Local autarkies searching for the dynamic partition of CNF formulae
Éric Grégoire, Bertrand Mazure, Lakhdar Saïs

To cite this version:
Éric Grégoire, Bertrand Mazure, Lakhdar Saïs. Local autarkies searching for the dynamic partition of CNF formulae. 21st International Conference on Tools with Artificial Intelligence (ICTAI’09), 2009, Newark, United States. pp.107-114. hal-00865354

HAL Id: hal-00865354
https://hal.archives-ouvertes.fr/hal-00865354
Submitted on 24 Sep 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Local autarkies searching for the dynamic partition of CNF formulae

Éric Grégoire  Bertrand Mazure   Lakhdar Sais
Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens
{gregoire,mazure,sais}@cril.fr

Abstract

In this paper an original dynamic partition of formulae in Conjunctive Normal Form (CNF) is presented. It is based on the autarky concept first introduced by Monien and Speckenmeyer and further investigated by Kullmann and Van Gelder. Intuitively, an autarky is a partial assignment satisfying some clauses while not affecting any literal in any other clause, leading to a partition of the CNF formula. Autarkies can play a dramatic role in the efficiency of modern SAT solvers. The approach in this paper aims to dynamically extend the current partial assignment to a local autarky thanks to an inference rule based on unit propagation. More precisely, at each node of the search tree, it is checked whether the current decision literal can be made monotone by subsuming all the clauses where it appears negatively. The formal framework is detailed and its technical features discussed.

1. Introduction

The SAT problem, i.e., the problem of checking whether a set of Boolean clauses is satisfiable or not, is central in many computer science and artificial intelligence domains, including e.g. constraint satisfaction problems (CSP), planning, non-monotonic reasoning and VLSI correctness checking. Today, SAT has gained a considerable audience with the advent of a new generation of SAT solvers able to solve large complex instances encoding real-world applications. These solvers also appear to provide crucial base-building blocks for various other problem-solving technologies, like SMT solving, theorem proving, model finding and QBF solving.

These solvers, called modern SAT solvers [19, 6], are based on the standard unit propagation technique [4] combined in a smart way with efficient data structures, together with: (i) restart policies [7, 12], (ii) activity-based variable selection heuristics (VSIDS-like) [3, 19], and (iii) clause learning [17, 1, 19]. Modern SAT solvers can be interpreted as extended versions of the well-known DPLL-like (short for Davis, Putnam, Logemann and Loveland) procedure augmented with these various enhancements. Let us stress that the well-known resolution rule still plays a dramatic role in the efficiency of modern SAT solvers which can be understood as a particular form of general resolution [2]. These lookback-based SAT solvers are particularly efficient on SAT instances encoding practical applications. Other lookahead-based SAT solvers are designed using efficient local processing techniques and variable branching heuristics (e.g. Satz [14], kcnfs [5], March-DL [8]). This last family of solvers are particularly efficient on random and crafted SAT instances.

The recent dramatic breakthrough in solving complex industrial SAT instances can often be related to the presence of hidden structures exploited in a smart way by the SAT solvers. Indeed, such instances are very suitable for decomposition-based techniques. However, most of the decomposition techniques are much time consuming. Light divide-and-conquer approaches have been proposed previously, using for example structure-based variable ordering heuristics to recursively decompose the SAT instances [9]. In [15], a dynamic decomposition method based on hypergraph separators is proposed. The authors give a nice integration of the separator decomposition into variable ordering of a modern SAT solver that speedups the search on real-world SAT instances. Other approaches are based on the exploitation of autarky assignments originally proposed by Monien and Speckenmeyer [18] and further investigated by Kullmann [13] and Van Gelder [21]. More recently, Liffton and Sakallah [16] proposed a novel algorithm that searches for autarkies directly using a standard satisfiability solver and explore the potential of trimming autarkies in minimal unsatisfiable subsets (MUS) and minimal correction subsets (MCS) extraction flows. Let us stress that the dynamic exploitation of autarkies assignments in DPLL-like search procedures actually goes back to Oxusoff’s and Rauzy’s work [20]. In their paper, these latter authors pro-
posed several variable branching heuristics to dynamically partition the CNF formula into disjoint sets of clauses. More precisely, the proposed variable ordering heuristics aim to partition the formula: the next variable to assign is selected in such a way that its assignment leads to a new sub-formula included in previous ones at the same branch of the tree. The approach that will be presented in this paper follows this idea but is heuristic-independent. More precisely, let \( x \) be the next variable to assign, the proposed approach tries to subsume the clauses where \( \neg x \) appears and then deduce that such a literal is monotone\(^1\). This clearly extends the local autarky assignments, using subclause deduction. Indeed, a partial assignment that is not a local autarky can be transformed into a local autarky.

