pointers exist, we do not have to consider implications caused by this clause

- Example:
  \[(c + \overline{d})(\overline{d} + e)(\overline{e} + c + f + b)(\overline{d} + a + \overline{f})\]...

Assume the decision: \(a = 0\)

\[(c + \overline{d})(\overline{d} + e)(\overline{e} + c + f + b)(\overline{d} + a + \overline{f})\]...

Next: \(b = 0\)

\[(c + \overline{d})(\overline{d} + e)(\overline{e} + c + f + b)(\overline{d} + a + \overline{f})\]...

Next: \(c = 0\)

\[(c + \overline{d})(\overline{d} + e)(\overline{e} + c + f + b)(\overline{d} + a + \overline{f})\]...

Implies: \(d = 0\)

\[(c + \overline{d})(\overline{d} + e)(\overline{e} + c + f + b)(\overline{d} + a + \overline{f})\]...

Satisfied clauses are not considered further, they can not cause implications

---

- For backtracking, the watches can be kept
- Backtrack to: \(a = 0, b = 1\)

\[(c + \overline{d})(\overline{d} + e)(\overline{e} + c + f + b)(\overline{d} + a + \overline{f})\]...

We have to process only those clauses where the assignment changes watched literals

We have to process 2 clauses
3. Satisfiability Checking

3.1 SAT-checking procedures

- Which variable should be decided next?
  - Random choice
  - Dynamic Largest Individual Sum (DLIS) heuristic (GRASP): select literal appearing most frequently in unresolved clauses
  - Dynamic Largest Combined Sum (DLCS) heuristic:
    - select variable appearing most frequently (positive and negative) in unresolved clauses
  - Variable State Independent Decaying Sum (VSIDS, Chaff):
    - Compute #occurrences of all literals = "activities"
    - If a learnt clause contains a literal then increment associated activity
    - Periodically divide all activities by same constant
  - ...
3. Satisfiability Checking

3.1 SAT-checking procedures

- The characteristic function $C$ of a gate-network is the product of the characteristic functions $C_i$ of the $n$ individual gates, i.e., $C = C_1 \cdot \ldots \cdot C_n$:
  - If we have a observable set of values then for each individual gate $i$ we have $x_i = f_i$. Hence, each of the individual characteristic functions equals 1 and thus the product $C$ of all characteristic functions.
  - If $C = 1$ then each of the $C_i = 1$. Thus, at each gate we have observable values.

---

3. Satisfiability Checking

3.1 SAT-checking procedures

- Example: AND-gate
  - Characteristic function:
    
    $$x = a \cdot b = x a b + x (a \cdot b) = (x + a)(x + b)(x + a + b)$$

  ![AND-gate diagram]

- Example: OR-gate
  - Characteristic function:
    
    $$y = (x + z) = (y + x)(y + z)(y + x + z)$$

  ![OR-gate diagram]

---

3. Satisfiability Checking

3.1 SAT-checking procedures

- Example: simple network
  - Characteristic function:
    
    $$(x + a)(x + b)(x + a + b) \cdot (y + x + z)(y + x + z)$$

  ![Simple network diagram]

---

3. Satisfiability Checking

3.1 SAT-checking procedures

- Example: simple network
  - Characteristic function:
    
    $$(x + a)(x + b)(x + a + b) \cdot (y + x)(y + z)(y + x + z)$$

  ![Simple network diagram]

---
3. Satisfiability Checking

3.1 SAT-checking procedures

- The number of clauses is proportional to the number of gate-inputs and -outputs
- The cost is the large number of variables (1 variable per gate-output)

3.1 SAT-checking procedures

- SAT-Checker
  - Historically developed ~1960
  - Became very efficient in the last 5-10 years
  - Very large #variables (20,000,000) and #clauses (60,000,000) are tractable
  - Successful application to many verification problems
  - Application to various industrial problems (railway interlocking systems, engine management units, ...)

3.2 SAT-Checking Variants

- SAT-based quantification
  - We observed in Chapter 1 that the universal quantification of a cnf w.r.t. a variable x is very easy: all literals of x are deleted
  - The existential quantification if difficult since this involves the or-operation of the two cofactors
    \[ \exists x: f(x) = f(0) + f(1) \]
  - If a unique representation, e.g., dnf or cnf is desired then many transformations may become necessary

- For the calculation of \( \exists x: f(x) = f(0) + f(1) \) we typically do not expect the result to be unsatisfiable, but we want to record all satisfying assignments (ALL-SAT), e.g., as dnf, OBDD, ...
- Can we benefit from SAT-checking techniques like, e.g., learning to speed up ALL-SAT?
3. Satisfiability Checking

3.2 SAT-checking variants

- A first observation is the following:
  - If a satisfying assignment A is found for one cofactor, e.g., \( f(0) \) then - for the calculation of \( \exists x: f(x) = f(0) + f(1) \) - the other cofactor has not to be investigated for A again: since \( f(0) \) is 1 for A, the values of \( f(1) \) for A are irrelevant for \( f(0) + f(1) \). A can be added as a **blocking clause**.
  - Example:

```latex
\begin{array}{c|c|c|c|}
& \text{b} & \text{f(0)} & \text{b} \\
\hline
\text{x} & \text{1} & \text{1} & \text{f(0) + f(1)} \\
\hline
\text{a} & \text{f} & & \\
\end{array}
```

- Example (cont’d):
  - \( f = (\bar{a} + \bar{b})(x + \bar{b})(a + b + \bar{x}) \)
  - \( x=0: f(0) = (\bar{a} + \bar{b})\bar{b} \) \( \Rightarrow \) satisfying assignment \( b = 0 \) \( \Rightarrow \) b is added as a **blocking clause**
  - \( x=1: f(1) = (\bar{a} + \bar{b})(a + \bar{b})b \) \( \Rightarrow \) satisfying assignment \( b = 1, a = 0 \) \( \Rightarrow \) \( \bar{b} + a \) is added as a blocking clause
  - The satisfying assignments (as dnf) are represented by the sum of the complements of the blocking clauses: \( \bar{b} + \bar{a} \)

```latex
\begin{array}{c|c|c|c|}
& \text{b} & \text{f(0)} & \text{b} \\
\hline
\text{x} & \text{1} & \text{1} & \text{f(0) + f(1)} \\
\hline
\text{a} & \text{f} & & \\
\end{array}
```

- The SAT-core problem:
  - Given a function \( f \) as cnf which is unsatisfiable, determine a subset of the clauses of \( f \) (an **unsatisfiable core**) which are also unsatisfiable
  - Example:
    \[ f = \overline{b}(b + \overline{a})(\overline{c} + \overline{a})(b + c)(\overline{a} + \overline{b} + c)(a + c) \]
    - important to determine the relevant part of a cnf which makes the cnf unsatisfiable

```latex
\begin{array}{c|c|c|c|}
& \text{b} & \text{f(0)} & \text{b} \\
\hline
\text{x} & \text{1} & \text{1} & \text{f(0) + f(1)} \\
\hline
\text{a} & \text{f} & & \\
\end{array}
```