SAT-Solving in Propositional Logics

To be SAT or to not be SAT, that is here the question!

Uwe Keller
uwe.keller@sti2.at

Semantic Technology Institute (STI) Innsbruck
University of Innsbruck, Austria

July 17, 2008

Outline

1. Propositional Logics and the SAT Problem
   - Propositional Logics
   - The SAT Problem
2. Some basic Algorithms for SAT
   - Systematic Methods
3. Multi-Strategy SAT-Solving and Parallelization

The Power of modern SAT Solver

Propositional Logic (PL) is the most fundamental logic and the SAT problem a core problem in AI with a lot of applications (e.g. circuit verification, recap: any NP-complete problem can be represented as a SAT problem!)

SAT has been studied extensively during many decades and really impressive progress has been achieved since $\sim$15 years. Hard and fairly large real-world problems can be solved by modern SAT solver.

A range of (very) different methods has been proposed to solve SAT for propositional logics. The single methods are effective on different problem sets.

However: in applications PL is often not enough, more expressive logics are either needed (or desired) to capture domain knowledge.

Examples: Verification of pipelines micro-processors, real-time or hybrid control systems, software ...

Propositional Logic

Propositional Logic (PL) is the most fundamental KR language that is essentially concerned with the meaning of logical connectors *and*, *or* and *not*

Syntax of formulae:

$$\phi \longrightarrow A \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi$$

Example: $\phi(p, q, r) = (p \vee q) \wedge r \wedge \neg(p \vee \neg r) \wedge q$

Semantics: An interpretation $I : A \rightarrow \{\top, \bot\}$ assigns meaning to all atomic statements in ϕ

Define when a formula ϕ is satisfied by I (denoted as $I \models \phi$):

$$\begin{array}{ll} I \models A & \text{iff. } I(A) = \top \\ I \models \neg\phi & \text{iff. not } I \models \phi \\ I \models \phi_1 \wedge \phi_2 & \text{iff. } I \models \phi_1 \text{ and } I \models \phi_2 \\ I \models \phi_1 \vee \phi_2 & \text{iff. } I \models \phi_1 \text{ or } I \models \phi_2 \end{array}$$

The SAT Problem

Kind of an inverse problem to evaluation of a formula:

Definition (The SAT Problem for PL)

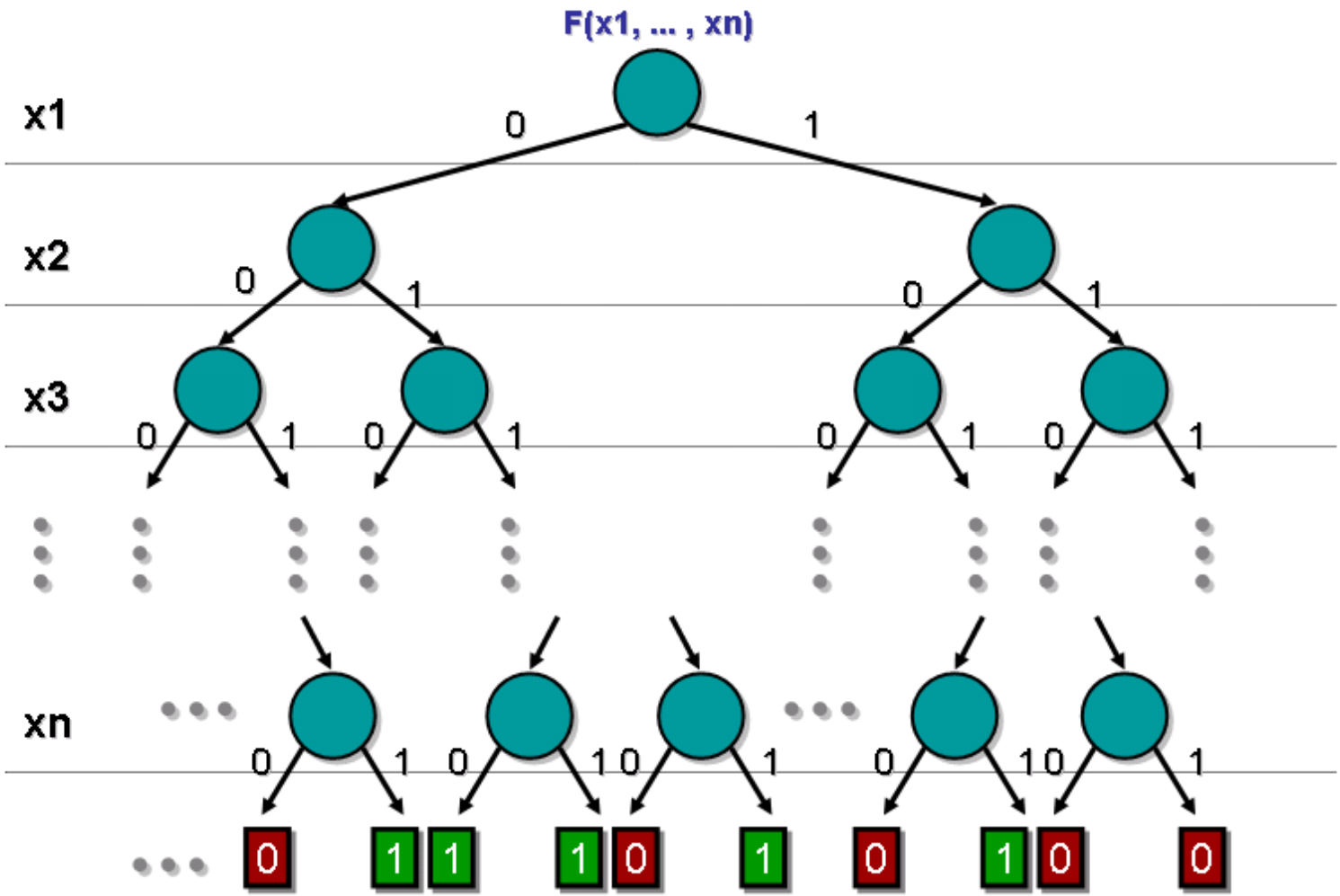
Given a formula ϕ of PL that is constructed from propositional variables $x_1, \ldots, x_n$, is there an interpretation I of $x_1, \ldots, x_n$ such that ϕ is satisfied wrt. I, i.e. $I \models \phi$?