The paper is organized as follows. After some preliminary definitions and notations, basic and local autarkies, Oxusoff’s and Rauzy’s dynamic approach are described in section 3. Then our generalized local autarkies are formally presented in section 4. Finally, we conclude by introducing several promising perspectives opened by this framework.

2. Technical background

2.1. Preliminary definitions and notations

A CNF formula \( \Sigma \) is a set (interpreted conjunctively) of clauses, where a clause is a disjunction of literals. A literal is a positive \( (x) \) or negated \( (\neg x) \) propositional variable. The two literals \( x \) and \( \neg x \) are called complementary. We note by \( I \) the complementary literal of \( l \). For a set of literals \( L, L \) is defined as \( \{ l \mid l \in I \} \). A unit clause is a clause containing only one literal (called unit literal), while a binary clause contains exactly two literals. The empty clause, noted \( \perp \), is interpreted as \( false \) (unsatisfiable), whereas the empty CNF formula, noted \( \top \), is interpreted as \( true \) (satisfiable). We define the size \( |\Sigma| \) of a CNF formula \( \Sigma \) as \( \sum_{c \in \Sigma} |c| \), where \( |c| \) is the number of literals in \( c \). The number of clauses in \( \Sigma \) is denoted \( C_\Sigma \).

The set of variables occurring in \( \Sigma \) is noted \( V_\Sigma \). A set of literals is complete if it contains one literal for each variable in \( V_\Sigma \), and fundamental if it does not contain complementary literals. An assignment \( \rho \) of a Boolean formula \( \Sigma \) is a function that associates a value \( \rho(x) \in \{ false, true \} \) to some of the variables \( x \in V_\Sigma \). \( \rho \) is complete if it assigns a value to every \( x \in V_\Sigma \), and partial otherwise. An assignment is alternatively represented by a fundamental set of literals, in the obvious way. A model of a formula \( \Sigma \) is an assignment \( \rho \) that makes the formula true; noted \( \rho \models \Sigma \).

The following notations will also be used throughout the paper:

- \( \Sigma|_x \) will denote the formula obtained from \( \Sigma \) by assigning \( x \) the truth-value true. Formally \( \Sigma|_x = \{ c \mid c \in \Sigma, \{ x, \neg x \} \cap c = \emptyset \} \cup \{ c \mid c \in \Sigma, \neg x \in c \} \).
- \( \Sigma^* \) denotes the formula \( \Sigma \) closed under unit propagation, defined recursively as follows: (1) \( \Sigma^* = \Sigma \) if \( \Sigma \) does not contain any unit clause, (2) \( \Sigma^* = \perp \) if \( \Sigma \) contains two unit-clauses \( \{ x \} \) and \( \{ \neg x \} \), (3) otherwise, \( \Sigma^* = (\Sigma|_x)^* \) where \( x \) is the literal appearing in a unit clause of \( \Sigma \). A clause \( c \) is deduced by unit propagation from \( \Sigma \), noted \( \Sigma \models c \), if \( (\Sigma|_x)^* \models c \).

Let \( c_1 \) and \( c_2 \) be two clauses of a formula \( \Sigma \). We say that \( c_1 \) (respectively \( c_2 \)) subsumes \( c_2 \) (respectively \( c_1 \)) if \( c_1 \subseteq c_2 \). If \( c_1 \) subsumes \( c_2 \), then \( c_1 \models c_2 \) (the converse is not true). Also \( \Sigma \) and \( \Sigma - c_2 \) are equivalent with respect to satisfiability.

