Heuristic Optimization

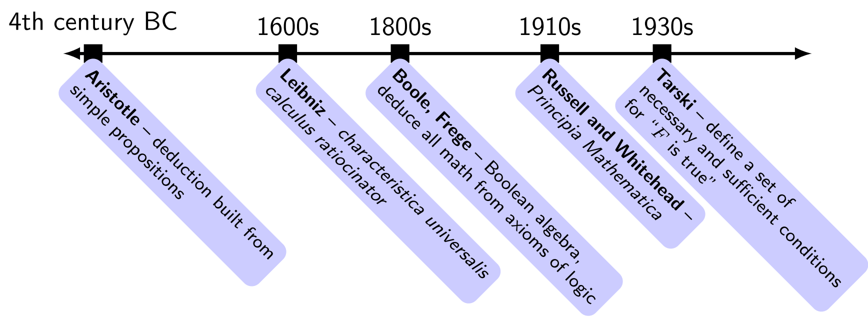
Lecture 8

Algorithm Engineering Group
Hasso Plattner Institute, University of Potsdam

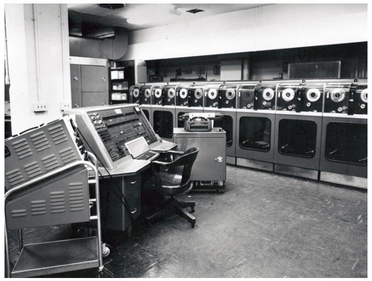
9 June 2015

The SATISFIABILITY problem

Quest throughout history to establish an **effective process** (e.g., a *mechanical process*) for human reasoning.



The SATISFIABILITY problem



UNIVAC (www.computerhistory.org)

In the 20th century, the advent of computers inspired mathematicians to

- try to understand what people do when they create proofs
- reduce logical reasoning to some canonical form that can be implemented by an algorithm

Given a statement S in some well-defined logical syntax

- is there an algorithm to prove S is true (or false)?
- what is the complexity of such an algorithm?

DEC VT100 (commons.wikimedia.org)

SATISFIABILITY: A formal definition

A **propositional logic formula** is built from

- *variables* that can take on one of two values (true/false) $x, y, z, \ldots$
- *operators* $\{\wedge, \vee, \neg\}$
  - conjunction (logical AND), e.g., $x \wedge y$
  - disjunction (logical OR), e.g., $x \vee y$
  - negation (logical NOT), e.g., $\neg x$
- *parentheses* that can group expressions, e.g., $(x) \wedge (\neg x \vee y)$

A formula F is said to be *satisfiable* if it can be made true by assigning appropriate logical values (true or false) to its variables.

Problem: given a formula, F, decide whether F is satisfiable.

Many applications: theoretical computer science, complexity theory, algorithmics, cryptography and artificial intelligence.

SATISFIABILITY: Basics

A **well-formed** Boolean expression can be described by the grammar:

⟨*expr*⟩ ::= ⟨*variable*⟩
| ⟨*expr*⟩ ∧ ⟨*expr*⟩
| ⟨*expr*⟩ ∨ ⟨*expr*⟩
| (⟨*expr*⟩)
| ¬⟨*expr*⟩

The *assignment* of a Boolean variable v is a binding to a value in $\{0, 1\}$.

If all variables in an expression are bound, the evaluation can be done recursively:

E	F	$E \wedge F$	$E \vee F$	(E)	$\neg E$
0	0	0	0	0	1
0	1	0	1	0	1
1	0	0	1	1	0
1	1	1	1	1	0

Definitions

The assignment of n Boolean variables can be represented as $x \in \{0, 1\}^n$.

Let F be a formula on n variables. We write $F[x] \in \{0, 1\}$ as the evaluation of F under the assignment $x \in \{0, 1\}^n$.

Given a Boolean expression F on n Boolean variables, we say an assignment $x \in \{0, 1\}^n$ *satsifies* F if $F[x] = 1$.