The „origin" of all NP-complete problems

SAT is a core problem in AI and computer science!

- **How to solve it?**

Space of Possible Solutions for SAT-Solver



2^n pathes = possible candidate interpretations!

Propositional SAT Solver: Algorithms

- **Well-known methods ...**

Most naive method: **Truth Table** (simple, but not guided by ϕ)

	p	q	r	$\phi(p,q,r)$	
I_j	T	F	T	T	$\triangleleft$ Model of ϕ
I_k	T	T	T	F	

Other methods (based on truth-functional analysis of ϕ):
Analytic Tableau, Resolution

Most successful (but PL-specific) methods:
Davis-Putnam-Loveland-Longman (DPLL), Clause Learning, (RO)BDDs , WalkSAT, ...

Even more methods: UnitWalk, Complete Local Search, BED-Sat, Survey Propagation, B-Cubing, Genetic Algorithms, ...

Analytic Tableau Procedure: Underlying Principles

Underlying principles are very intuitive ...

1. Assume a truth value for ϕ as $\top$
2. Derive truth values for direct subformula of a formula ϕ from the truth value of ϕ according to the semantics of the operators (**Analytical Decomposition**)
3. Represent the different possibilities as an and-or-tree. A branch in the tree represents a specific case, and the tree is called tableau
4. A branch is closed (or contradictory), if it contains a literal and its complement. Closed branches do not have to be extended further.
5. If all branches are closed (contradictory), then the input formula is unsatisfiable. Otherwise, a non-closed but maximally expanded branch represents a model of the input formula.

The procedure does not require a specific normal form for the input problem!

Analytic Tableau Procedure: Syntactic Decomp. Rules

$$\frac{}{\top : \phi}$$

$$\frac{\top : A \wedge B}{\begin{array}{c}\top : A \\ \top : B\end{array}} \qquad \frac{\bot : A \vee B}{\begin{array}{c}\bot : A \\ \bot : B\end{array}}$$

$$\frac{\top : A \vee B}{\top : A \mid \top : B} \qquad \frac{\bot : A \wedge B}{\bot : A \mid \bot : B}$$

$$\frac{\top : A \leftrightarrow B}{\begin{array}{ll}\top : A & \mid \bot : A \\ \top : B & \mid \bot : B\end{array}} \qquad \frac{\bot : A \leftrightarrow B}{\begin{array}{ll}\top : A & \mid \bot : A \\ \bot : B & \mid \top : B\end{array}}$$

$$\frac{\top : \neg A}{\bot : A} \qquad \frac{\bot : \neg A}{\top : A}$$

$$\frac{\begin{array}{c}\top : A \\ \bot : A\end{array}}{clash}$$

Towards more efficient Methods: Semantic Decomposition

Semantic Decomposition over a variable assignment:

