March, 2020

Algorithms in Backtracking Search behind SAT and ASP

Yuliya Lierler

Handout on Algorithms in Backtracking Search behind SAT and ASP

Yuliya Lierler
University of Nebraska Omaha

Introduction

We now turn out attention to search algorithms underlying ASP technology. In particular, we will focus on the techniques employed by answer set solver such as CLASP. Recall that CLASP is only one building block of an answer set system CLINGO that also incorporates grounder called GRINGO. In the scope of this course we ignore the details behind grounders, but note that these are highly nontrivial systems solving a complex and computationally intense task of *intelligent* instantiation.

The algorithms behind majority answer set solvers fall into group of so called backtracking search algorithms.

> Backtracking is a general algorithm for finding all (or some) solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution. *(Wikipedia)*

The Davis-Putnam-Logemann-Loveland (DPLL) procedure is a classic example of backtracking search algorithms. DPLL is a method for deciding the satisfiability of propositional logic formula in conjunctive normal form, or, in other words, for solving the propositional satisfiability problem. Algorithms used by answer set solvers share a lot in common with DPLL. In this handout, we thus begin by presenting DPLL procedure. We then discuss its extensions suitable for computing answer sets of a program in place.

1 Satisfiability Solving: Davis-Putnam-Logemann-Loveland Procedure

Recall that a *literal* is an atom or a negated atom. A signature is a set of atoms. Given a propositional formula, the set of atoms occurring in it is considered to be its signature by default. A *clause* is a disjunction of literals (possibly the empty disjunction $\bot$). A formula is said to be in *conjunctive normal form (CNF)* if it is a conjunction of clauses (possibly the empty conjunction $\top$). The task of deciding whether a CNF formula is satisfiable is called a satisfiability (SAT) problem. Recall that an *interpretation/assignment* over a signature is a mapping from the elements of the signature to truth values $\mathtt{f}$ or $\mathtt{t}$. For example, given formula

$$(p \wedge q) \vee r \tag{1}$$

there are 8 interpretations in its signature $\{p, q, r\}$ including the following

interpretation	p	q	r
I_1	f	t	t
I_2	f	t	f

A formula is called *satisfiable* if we can find an interpretation over its signature so that this formula is evaluated to true under this interpretation. (We assume the familiarity with the interpretation functions for the classical logic connectives $\top, \bot, \neg, \wedge$ and $\vee$.) We say that in such case an interpretation *satisfies* a formula and also call it a *model*. For instance, interpretation I_1 satisfies formula (1) while I_2 does not. In other words, I_1 is a model of formula (1). Hence this formula is also satisfiable. It is common to identify an interpretation over signature σ with the set of literals and also with the set of atoms in an intuitive way. For instance, the table below presents such a mapping for interpretations I_1 and I_2.

interpretation	p	q	r	set of literals	set of atoms w.r.t. $\sigma = \{p, q, r\}$
I_1	f	t	t	$\{\neg p, q, r\}$	$\{q, r\}$
I_2	f	t	f	$\{\neg p, q, \neg r\}$	$\{q\}$

Later in the discourse we frequently use the word interpretation to denote a set of literals.

1.1 DPLL by means of Pseudocode

The Davis-Putnam-Logemann-Loveland (DPLL) procedure is an algorithm for deciding the satisfiability of propositional logic formula in CNF. DPLL also allows to find a satisfying interpretation of a formula if it exists. Enhancements of DPLL form modern SAT solving technology.

We now state some terminology useful in presenting DPLL. For a literal of the form A we say that $\neg A$ is its *complement*, whereas for a literal of the form $\neg A$, atom A is its *complement*. A set M of literals is *consistent* when it does not contain complementary pairs $A, \neg A$. Otherwise, we call a set *inconsistent*. Sets $\{a, b, \neg c\}$ and $\{a, b, c, \neg c\}$ exemplify consistent and inconsistent sets of literals, respectively. In the sequel by *wrt* we abbreviate the phrase *with respect to*. We say that a clause $l_1 \vee \cdots \vee l_n$ is *unit-ary wrt* the set M of literals, when

- there is i such that $0 \leq i \leq n$ and $l_i \notin M$ (we call literal l_i a *unit literal*), and
- for every j such that $0 \leq j \leq n$ and $j \neq i$, $\overline{l_j} \in M$.