Example

$$F = (\neg x_1 \vee x_2) \wedge \neg x_1 \wedge (\neg x_3 \vee \neg x_1)$$
$$x = (0, 0, 0), F[x] = 1$$
$$x = (1, 0, 1), F[x] = 0$$

Definitions

Two Boolean formulas E and F on n Boolean variables are said to be *equivalent* if $\forall x \in \{0, 1\}^n$, $F[x] = E[x]$. In this case we write $F \equiv E$

A *literal:* a variable v or its negation $\neg v$. A *clause:* a disjunction of literals, e.g., $(x_1 \vee \neg x_2 \vee \neg x_3 \vee \cdots \vee x_i)$

A formula F is said to be in *conjunctive normal form* (CNF) when F is written as a conjunction of clauses.

Lemma

For every well-formed formula F, there is a formula E such that (1) E is in CNF, and (2) $F \equiv E$.

CNF form is much easier to work with!

Is SATISFIABILITY easy or hard?

Let $\mathscr{F}$ be the set of all admissible formulas. We consider some subsets of $\mathscr{F}$:

- $\mathscr{F}_1$ formulas satisfied when all variables are set true (false).
- $\mathscr{F}_2$ formulas $F \equiv E$, where E is in CNF and each clause contains at most one positive (resp., negative) literal. (Horn formulas)
- $\mathscr{F}_3$ formulas $F \equiv E$, where E is in CNF and each clause contains ≤ 2 literals. (2-CNF formulas)
- $\mathscr{F}_4$ formulas $F \equiv E$, where E is conjunction of exclusive-or clauses. (Affine formulas)

Schaefer's Dichotomy Theorem (1978)

1. Every formula $F \in (\mathscr{F}_1 \cup \mathscr{F}_2 \cup \mathscr{F}_3 \cup \mathscr{F}_4)$ can be decided in time polynomial in the length of F.
2. The class $\mathscr{F} \setminus (\mathscr{F}_1 \cup \mathscr{F}_2 \cup \mathscr{F}_3 \cup \mathscr{F}_4)$ is NP-complete.[a]

[a] **Technical note:** Schaefer's approach is constrained to classes that can be recognized in log space.

Resolution for first-order logics

1958 Martin Davis & Hilary Putnam developed a resolution procedure for first-order logic (quantifiers allowed)

Herbrand's theorem: if a first-order formula is *unsatisfiable* then it has some ground formula in *propositional logic* (quantifier-free) that is unsatsifiable.

Davis-Putnam procedure

1. Generate all propositional ground instances
2. Check if each instance F is satisfiable

The main innovation is in (2), where we must solve SATISFIABILITY

Given a propositional logic formula F in CNF, assign variables using three *reduction rules*.

Davis-Putnam procedure

Rule 1: unit rule

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (x_1) \wedge (\neg x_1 \vee x_4 \vee x_3) \wedge (\neg x_2)$$

$$(x_2 \vee \neg x_3) \wedge (x_4 \vee x_3) \wedge (\neg x_2)$$

Reduced formula: $(\neg x_3) \wedge (x_4 \vee x_3)$

set $x_1 = 1$ set $x_2 = 0$

For each **unit clause** (ℓ)

- set ℓ to 1, $\neg\ell$ to 0
- remove clauses containing ℓ, delete occurrences of $\neg\ell$
- repeat until no unit clauses exist

Davis-Putnam procedure

Rule 2: pure literal rule

pure literal

$$(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_4) \wedge (x_3 \vee \neg x_4)$$

pure literal

Reduced formula: $(x_3 \vee \neg x_4)$

set $x_1 = 1$ set $x_2 = 0$

For each **pure literal** ℓ

- set ℓ to 1, $\neg\ell$ to 0
- remove clauses containing ℓ
- repeat until no pure literals exist

Davis-Putnam procedure

Rule 3: rule for eliminating atomic formulas *(ground resolution)*

$$(x \vee \ell_{1,1} \vee \cdots \vee \ell_{1,k_1}) \wedge (\neg x \vee \ell_{2,1} \vee \cdots \vee \ell_{2,k_2}) \wedge C$$

$$(\ell_{1,1} \vee \cdots \vee \ell_{1,k_1} \vee \ell_{2,1} \vee \cdots \vee \ell_{2,k_2}) \wedge C$$

When $x = 0$, $(\ell_{1,1} \vee \cdots \vee \ell_{1,k_1})$ must be true

When $(x = 1$, $\ell_{2,1} \vee \cdots \vee \ell_{2,k_2})$ must be true

Replace $(x \vee A) \wedge (\neg x \vee B) \wedge C$ with $(A \vee B) \wedge C$ as long as there are no $\ell_{1,i} \in A$, $\ell_{2,j} \in B$ that are complementary

Using memory wisely

In 1962, Loveland and Logemann tried to implement DP procedure on an IBM 704, but found that it used too much RAM.

