Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Español
Français
Italiano
Nederlands
日本語
Português
Русский
Српски / srpski
中文
Edit links

DPLL algorithm

From Wikipedia, the free encyclopedia

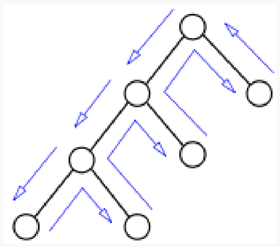
In computer science, the **Davis–Putnam–Logemann–Loveland** (**DPLL**) **algorithm** is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form, i.e. for solving the CNF-SAT problem.

It was introduced in 1962 by Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland and is a refinement of the earlier Davis–Putnam algorithm, which is a resolution-based procedure developed by Davis and Putnam in 1960. Especially in older publications, the Davis–Logemann–Loveland algorithm is often referred to as the "Davis–Putnam method" or the "DP algorithm". Other common names that maintain the distinction are DLL and DPLL.

After almost 50 years the DPLL procedure still forms the basis for most efficient complete SAT solvers. It has recently been extended for automated theorem proving for fragments of first-order logic.[1]

Contents [hide]

DPLL



Class	Boolean satisfiability problem
Worst case performance	$O(2^n)$
Worst case space complexity	$O(n)$ (basic algorithm)

Implementations and applications [edit]

The SAT problem is important both from theoretical and practical points of view. In complexity theory it was the first problem proved to be NP-complete, and can appear in a broad variety of applications such as *model checking*, automated planning and scheduling, and diagnosis in artificial intelligence.

As such, it was and still is a hot topic in research for many years, and competitions between SAT solvers regularly take place.[2] DPLL's modern implementations like Chaff and zChaff,[3][4] GRASP or Minisat[5] are in the first places of the competitions these last years.

Another application which often involves DPLL is automated theorem proving or satisfiability modulo theories (SMT) which is a SAT problem in which propositional variables are replaced with formulas of another mathematical theory.

The algorithm [edit]

The basic backtracking algorithm runs by choosing a literal, assigning a truth value to it, simplifying the formula and then recursively checking if the simplified formula is satisfiable; if this is the case, the original formula is satisfiable; otherwise, the same recursive check is done assuming the opposite truth value. This is known as the *splitting rule*, as it splits the problem into two simpler sub-problems. The simplification step essentially removes all clauses which become true under the assignment from the formula, and all literals that become false from the remaining clauses.

The DPLL algorithm enhances over the backtracking algorithm by the eager use of the following rules at each step:

Unit propagation

If a clause is a *unit clause*, i.e. it contains only a single unassigned literal, this clause can only be satisfied by assigning the necessary value to make this literal true. Thus, no choice is necessary. In practice, this often leads to deterministic cascades of units, thus avoiding a large part of the naive search space.

Pure literal elimination

If a propositional variable occurs with only one polarity in the formula, it is called *pure*. Pure literals can always be assigned in a way that makes all clauses containing them true. Thus, these clauses do not constrain the search anymore and can be deleted.

Unsatisfiability of a given partial assignment is detected if one clause becomes empty, i.e. if all its variables have been assigned in a way that makes the corresponding literals false. Satisfiability of the formula is detected either when all variables are assigned without generating the empty clause, or, in modern implementations, if all clauses are satisfied. Unsatisfiability of the complete formula can only be detected after exhaustive search.

The DPLL algorithm can be summarized in the following pseudocode, where Φ is the CNF formula:

```
Algorithm DPLL
  Input: A set of clauses Φ.
  Output: A Truth Value.
```

```
function DPLL(Φ)
   if Φ is a consistent set of literals
       then return true;
   if Φ contains an empty clause
       then return false;
   for every unit clause l in Φ
      Φ ← unit-propagate(l, Φ);
   for every literal l that occurs pure in Φ
      Φ ← pure-literal-assign(l, Φ);
   l ← choose-literal(Φ);
   return DPLL(Φ ∧ l) or DPLL(Φ ∧ not(l));
```

- "←" is a shorthand for "changes to". For instance, "*largest* ← *item*" means that the value of *largest* changes to the value of *item*.
- "**return**" terminates the algorithm and outputs the value that follows.

In this pseudocode, `unit-propagate(l, Φ)` and `pure-literal-assign(l, Φ)` are functions that return the result of applying unit propagation and the pure literal rule, respectively, to the literal `l` and the formula `Φ`. In other words, they replace every occurrence of `l` with "true" and every occurrence of `not l` with "false" in the formula `Φ`, and simplify the resulting formula. The **`or`** in the **`return`** statement is a short-circuiting operator. `Φ ∧ l` denotes the simplified result of substituting "true" for `l` in `Φ`.