For instance, $a \vee \neg b \vee c$ is a unit-ary clause wrt $\{\neg a, b\}$ and c is its unit literal. Clause $a \vee \neg b$ is a unit-ary clause wrt set $\{\neg a, b\}$, where both a and $\neg b$ are unit literals. We say that a clause $l_1 \vee \cdots \vee l_n$ is *satisfied by* the set M of literals, when for some i such that $0 \leq i \leq n$ the following holds $l_i \in M$. We say that a literal l is *unassigned* by a set M of literals if neither l nor its complement $\bar{l}$ is in M; otherwise we say that literal l is *assigned* by M.

Consider the procedure called *unit propagation* presented in Figure 1. This procedure is invoked on a consistent set M of literals. To apply unit propagation to a given CNF formula F, Unit-Propagate is invoked with F and $M = \emptyset$. For instance, to apply unit propagation to

$$p \wedge (\neg p \vee \neg q) \wedge (\neg q \vee r) \tag{2}$$

we invoke Unit-Propagate with this formula as F and with $\emptyset$ as M. After the first execution of the body of the loop,

$$M = \{p\};$$

UNIT-PROPAGATE(F, M)

while M *is a consistent set of literals*,

and F has a unit-ary clause wrt M so that l is a unit literal of that clause

$M \leftarrow M \cup \{l\}$

end

Figure 1: Unit propagation

DPLL(F, M)

UNIT-PROPAGATE(F, M);

if M *is an inconsistent set of literals* **then** return;

if every atom occurring in F is assigned by M **then** exit with *a model* of M;

$l \leftarrow$ a literal containing an atom from F unassigned by M;

DPLL$(F, M \cup \{l\})$;

DPLL$(F, M \cup \{\bar{l}\})$

Figure 2: Davis-Putnam-Logemann-Loveland procedure

after the second iteration

$$M = \{p, \neg q\}.$$

This computation shows that any model of the given formula is such that p is assigned to t and q is assigned to f.

There are two cases when the process of unit propagation alone is sufficient for solving the satisfiability problem for given F. Consider the value of M upon the termination of UNIT-PROPAGATE$(F, \emptyset)$. First, if F is such that all of its clauses are satisfied by M, as in the example above, then F is satisfiable, and any satisfying interpretation can be easily extracted from M. In the example above

$$M = \{p, \neg q\}$$

gives rise to two models of (2):

interpretation	p	q	r
I_1	t	f	f
I_2	t	f	t

Second, if M is an inconsistent set of literals then F is not satisfiable.

Problem 1. *Use unit propagation to decide whether the formula*

$$p \wedge (p \vee q) \wedge (\neg p \vee \neg q) \wedge (q \vee r) \wedge (\neg q \vee \neg r)$$

is satisfiable.

The Davis-Putnam-Logemann-Loveland procedure presented in Figure 2 is an extension of the unit propagation method that can solve the satisfiability problem for *any* CNF formula. Like UNIT-PROPAGATE, it is initially invoked with F and $M = \emptyset$.

Example 1. *Consider, for instance, the application of the DPLL procedure to*

$$(\neg p \vee q) \wedge (\neg p \vee r) \wedge (q \vee r) \wedge (\neg q \vee \neg r). \tag{3}$$

First DPLL *is called with this formula as F and with $\emptyset$ as M (Call 1). After the call to* UNIT-PROPAGATE*, the value of M remains the same. Assume that the literal selected as l is p. Now* DPLL *is called recursively with F and $\{p\}$ as M (Call 2). After the call to* UNIT-PROPAGATE*, M turns into an inconsistent set $\{p, q, r, \neg r\}$ (or an inconsistent set $\{p, q, r, \neg q\}$). Thus* DPLL *returns. Next* DPLL *is called with $\{\neg p\}$ as M (Call 3). After the call to* UNIT-PROPAGATE*, M remains the same. Assume that the literal selected as l is q. Then* DPLL *is called with $\{\neg p, q\}$ as M (Call 4). After the call to* UNIT-PROPAGATE*, $M = \{\neg p, q, \neg r\}$. Since M is consistent and assigns every atom occurring in F,* DPLL *returns M as a model:*

p	q	r
f	t	f

Problem 2. *How would this computation be affected by selecting $\neg p$ as l in Call 1? By selecting $\neg q$ as l in Call 3?*