L&L insight: keep a stack for formulas in external storage (tape drive) so the formulas in RAM don't get too large.

IBM 704 at NASA in 1957 (commons.wikimedia.org)

Rule 3a: splitting rule

From $(x \vee A) \wedge (\neg x \vee B) \wedge C$, create a pair of separate formulas[a]

$$(A \wedge C), (B \wedge C).$$

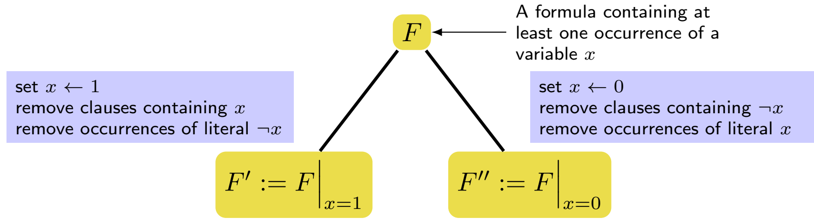
Recursively check $(A \wedge C)$ and $(B \wedge C)$ for satisfiability.

[a] where A, B and C don't contain any occurrences of the variable x

A closer look at the **splitting rule:**

$$(x \vee A) \wedge (\neg x \vee B) \wedge C \xRightarrow{\text{split}} (A \wedge C), (B \wedge C)$$

$$F \xRightarrow{\text{split}} F', F''$$



Observation:

- If F' or F'' contain an empty clause: then unsatisfied
- If F' or F'' contain no clauses: then satisfied

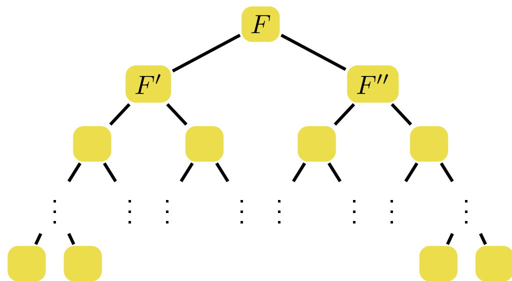
Davis-Putnam-Logemann-Loveland (DPLL)

Davis-Putnam procedure with Logemann-Loveland enhancement (splitting rule)

DPLL(F)

Input: A set of clauses F
Output: A truth value
if F *is a consistent set of literals* **then return** true;
if F *contains an empty clause* **then return** false ;
for *each unit clause* (ℓ) *in* F **do**
  $F \leftarrow$ `unit-propagate`(ℓ, F);
end
for *each pure literal* ℓ *in* F **do**
  $F \leftarrow$ `pure-literal-assign`(ℓ, F);
end
$\ell \leftarrow$ `choose-literal`(F);
return DPLL$(F \wedge \ell) \vee$ **DPLL**$(F \wedge \neg\ell)$;

DPLL search space



Total size of search tree? $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

How can we reduce the total number of nodes expanded?

DPLL heuristics: Branching policies

Pick a good variable on which to branch

Come up with a *scoring function* $\text{score}(\ell)$ that gives a value for picking a variable that makes ℓ true.

Some scoring functions:

$\max(\ell)$ # occurrences of ℓ in F.

Idea: Picking ℓ to maximize $\max(\ell)$ satisfies as many clauses as possible.

$\text{moms}(\ell)$ # occurrences of ℓ in F appearing in clauses of minimum size.

Idea: reducing minimum clauses can lead to a unit-propagation sooner or reveal a contradiction faster

$\text{mams}(\ell)$ $:= \max(\ell) + \text{moms}(\neg\ell)$.

Idea: satisfy as many clauses as possible, create as many minimum-size clauses as possible

DPLL heuristics: Branching policies

Jeroslow-Wang: $\text{jw}(\ell) := \sum_{C \ni \ell} 2^{-|C|}$.

Idea: exponential weighting: smaller clauses have more weight than larger ones.

$\text{up}(\ell)$ # of unit propagations triggered by setting $\ell = true$.