The pseudocode DPLL function only returns whether the final assignment satisfies the formula or not. In a real implementation, the partial satisfying assignment typically is also returned on success; this can be derived from the consistent set of literals of the first `if` statement of the function.

The Davis–Logemann–Loveland algorithm depends on the choice of *branching literal*, which is the literal considered in the backtracking step. As a result, this is not exactly an algorithm, but rather a family of algorithms, one for each possible way of choosing the branching literal. Efficiency is strongly affected by the choice of the branching literal: there exist instances for which the running time is constant or exponential depending on the choice of the branching literals. Such choice functions are also called heuristic functions or branching heuristics.[6]

Current work [edit]

In the 2010s years, work on improving the algorithm has been done on three directions:

1. Defining different policies for choosing the branching literals.
2. Defining new data structures to make the algorithm faster, especially the part on **unit propagation**
3. Defining variants of the basic backtracking algorithm. The latter direction include *non-chronological backtracking* (aka. *backjumping*) and *clause learning*. These refinements describe a method of backtracking after reaching a conflict clause which "learns" the root causes (assignments to variables) of the conflict in order to avoid reaching the same conflict again. The resulting Conflict-Driven Clause Learning SAT solvers are the state of the art in 2014.

A newer algorithm from 1990 is Stålmarck's method. Also since 1986 (reduced ordered) binary decision diagrams have also been used for SAT solving.

Relation to other notions [edit]

Runs of DPLL-based algorithms on unsatisfiable instances correspond to tree resolution refutation proofs.[7]

See also [edit]

- Davis–Putnam algorithm
- Chaff algorithm
- Proof complexity
- Herbrandization

References [edit]

General

- Davis, Martin; Putnam, Hilary (1960). "A Computing Procedure for Quantification Theory". *Journal of the ACM* **7** (3): 201–215. doi:10.1145/321033.321034.
- Davis, Martin; Logemann, George; Loveland, Donald (1962). "A Machine Program for Theorem Proving". *Communications of the ACM* **5** (7): 394–397. doi:10.1145/368273.368557.
- Ouyang, Ming (1998). "How Good Are Branching Rules in DPLL?". *Discrete Applied Mathematics* **89** (1–3): 281–286. doi:10.1016/S0166-218X(98)00045-6.
- John Harrison (2009). *Handbook of practical logic and automated reasoning*. Cambridge University Press. pp. 79–90. ISBN 978-0-521-89957-4.

Specific

1. ^ Nieuwenhuis, Robert; Oliveras, Albert; Tinelly, Cesar (2004), "Abstract DPLL and Abstract DPLL Modulo Theories" (PDF), *Proceedings Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2004*, pp. 36–50
2. ^ *The international SAT Competitions web page*, sat! live
3. ^ *zChaff website*
4. ^ *Chaff website*
5. ^ "Minisat website".
6. ^ Marques-Silva, João P. (1999). "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms". In Barahona, Pedro; Alferes, José J. *Progress in Artificial Intelligence: 9th Portuguese Conference on Artificial Intelligence, EPIA '99 Évora, Portugal, September 21–24, 1999 Proceedings*. LNCS **1695**. pp. 62–63. doi:10.1007/3-540-48159-1_5. ISBN 978-3-540-66548-9.
7. ^ Peter Van Beek (2006). "Backtracking search algorithms". In Francesca Rossi, Peter Van Beek, Toby Walsh. *Handbook of constraint programming*. Elsevier. p. 122. ISBN 978-0-444-52726-4.

Further reading [edit]

- Malay Ganai; Aarti Gupta; Dr. Aarti Gupta (2007). *SAT-based scalable formal verification solutions*. Springer. pp. 23–32. ISBN 978-0-387-69166-4.
- Carla P. Gomes, Henry Kautz, Ashish Sabharwal, Bart Selman (2008). "Satisfiability Solvers". In Frank Van Harmelen, Vladimir Lifschitz, Bruce Porter. *Handbook of knowledge representation*. Foundations of Artificial Intelligence **3**. Elsevier. pp. 89–134. doi:10.1016/S1574-6526(07)03002-7. ISBN 978-0-444-52211-5.

Categories: Constraint programming | Automated theorem proving | SAT solvers