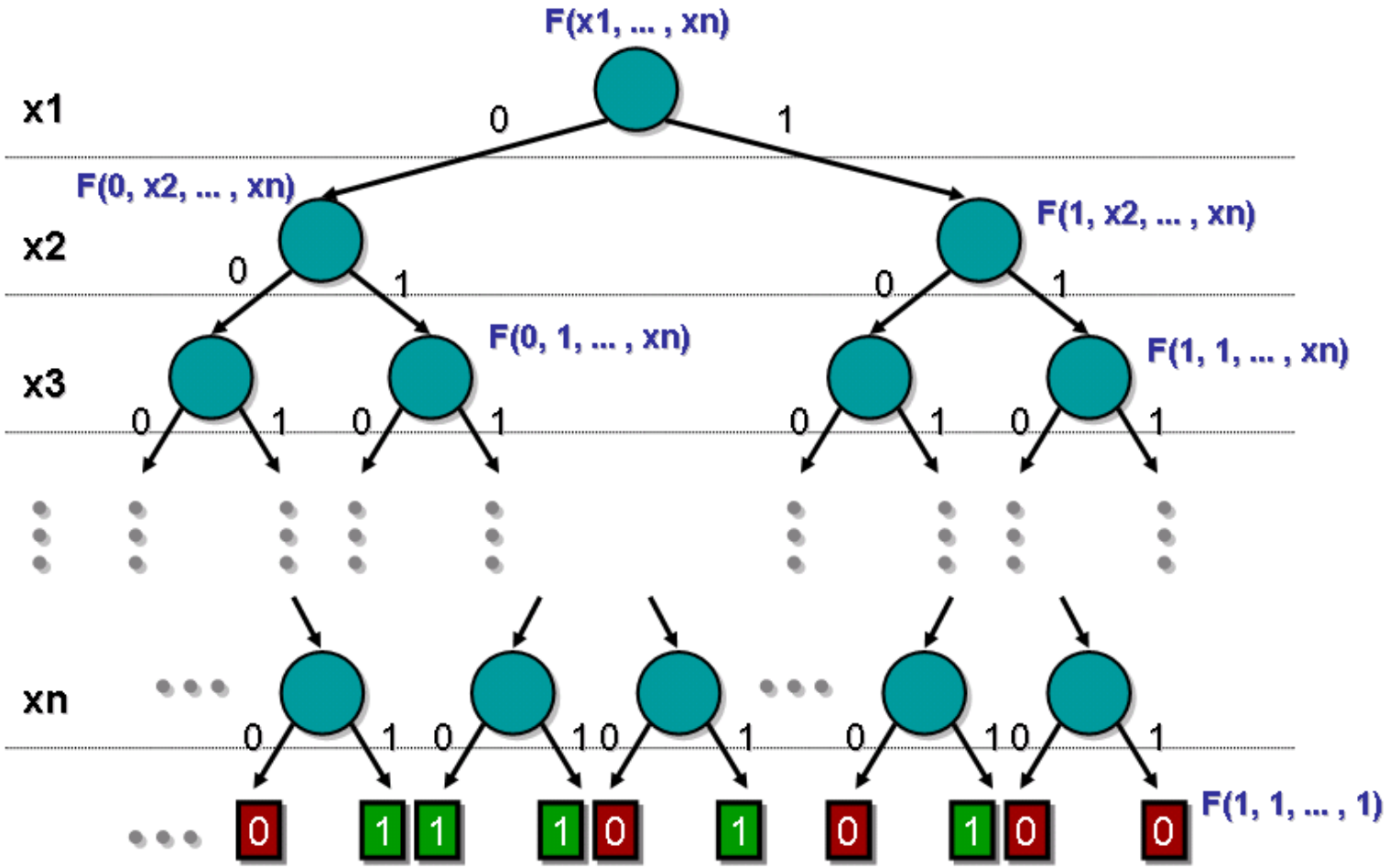
$$F(x_1, x_2, \ldots, x_n) \equiv F(\bot, x_2, \ldots, x_n) \vee F(\top, x_2, \ldots, x_n)$$

where $F(\bot, x_2, \ldots, x_n)$ represents the formula F after replacing x_1 with $\bot$ and subsequent simplification

Divide-and-Conquer Approach:

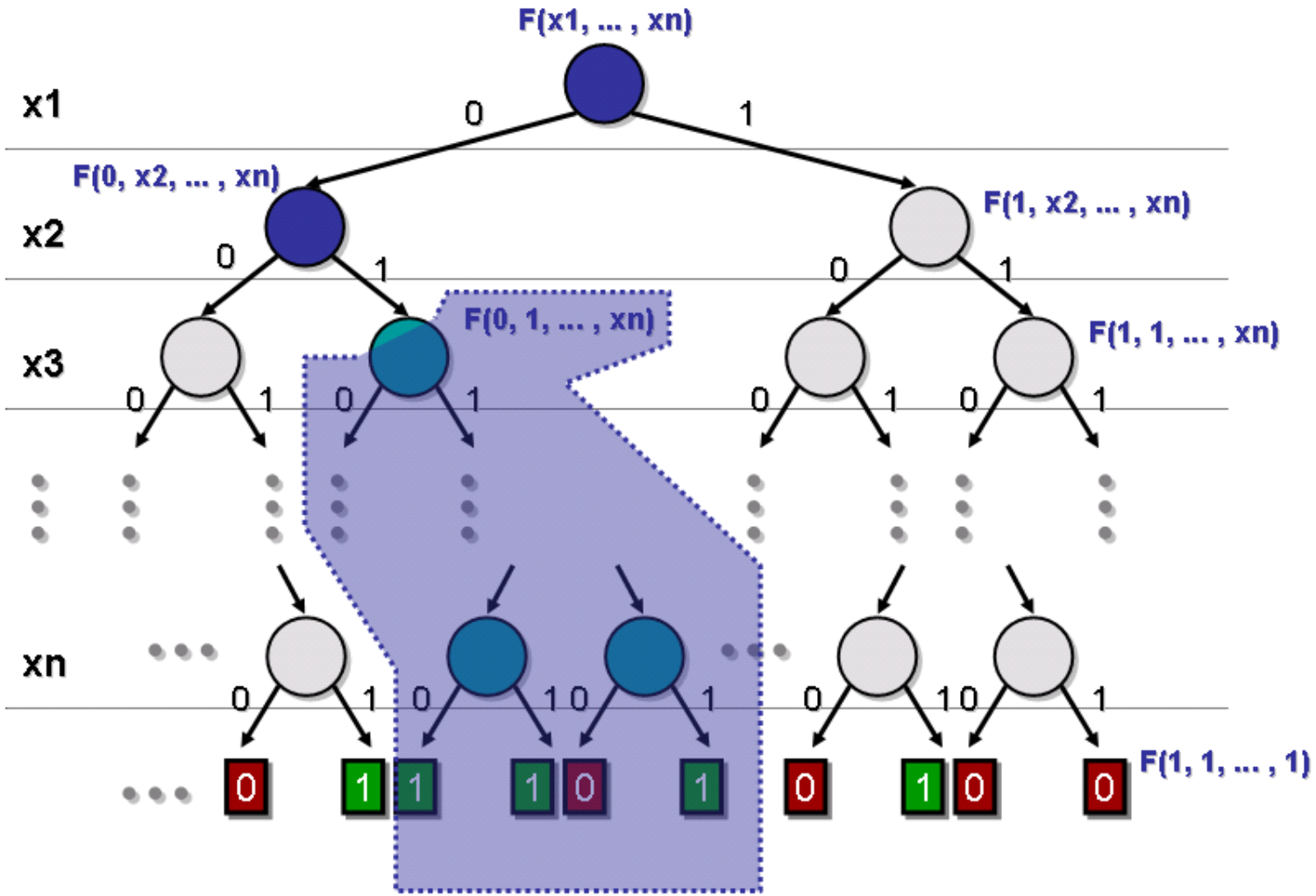
Create two independent, non-overlapping subproblems that are simpler (one variable less), solve them (in any order, concurrently, in parallel ...) and combine the result.