1.2 DPLL by means of Transition Systems

In the previous section we described the DPLL procedure using pseudocode. Here we use a different method to present this algorithm. In particular, we use a *transition system* that can be viewed as an abstract representation of the underlying DPLL computation. This transition system captures what "states of computation" are, and what transitions between states are allowed. In this way, a transition system defines a directed graph such that every execution of the DPLL procedure corresponds to a path in this graph. Some edges may correspond to unit propagation steps, some to decision, some to backtracking. Later in the handout, we will follow this approach for describing a search algorithm suitable for computing answer sets of a program.

For a set σ of atoms, a *record* relative to signature σ is a sequence $l_1 \dots l_{n-1}\ l_n$ of distinct literals over σ, with some literals possibly annotated by Δ, which marks them as *decision* literals, so that

1. $l_1 \dots l_{n-1}$ contains no complementary pairs $A, \neg A$, and
2. a decision literal may not be preceded by the complement of this literal in the sequence.

A *state* relative to σ is either a distinguished state *FailState* or a record relative to σ. For instance, the states relative to a singleton set $\{p\}$ are

$$\mathit{FailState},\ \ \emptyset,\ \ p,\ \ \neg p,\ \ p^\Delta,\ \ \neg p^\Delta,\ \ p\ \neg p,\ \ p^\Delta\ \neg p,\ \ \neg p\ p,\ \ \neg p^\Delta\ p, .$$

Note how sequences of literals such as $p\ p$, $p\ p^\Delta$, $p\ \neg p^\Delta$, $p^\Delta\ \neg p^\Delta$ do not form records (the former two sequences are not formed by distinct literals; the later two sequences do not satisfy Condition 2). Similarly, while sequence $p\ q\ \neg q$ forms a record relative to signature $\{p, q\}$, sequence $p\ q\ \neg q\ \neg p$ is not a record (it does not satisfy Condition 1).

Frequently, we identify a record M with a set of literals, ignoring both the annotations and the order among its elements. This allows us to use notation stemming from set theory. For example, let M be a record $p\ q\ \neg q$, we identify an expression $p \in M$ with the condition $p \in \{p, q, \neg q\}$ that "checks" whether p is a member of set $\{p, q, \neg q\}$ Similarly we can speak of literals being unassigned by a record, or a record being inconsistent. These terms were defined earlier in the handout for the sets of literals. For instance, states $p^\Delta\ \neg p$ and $p\ q\ \neg p$ are inconsistent. Also both q and $\neg q$ are unassigned by state $p^\Delta\ \neg p$, whereas both of them are assigned by $p\ q\ \neg p$.

Each CNF formula F determines its *DPLL graph* DP_F. The set of nodes of DP_F consists of the states relative to the signature of F. The edges of the graph DP_F are specified by four transition rules:

$$
\begin{array}{llll}
\textit{Unit Propagate}: & M \ \Rightarrow\ M\ l & \text{if} & \begin{cases}\text{there is a unit-ary clause in } F \text{ w.r.t. } M \text{ so that}\\ \text{literal } l \text{ is its unit literal}\end{cases}\\[2ex]
\textit{Decide}: & M \ \Rightarrow\ M\ l^\Delta & \text{if} & l \text{ is unassigned by } M\\[2ex]
\textit{Fail}: & M \ \Rightarrow\ \textit{FailState} & \text{if} & \begin{cases} M \text{ is inconsistent, and}\\ M \text{ contains no decision literals}\end{cases}\\[2ex]
\textit{Backtrack}: & P\ l^\Delta\ Q \Rightarrow\ P\ \bar{l} & \text{if} & \begin{cases} P\ l^\Delta\ Q \text{ is inconsistent, and}\\ Q \text{ contains no decision literals.}\end{cases}
\end{array}
$$

A node (state) in the graph is *terminal* if no edge originates in it. The transition rule *Unit Propagate* is also often called a *propagator/inference* rule of DPLL.

The following proposition gathers key properties of the graph DP_F.