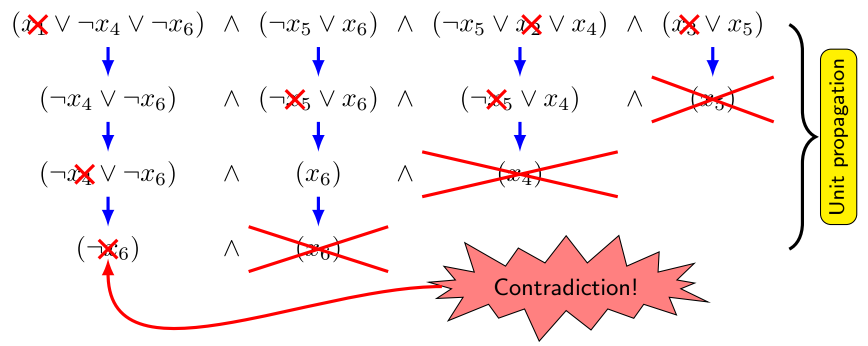
adaptive learning: adapt branching rule during execution

DPLL heuristics: Clause learning

When unit propagation results in a conflict (produces an empty clause),

- analyze the unit propagation process that resulted in the conflict
- add a new clause to the formula that explains and prevents repeating the same conflict later in the search

branches taken so far: set $x_1 = 0$ set $x_2 = 0$ set $x_3 = 0$



DPLL heuristics: Clause learning

branches taken so far: set $x_1 = 0$ set $x_2 = 0$ set $x_3 = 0$

We can conclude the branch $x_1 = 0, x_2 = 0, x_3 = 0$ leads to an unsatisfied formula
In other words,

$$
\begin{aligned}
&(x_1 = 0) \wedge (x_2 = 0) \wedge (x_3 = 0) \implies (F = 0) \\
\equiv &(F = 1) \implies \neg((x_1 = 0) \wedge (x_2 = 0) \wedge (x_3 = 0)) \quad \text{(contrapositive)} \\
\equiv &(F = 1) \implies (x_1 = 1) \vee (x_2 = 1) \vee (x_3 = 1)
\end{aligned}
$$

So in order for F to be satisfied, $(x_1 \vee x_2 \vee x_3)$ must be true.

Learned clause: $F' := F \wedge (x_1 \vee x_2 \vee x_3)$

Note: many very sophisticated procedures for analyzing the structures of contradictions exist.

A local search algorithm

DPLL: construct an assignment from scratch

Another approach: start from a complete assignment. While not satisfied, make some small change. Repeat.

Random local search algorithm for SATISFIABILITY

Choose $x \in \{0,1\}^n$ uniformly at random;
while *F is not satisfied* **do**
    $y \leftarrow x$;
    Choose $C \in F$ not satisfied by x;
    Choose a literal $\ell \in C$ uniformly at random;
    Let i be the index such that $\{x_i, \neg x_i\} \ni \ell$;
    $y[i] \leftarrow 1 - y[i]$;
end

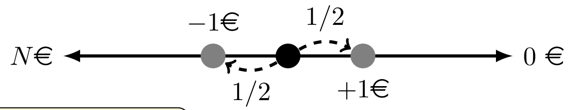
How efficient is the random local search algorithm?

Theorem. (Papadimitriou, 1991)

Let $F \in \mathscr{F}_3$ (formulas that have at most two literals per clause). If F is satisfiable, then the local search algorithm finds the satisfying assignment in $O(n^2)$ time in expectation.

Proof sketch.

Gambler's ruin



Expected flips until win/loss: $O(N^2)$

- Let $x^\star$:= satisfying assignment, x := be the current assignment.
- For any clause $C \in F$ not satisfied by x, at least one of the values $x[i]$ doesn't match the value in $x^\star[i]$.
- Probability to pick that variable $\geq 1/2$.
- Move closer to $x^\star$ with probability $\geq 1/2$ (further away w/ prob. $\leq 1/2$). □

k-CNF formulas

What about k-CNF formulas for $k > 2$?

Run local search algorithm, starting from a new random solution every $O(n)$ steps.

Theorem. (Schöning, 1991)

Let F be a k-CNF formula. If F is satisfiable, then the (restarting) local search algorithm finds the satisfying assignment in T steps where T is within a polynomial factor of $(2(1-1/k))^n$.

For 3-CNF formulas: $(1.333)^n$

Current best-known bound[1] for 3-SAT: $O(1.308^n)$

[1] Timon Hertli, FOCS 2011