Towards more efficient Methods: Semantic Decomposition



2^n pathes = possible candidate interpretations!

Towards more efficient Methods: Semantic Decomposition



Binary Decision Diagrams (BDDs) [?]

- Every formula ϕ with n free prop. vars represents a n-ary boolean function $\phi(x_1, \ldots, x_n)$
- BDD = simple data structure for representing an (n-ary) boolean function $f : \{0,1\}^n \to \{0,1\}$

Definition (Binary Decision Diagram (BDD))

A *Binary Decision Diagram* is a rooted, directed, acyclic graph where leaf nodes are labeled either with 0 or 1. Each non-leaf node n has exactly two child nodes $low(n)$ and $high(n)$ and is labeled with a boolean variable $var(n)$. The edge from a node to a low (high) child represents an assignment of the variable to 0 (1).

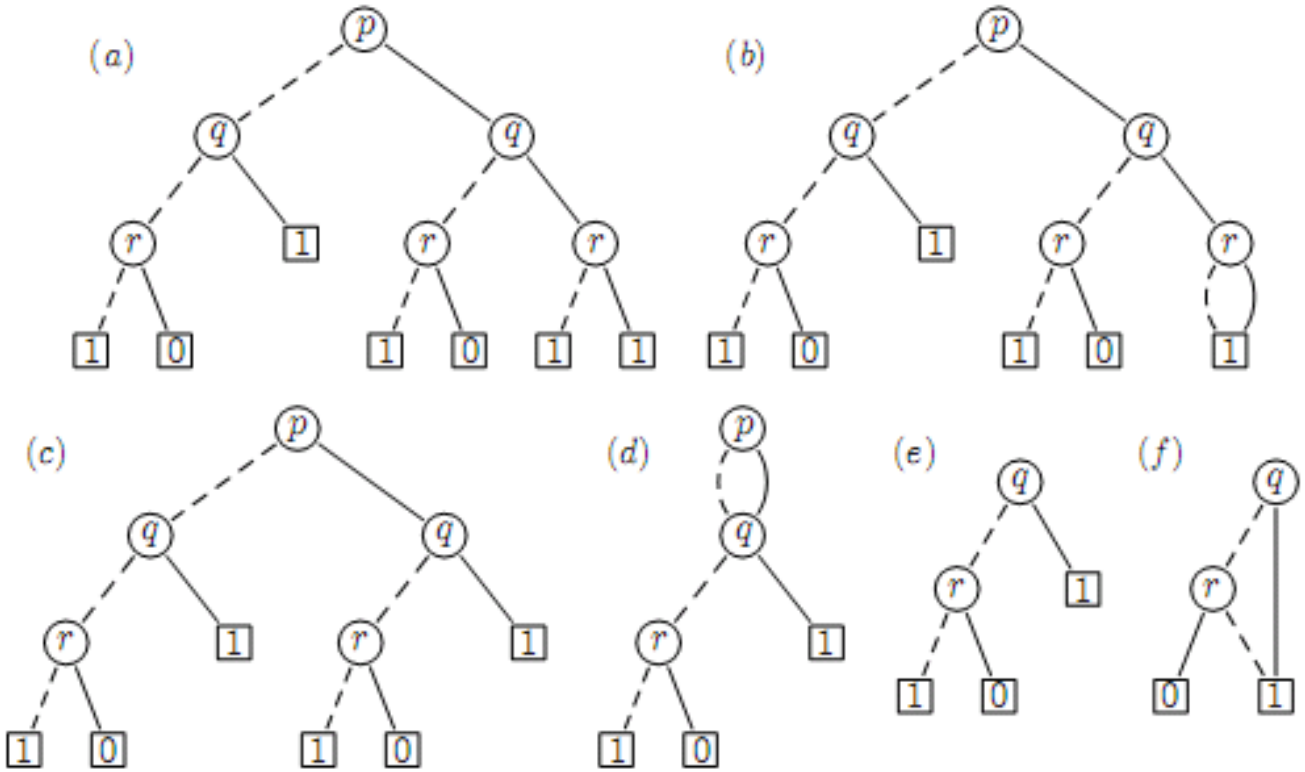
Such a BDD is called *ordered* if on each path from the root to the leaf nodes all variables are consistent with a linear order on the boolean variables. It is called *reduced* if the graph is reduced according to two rules: (i) merge any isomorphic subgraphs, and (ii) eliminate any node whose two children are the same.