Proposition 1. *For any CNF formula F,*

(a) graph DP_F is finite and acyclic,

(b) any terminal state of DP_F other than FailState is a model of F,

(c) FailState is reachable from $\emptyset$ in DP_F if and only if F is unsatisfiable.

Thus, to decide the satisfiability of a CNF formula F it is enough to find a path leading from node $\emptyset$ to a terminal node M. If $M = \textit{FailState}$, F is unsatisfiable. Otherwise, F is satisfiable and M is a model of F.

For instance, let $F_1 = \{p \vee q, \neg p \vee r\}$. Below we show a path in DP_{F_1} with every edge annotated by the name of the transition rule that gives rise to this edge in the graph:

$$\emptyset \quad \stackrel{\textit{Decide}}{\Rightarrow} \quad p^\Delta \quad \stackrel{\textit{Unit Propagate}}{\Rightarrow} \quad p^\Delta\ r \quad \stackrel{\textit{Decide}}{\Rightarrow} \quad p^\Delta\ r\ q^\Delta. \tag{4}$$

The state $p^\Delta\ r\ q^\Delta$ is terminal. Thus, Proposition 1(b) asserts that F_1 is satisfiable and $\{p, r, q\}$ is a model of F_1. Another path in DP_{F_1} that leads us to concluding that set $\{p, r, q\}$ is a model of F_1 follows

$$\emptyset \quad \stackrel{\textit{Decide}}{\Rightarrow} \quad p^\Delta \quad \stackrel{\textit{Decide}}{\Rightarrow} \quad p^\Delta\ r^\Delta \quad \stackrel{\textit{Decide}}{\Rightarrow} \quad p^\Delta\ r^\Delta\ q^\Delta. \tag{5}$$

We can view a path in the graph DP_F as a description of a process of search for a model of a formula F by applying transition rules of the graph. Therefore, we can characterize an algorithm of a SAT solver that utilizes the inference rules of DP_F by describing a strategy for choosing a path in DP_F. A strategy can be based, in particular, on assigning priorities to some or all transition rules of DP_F, so that a solver will never apply a transition rule in a state if a rule with higher priority is applicable to the same state. The DPLL algorithm can be captured by the following priorities:

$$\textit{Backtrack}, \textit{Fail} >> \textit{Unit Propagate} >> \textit{Decide}.$$

Note how path (6) in the graph DP_{F_1} respects priorities above, while path (5) does not. Thus DPLL will never explore the latter search trajectory given input F_1.

Problem 3. *Let G be formula (3). Then a pass in DP_G that can be seen as capturing the computation of DPLL described in Example 1 follows:*

$$\begin{array}{l} \emptyset \overset{Decide}{\Rightarrow} p^\Delta \overset{Unit\ Propagate}{\Rightarrow} p^\Delta\ q \overset{Unit\ Propagate}{\Rightarrow} p^\Delta\ q\ r \overset{Unit\ Propagate}{\Rightarrow} \\ p^\Delta\ q\ r\ \neg q \overset{Backtrack}{\Rightarrow} \neg p \overset{Decide}{\Rightarrow} \neg p\ q^\Delta \overset{Unit\ Propagate}{\Rightarrow} \neg p\ q^\Delta\ \neg r \end{array} \tag{6}$$

*(a) List an alternative path to (6) in DP_G that also can be seen as capturing the computation of DPLL. (*Hint*: think of nondeterminism in* Unit-Propagate *procedure.)*

(b) Consider node q in graph DP_G. List all the edges that leave this node in DP_G. Annotate these edges by transition rules that they are due. Specify nodes to which these edges lead. For instance,

$$q \overset{Decide}{\Rightarrow} q\ p^\Delta$$

is one of these edges. (c) Consider node $p^\Delta\ q\ r\ \neg q$ in graph DP_G. List all the edges that leave this node in DP_G (as in the previous question).

2 From ASP to SAT

A number of transformations from logic programs under answer set semantics to SAT exist. Given a propositional logic program Π, there are two kinds of transformations:

- transformations that preserve the vocabulary of Π and form a propositional theory F_Π that is *equivalent* to Π. In other words, models of Π and F_Π coincide.
- transformations that may contain "new atoms" so that the answer sets for Π can be obtained by removing these atoms from the models of constructed F_Π.