In the literature: most often „BDD” actually refers to ROBDD!

What is a BDD? Representing a formula as a BDD!

Stepwise transformation to a (RO)BDD for the formula

$\phi(p,q,r) = (p \rightarrow q) \wedge r \rightarrow (p \leftrightarrow r) \wedge q$



Construction can be done efficiently (in the size of the graph), that in turn depends critically on the chosen variable order!

Properties of BDDs

- Given a fixed ordering on the variables, BDDs are a canonical representation of a boolean function, i.e. for any boolean function f there is *exactly one* ROBDD B for the given ordering s.t. $I(B) = f$
- This makes BDDs unique, semantic representations of formulae (= independent of syntax) since any two equivalent formulae have the same ROBDD.
- In practice BDDs are often small representations (e.g. polynomial) wrt. truth tables (given a suitable var ordering)
- BDDs are are semantically rich representations: $1/0$-pathes, complete coverage, DNF and CNF
- Can be used as the basis for various inference calculus dealing with cubes or clauses, especially tableau- and resolution-like inferences, *even at the same time*
- Have been used very successfully to represent and deal with large-scale problems in various areas of computer science.

A historic note on the Davis-Putnam / DPLL procedure

A quote from Martin Davis (Invited Talk at SAT 2006)

During the summer of 1957, Hilary Putnam and I, both junior faculty, were attending an unprecedented month-long "institute" devoted to logic at Cornell University along with 82 other logicians. Our families were sharing a house and the two of us were together every day working together and separately on a number of things, but not on the satisfiability problem. After we had made some progress towards a negative solution of Hilbert's 10th Problem (H10: the question of the existence of an algorithm for determining whether a given polynomial equation has an integer solution), we were eager to continue collaborating. Our idea was to seek funding through my institution which was a branch of Rensselaer Polytechnic in Eastern Connecticut so Hilary and his family could escape steamy summers in Princeton for the attractive lake side accommodations available in my locale. Not believing that anyone would pay us to work on H10, considered a super long shot, we patched together a proposal to investigate procedures for theorem-proving in first-order logic. Because it was too late for the usual funding agencies, following a tip we submitted our proposal to the National Security Agency. They funded it on condition that our report not mention them, and that we forget about first-order logic, and just concentrate on satisfiability. Our report, which was submitted at the end of the summer of 1958, contained all the procedures that were eventually combined in the algorithms later designated as DP and DPLL. During the summer of 1959, we were supported by the US Air Force Office of Scientific Research.We worked very hard on H10 and made some significant progress. But because our proposal had emphasized theorem-proving procedures,we hastily concocted one using some of the work from the previous summer, and submitted it to the JACM. That was the origin of Davis-Putnam. After I moved to New York, I wanted to see our procedure implemented, and NYU put two very talented student programmers at my disposal for the purpose: Donald Loveland (who later became one of my first doctoral students) and George Logemann. The crude search we implemented led to satisfiability questions involving thousands of clauses and the original DP swamped the memory of the IBM 704. So we replaced the "rule for eliminating propositional variables" (i.e. ground binary resolution) with the splitting rule giving the algorithm a "divide and conquer" form with instances waiting to be processed swapped out onto a tape. This was the DPLL algorithm.

SAT-Solving by Search and Semantic Decomposition

Most direct approach to check satisfiability of $F(x_1, \ldots, x_n)$

- Fix an order π of the variables : $x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}$
- **Guess**: For the next variable $x_{\pi(j)}$ select a truth value t_j from $\{\top, \bot\}$
- Determine the satisfiability of $F(x_1, \ldots, x_n)\{x_{\pi(j)} \leftarrow t_j\}$ (in the same way). If the subproblem is satisfiable, we know that F is satisfiable.
- Otherwise: Determine the satisfiability of $F(x_1, \ldots, x_n)\{x_{\pi(j)} \leftarrow \overline{t_j}\}$ (in the same way). If the subproblem is satisfiable, we know that F is satisfiable.
- Otherwise: F is unsatisfiable.

Space-efficiency: this can be implemented (on a sequential machine) as a depth-first search with backtracking.

However: guessing happens unnecessarily often!
If we can deduce a truth value that must necessarily hold for a variable x_j then we do not need to guess (and consider two cases). This reduces the potential search space by half for every deduced assignment!

To simplify the algorithm: Convert formula in conjunctive normal form (CNF) ($\simeq$ **DPLL procedure**)

The DPLL Procedure: Example 1

Formula to check for satisfiability: $\phi(p,q,r) = (p \vee q) \wedge r \wedge \neg(p \vee \neg r) \wedge q$

(1) Convert ϕ to CNF: $CNF(\phi) = \{p \vee q\} \wedge \{r\} \wedge \{\neg p\} \wedge \{q\}$
Note: the search space consists of $2^3 = 8$ possible interpretations for $\{p,q,r\}$

(2) Perform DPLL search on $CNF(\phi)$:

$F_0 := \{p \vee q\} \wedge \{r\} \wedge \{\neg p\} \wedge \{q\}$ and $I_0 := \{\}$

(a) Are we done with F_0? No! No empty clause, and still clauses left!

(b) Derive determ. variable assignments from F_0:
Unit clauses $\{r\}$, $\{\neg p\}$ and $\{q\}$ enforce $I_1 := I_0 \cup \{r, \neg p, q\}$

(c) Simplify F_0 according to the inferred variable assignments:
$F_1 := \{\bot \vee \top\} \wedge \{\top\} \wedge \{\top\} \wedge \{\top\} = \top$

(d) Are we done with F_1? Yes! No clauses are left which can not be be assigned a truth value under I_1. Since $F_1 = \top$, $I_1 = \{r, \neg p, q\}$ is a model of $CNF(\phi)$ and hence for ϕ.

Observation: No guessing was needed. Semantic analysis of $CNF(\phi)$ lead deterministically to the (only) model for $CNF(\phi)$!

The DPLL Procedure: Example 2

Formula to check for satisfiability:
$\phi(p,q,r,s) = (p \leftrightarrow q) \wedge (\neg p \leftrightarrow q) \wedge (r \vee s)$ (UNSAT)

(1) Convert ϕ to CNF:
$CNF(\phi) = \{\neg p \vee q\} \wedge \{\neg q \vee p\} \wedge \{\neg q \vee \neg p\} \wedge \{p \vee q\} \wedge \{r \vee s\}$
Note: the search space consists of $2^4 = 16$ possible interpretations for $\{p,q,r,s\}$

(2) Perform DPLL search on $CNF(\phi)$:

$F_0 := \{\neg p \vee q\} \wedge \{\neg q \vee p\} \wedge \{\neg q \vee \neg p\} \wedge \{p \vee q\} \wedge \{r \vee s\}$ and $I_0 := \{\}$

Observation: Using a „suitable" variable ordering (e.g. MOMS: $p, \neg p, q, \neg q \prec r, s$), we only construct (therefore consider) 2 cases (i.e. partial truth assignments) instead of 16 possible candidate interpretations.

The DPLL Procedure: Recursive Algorithm

`function` satisfiable(Formula ϕ);
$\mathcal{C}$:= convert-to-cnf(ϕ);
result := DPLL-search($\mathcal{C}$,$\emptyset$, 0);
if *result* = $(SATISFIABLE, \star)$ **then**
  | **return** true;
else
  | **return** false;
end

`function` assign(Literal l, ClauseSet $\mathcal{C}$);
$\mathcal{C}'$:= $\mathcal{C} \setminus \{C \mid l \in C\}$;
$\mathcal{C}''$:= $\{C' \mid C \in \mathcal{C}', C' = C \setminus \{\neg l\}\}$;
return $\mathcal{C}''$;

The DPLL Procedure: Recursive Algorithm

```
function DPLL-search(ClauseSet C, AssignedLiterals μ, CurrentDecisionLevel current-level);
if C = ∅ then
  |   return (SATISFIABLE, μ);
end
if {} ∈ C then
  |   blevel := analyze-conflict(C, μ, current-level);
  |   C := learn-conflict-clause(C, μ, current-level);
  |   return (BACKTRACK, blevel, C);
end
if unit clause {l} occurs in C then
  |   return DPLL-search(assign(l,C), μ ∪ {l}, current-level);
end
l := choose-literal(C,μ);
status := DPLL-search(assign(l,C), μ ∪ {l}, current-level + 1);
if status = (SATISFIABLE, μ) then
  |   return status;
end
if status = (BACKTRACK, blevel, C) and blevel = current-level then
  |   if current-level = 0 then
  |     |   return UNSATISFIABLE
  |   else
  |     |   return DPLL-search(assign(¬l,C ∪ C), μ ∪ {¬l}, current-level + 1);
  |   end
else
  |   return status;
end
```

Optimization: Conflict Analysis

Analysis of the Implication Graph from the current Assignment Stack:
Non-chronological backtracking and Conflict Clause Learning