Remarkable transformation of the former kind is called *completion*. For a large syntactic class of programs ("tight" programs), the models of program's completion coincide with the answer sets of a program. This fact is exploited in several state-of-the-art answer set solvers including CLASP (a solver of CLINGO). For example, for tight programs CLASP practically runs a (significantly enhanced) DPLL procedure on program's completion to obtain answer sets of a program.

Answer Set Solving. We are now ready to present an extension to the DPLL algorithm that captures a family of backtrack search procedures for finding answer sets of a propositional logic program.

Recall that a *propositional logic program* is a finite set of *rules* of the form

$$a_0 \leftarrow a_1, \dots, a_k, \mathit{not}\ a_{k+1}, \dots, \mathit{not}\ a_m, \tag{7}$$

where a_0 is a propositional atom or symbol $\bot$; $a_1, \dots, a_n$ are propositional atoms. For a rule r of the form (7), by r^{cl} we denote a clause

$$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_k \vee a_{k+1} \vee \cdots \vee a_m \tag{8}$$

when a_0 is an atom; and a clause

$$\neg a_1 \vee \cdots \vee \neg a_k \vee a_{k+1} \vee \cdots \vee a_m, \tag{9}$$

when a_0 is $\bot$. For a program Π, by Π^{cl} we denote a CNF formula composed of the respective clauses r^{cl} for rules r in Π. For example let Π stand for program

$$\begin{array}{l} p \\ r \leftarrow p, q \end{array} \tag{10}$$

then Π^{cl} follows

$$\begin{array}{ll} p & \wedge \\ r \vee \neg p \vee \neg q. & \end{array} \tag{11}$$

For a program Π, by σ_Π we denote the set of atoms occurring in it. We call σ_Π a program's signature. For a program Π, we call an interpretation M over σ_Π a *classical model* of Π if it is a model of Π^{cl}. For example, program (10) has three classical models $\{p, \neg q, \neg r\}$, $\{p, \neg q, r\}$, and $\{p, q, r\}$. In a sense, a concept of a classical model generalizes the definition of what does it mean for a set of atoms to satisfy a definite program to arbitrary programs.

A set U of atoms occurring in a propositional program Π is *unfounded* on a consistent set M of literals with respect to Π if for every atom $a \in U$ the following condition holds: for every rule in Π of the form

$$a \leftarrow a_1, \ldots, a_k, not\ a_{k+1}, \ldots, not\ a_m$$

(note that a is the head atom in this rule) the property below holds:

- either $M \cap \{\neg a_1, \ldots, \neg a_k, a_{k+1}, \ldots, a_m\} \neq \emptyset$
- or $U \cap \{a_1, \ldots, a_k\} \neq \emptyset$

For instance, set $\{r\}$ is unfounded on set $\{p, \neg q, r\}$ with respect to program (10), while set $\{q\}$ is unfounded on any set of literals with respect to program (10). We may also note that any set of atoms containing atom p will not be unfounded on any set of literals with respect to program (10) (this fact is explained by the presence of fact p. in the program). It is easy to see that the $\emptyset$ of atoms is unfounded on any set of literals with respect to any program.

For a set M of literals, by M^+ we denote the set composed of all the literals that occur without classical negation in M. E.g., $\{p, q, \neg r\}^+ = \{p, q\}$.

We now state a formal result that relates the notions of an unfounded set and answer sets. This result is crucial for understanding key inference rules used in propagators of modern answer set solvers.

Proposition 2. *For a program Π and a set M of literals over σ_Π, M^+ is an answer set of Π if and only if M is a classical model of Π and no non-empty subset of M^+ is an unfounded set on M with respect to Π.*

This proposition gives an alternative characterization of an answer set. I.e., we may bypass the reference to a reduct in our argument that a set of atoms is an answer set. It is sufficient to verify that (i) this set of atoms corresponds to a classical model of a program and (ii) no non-empty subset of this set is unfounded. For example, this proposition asserts that

- classical models of program (10) are the only interpretations that may correspond to answer sets of (10)
- sets $\{p, \neg q, \neg r\}$, $\{p, \neg q, r\}$, $\{p, q, r\}$ of literals are the classical models of program (10). Thus, sets $\{p, \neg q, \neg r\}^+ = \{p\}$, $\{p, \neg q, r\}^+ = \{p, r\}$, $\{p, q, r\}^+ = \{p, q, r\}$ of atoms form the candidates for being answer sets,

- sets $\{p, r\}$ and $\{p, q, r\}$ are not answer sets of the program due to unfounded sets $\{r\}$ and $\{q\}$ respectively. Set $\{p\}$ is an answer set (since the only nonempty subset of it, namely, $\{p\}$, is not an unfounded set on $\{p, \neg q, \neg r\}$ with respect to program (10)).