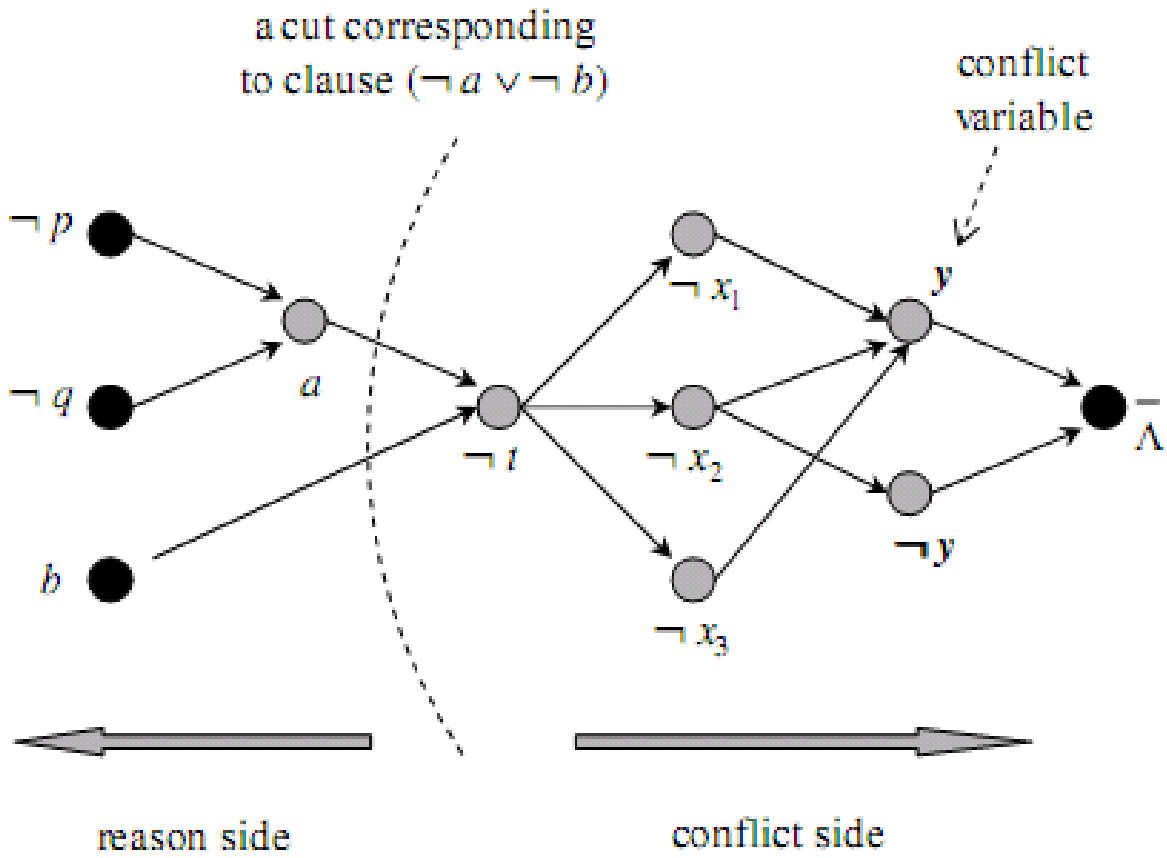


Figure 1.1: A conflict graph

Combining SAT Techniques to Complex Strategies

There are a range of different algorithms to approach the prop. SAT problem

Often they are specifically effective on different problem sets

The SAT problem is an NP-complete problem, hence a „perfect" method that works equally well in all cases is unlikely to exist

Idea: It might be useful to combine different SAT solving algorithms to more complex procedures in a flexible way!

How to do that? Consider the different SAT solving methods as atomic strategies and combine them by functional composition into more complex ones. The single atomic steps in a complex strategy can be derived from generic algorithms by fixing a set of parameters (e.g. runtime, considered depth, ...) to bound resources

This allows to construct domain-specific procedures from generic building blocks (Customization from pre-existing components)

What is needed to realize the idea?

We basically need to answer a few questions:

(1) For each atomic component to be integrated: what are useful resource-bounding parameters and what output is generated for what input given the resource bound **(Parametrization and Component Semantics)**

(2) How can we represent / comprehend an atomic strategy in an abstract, uniform way? What information shall be exchanged between subsequently executed strategies? **(Interface)**

(3) How can we describe complex composition pattern of atomic strategies? **(Composition Description Language: Syntax, Semantics)**

Interface Definition for Strategies

From Tacticals and Tactics in Automated Theorem Proving: Strategies take as input a proof state and constructs a (set of) proof state(s).

Hence, we consider a strategy (uniformly) as a function

$$(State \times Parameters) \rightarrow (List < State > \times ResultStatus)$$

Proof state formally allows different strategy components to exchange information that has been gathered during their execution

Parameters enable to set the resource-bounds for the strategy, resulting in a strategy instance

$$State \rightarrow (List < State > \times ResultStatus)$$

$ResultStatus \in \{SAT, UNSAT, INDET\}$ with the convention that $INDET$ is returned precisely if the respective proof state list is non-empty.

Hence: a strategy invocation either decides the problem directly or splits the problem in a number of subproblems that need to be solved subsequently

Interface Definition for Strategies

What is a proof state: Contains a description of what problems need to be solved (Objectives), and a description of what can safely be assumed (e.g. given previously applied strategies) in order to do so (Assumptions)

$$s = (\{O_1, \ldots, O_k\}, A = \{E_1, \ldots, E_m\})$$

Objectives are formulae $O_1, \ldots, O_k$, Assumptions are represented as a set $A = \{E_1, \ldots, E_m\}$ of equivalence classes E_j of subformulae of the objectives and their complements.

Semantics: A proof state is satisfiable iff the following formula is satisfiable:

$$\bigwedge_{i=1,\ldots,k} O_k \wedge \bigwedge_{E \in A} \bigwedge_{\phi_1, \phi_2 \in E} (\phi_1 \leftrightarrow \phi_2)$$

Initial Proof State: $(\{\phi\}, \emptyset)$

Interface Definition for Strategies

Correctness of a strategy:

For all applications to proof states s the strategy fulfills two criteria:

- If the list l of returned proof states is empty, then the returned outcome correctly diagnoses s as satisfiable or unsatisfiable
- If the list l of returned proof states is non-empty, then some proof state $p' \in l$ is satisfiable iff. p is satisfiable

We can distinguish the following classes of strategies:

- **Decisive** – $l = \emptyset$
- **Refining** – s is a singleton list containing the original proof state with some extra assumptions
- **Splitting** – s has more than one element

To ensure that we can solve our initial goal, we need to ensure that our overall strategy always returns a decisive result.

Composing Strategies

A few examples of basic strategies (used in the EDA domain):

$$dpll : (State \times Time \times StratInst) \rightarrow (List < State > \times ResultStatus)$$

$$cut : (State \times Time \times Level \times StratInst) \rightarrow (List < State > \times ResultStatus)$$

$$saturate : (State \times Time \times Degree \times StratInst) \rightarrow (List < State > \times ResultStatus)$$

$$inst : (State \times Time \times StratInst \times StratInst) \rightarrow (List < State > \times ResultStatus)$$

$$lemma : (State \times Time \times StratInst) \rightarrow (List < State > \times ResultStatus)$$

Example for a Basic Strategy: Variable Instantiation

Basic Idea: Sometimes it is possible to instantiate a variable v in a formula ϕ by a particular truth value, without changing the satisfiability of the problem (equi-satisfiable transformation).

This can be very helpful, since the resulting problem $\phi\{v \leftarrow \top\}$ or $\phi\{v \leftarrow \bot\}$ then is simpler (contains one variable less)

Key question: When can we safely instantiate a variable v to $\top$?

Proof the validity of the following formula

$$\phi\{v \leftarrow \bot\} \rightarrow \phi\{v \leftarrow \top\}$$

If we can show the validity using some (cheap) SAT procedure (e.g. 1-saturation), we instantiate the variable v to $\top$.

If not, we check (in the same way) if the instantiation of v with $\bot$ is safe and perform a respective safe instantiation

We can do this iteratively for any unassigned variable in the proof state and return a proof state that contains the simplified objective formulae and a suitably extended assumption relation.

Composing Strategies

Defining Strategy instances by fixing parameter):

dpllInst(s) = $dpll(s, 100, trivial)$

cut-and-dpll(s) = $cut(s, -1, 20, dpllInst)$

Description of Complex Strategies: (Domain Specific) Scripting Language

cut 10;
instantiate(dpll 10);
sat 1; dpll 100;

Allowing for more sophisticated langauge constructs (if-then-else, loops) is possible

Effectiveness of Composing Strategies

An evaluation on industrial benchmark problems shows the effectiveness of the approach:

Instance	PROVER CL (s)	ZCHAFF (s)
ibm-1	0.5	2.5
ibm-2	0.0	0.0
ibm-3	0.8	0.9
ibm-4	1.2	4.3
ibm-5	0.2	0.4
ibm-6	5.2	14.6
ibm-7	0.2	0.2
galileo-8	7.0	92.3
galileo-9	8.6	140.2
ibm-10	7.1	404.2
ibm-11	3.5	34.8
ibm-12	16.4	165.4
ibm-13	7.4	10.1

Table 2: SATLIB benchmark results

We can observe: Speed-up and more balanced behaviour over the benchmark problems

Why is this interesting for LarKC?

Strategy Composition for SAT Solving can showchase some fundamental features of the LarKC platform:

- Reasoning using multiple plug-ins that need to be orchestrated in a suitable way (Meta-level interpreter as a separate plugin)
- Generated subproblem in principle can be solved in parallel, results need to be combined
- Extra-Plug-in Parallelization vs. Intra-Plug-in Parallelization: Plug-in may or may not be parallelized
- Algorithms for SAT solving are relatively simple and various usable libraries exist
- Propositional SAT is a practically relevant problem and a component of reasoning in many more expressive formalisms, e.g. DL Reasoning (so not completely useless to consider)
- Strategy description shows that the platform is customizable, domain specific strategies can be defined if needed
- Strategies could as well be automatically discovered for specific domains (e.g. evolutionary algorithms / genetic programming) ($\sim$ a form of Meta-level Optimization)