We define the transition graph $aset_\Pi$ for a program Π as follows. The set of nodes of the graph $aset_\Pi$ consists of the states relative to atoms occurring in Π. There are five transition rules that characterize the edges of $aset_\Pi$. The transition rules *Unit Propagate*, *Decide*, *Fail*, *Backtrack* of the graph $\mathrm{DP}_{\Pi^{cl}}$, and the transition rule

$$\textit{Unfounded}: \quad M \;\Rightarrow\; M\ \neg a \;\text{ if } \begin{cases} a \in U \text{ for a set } U \text{ unfounded on } M \\ \text{with respect to } \Pi. \end{cases}$$

The graph $aset_\Pi$ can be used for deciding whether a logic program has answer sets:

Proposition 3. *For any program* Π*,*

(a) graph $aset_\Pi$ *is finite and acyclic,*

(b) for any terminal state M *of* $aset_\Pi$ *other than FailState,* M^+ *is an answer set of* Π*,*

(c) FailState is reachable from $\emptyset$ *in* $aset_\Pi$ *if and only if* Π *has no answer sets.*

A Peek at Important Enhancements of ASP (and SAT) solvers The key difference of system CLINGO from the SMODELS algorithm that we presented lays in implementation of such advanced solving techniques as *learning and forgetting*, *backjumping* and *restarts*. Below we provide some intuitions behind these.

The *learning* technique allows a solver to extend its knowledge base (that originally is composed of a given program) by additional constraints so that certain inferences become readily available in the later states of search via propagation rules (eliminating the need for intermediate applications of decide rules). The *forgetting* allows the solver to make the learning process dynamic so that sometimes learned constraints are forgotten/removed to eliminate the chance of solver's knowledge base becoming of a prohibitive size.

Backjumping enhances backtracking mechanism by allowing to identify the decision level different from the last one that is safe to jump to so that (i) no solution is lost and (ii) part of the search space is escaped.

Restarting allows a solver to drop currently searched path and start over again with a hope to make better choices on a new path that lead to a solution quicker.

Problem 4. *Let* Π_1 *be a program*

$$\begin{array}{l} r. \\ p \leftarrow \; not\ q, r \\ q \leftarrow \; not\ p, r \end{array}$$

(a) List all classical models of Π_1*.*
(b) List all unfounded sets on set $\{r, p, q\}$ *with respect to program* Π_1*.*
(c) List all unfounded sets on set $\{r, \neg p, \neg q\}$ *with respect to program* Π_1*.*
(d) List all unfounded sets on set $\{r, p, \neg q\}$ *with respect to program* Π_1*.*
(e) List all answer sets of Π_1*.*
(f) List some five states in graph $aset_{\Pi_1}$*.*
(g) List some path in $aset_{\Pi_1}$ *from* $\emptyset$ *to state* $r \neg q^\Delta p$*. Think of another possible path in this graph from* $\emptyset$ *to the same state* $r \neg q^\Delta p$*. List that path. In both cases annotate all the transitions/edges in your path by the names of the respective rules.*

(h) Is state $r\neg q^{\Delta}p$ terminal in the graph $aset_{\Pi_1}$? If so what can you conclude about program Π_1 and state $r\neg q^{\Delta}p$ given Proposition 3.

Acknowledgments

Parts of this handout follow

- the lecture notes on Logic-based AI course, UT, Spring 2011[1] by Vladimir Lifschitz.
- *What is answer set programming to propositional satisfiability*, Yuliya Lierler, Constraints, July 2017, Volume 22, Issue 3, pp 307337 available at `https://link.springer.com/article/10.1007/s10601-016-9257-7`.

In class discussions of Fall 2020 course on *Introduction to AI* at UNO contributed to several examples listed in these notes.

[1] `http://www.cs.utexas.edu/~vl/teaching/lbai`