2.2. DPLL search

DPLL [4] is a tree-based backtrack search procedure; at each node of the search tree, the assigned literals (both the decision literal and the propagated ones) are labeled with the same decision level starting from 1 and increased at each decision (or branching). After backtracking, some variables are unassigned, and the current decision level is decreased accordingly. At the \( i \)-th level, the current partial assignment \( \rho \) can be represented as a sequence of decision-propagation steps of the form \((x_1^k, x_2^k, \ldots, x_n^k)\) where the first literal \( x_1^k \) corresponds to the decision literal \( x_k \) assigned at the \( i \)-th level and each \( x_j^k \) for \( 1 \leq j \leq n_k \) represents unit propagated literals at the \( i \)-th level. Let \( x \in \rho \), we note \( \delta(x) \) the assignment level of \( x \).

3. Autarky assignments

The autarky concept has been introduced in [18].

**Definition 1** Let \( \Sigma \) be a CNF formula and \( \rho \) a partial assignment. \( \rho \) is an autarky if \( \Sigma|_\rho \subseteq \Sigma \) i.e. \( \forall c \in \Sigma, \rho \cap c = \emptyset \) or \( \rho \cap c \neq \emptyset \).

For a CNF formula, computing an autarky when it exists is clearly (F)NP-complete. However, autarkies can be used to improve the efficiency of SAT solvers [21, 13]. Indeed, the refutation of \( \Sigma|_\rho \) leads to the proof of the unsatisfiability of \( \Sigma \). Following definition 1, the property prop:inclusion can be obtained.

**Property 1** If \( \rho \) is an autarky of the CNF formula \( \Sigma \) then \( \Sigma \models \Sigma|_\rho \).

---

\(^1\) A monotone literal is a literal that occurs either positively or negatively in the CNF formula (but not in both polarities).
In [10, 20] the authors exploit this property to prune the search tree of DPLL-like procedures. To this end, a local autarky concept is defined and exploited.

**Definition 2** A partial assignment \( \rho = \{l_0, \ldots, l_i\} \) of a CNF formula \( \Sigma \) is a local autarky if \( \exists \rho' = \{l_0, \ldots, l_j\} \) with \( j < i \) such that \( \Sigma|_{\rho'} \subseteq \Sigma|_{\rho} \).

Property 1 also applies for local autarkies. Indeed, from Definition 2 and Property 1, \( \Sigma|_{\rho'} \vdash \Sigma|_{\rho} \) can be deduced. This can be used to prune the search tree of the DPLL algorithm. Indeed, if \( \rho \) is a local autarky with respect to the partial assignment \( \rho' \) then \( \Sigma|_{\rho'} \) is satisfiable if and only if \( \Sigma|_{\rho} \) is satisfiable. All branches not explored between \( \rho' \) and \( \rho \) can be pruned if \( \Sigma|_{\rho} \) is proved unsatisfiable.

The following example illustrates this property.

**Example 1** Let \( \Sigma = \{(l_1 \lor l_2 \lor -l_3), (\neg l_1 \lor l_4), (-l_1 \lor l_2 \lor l_3), (\neg l_2), (l_5 \lor l_6), (\neg l_5 \lor \neg l_6), (\neg l_5 \lor -l_6), (\neg l_5 \lor \neg -l_6)\} \).

The search tree of the DPLL algorithm represented in Figure 1 is obtained thanks to the exploitation of local autarkies. This corresponds to the application of the local autarky pruning rule as defined by Jeannicot et al. in [10]. In the following example, the unit and monotone literals propagated during search are not mentioned for clarity reasons.

From the search tree, the following formulae associated to the different nodes are obtained:

- \( \Sigma|_{\{l_1\}} = \{(l_4), (l_2 \lor l_3), (\neg l_2), (l_5 \lor l_6), (\neg l_5 \lor \neg l_6), (\neg l_5 \lor l_6), (\neg l_5 \lor \neg l_6)\} \)
- \( \Sigma|_{\{l_1, \neg l_2\}} = \{(l_4), (l_1), (l_5 \lor l_6), (\neg l_5 \lor l_6), (\neg l_5 \lor \neg l_6)\} \)
- \( \Sigma|_{\{l_1, -l_2, l_3\}} = \{(l_4), (l_5 \lor l_6), (\neg l_5 \lor \neg l_6), (\neg l_5 \lor l_6), (\neg l_5 \lor \neg l_6)\} \)
- \( \Sigma|_{\{l_1, -l_2, l_4\}} = \{(l_5 \lor \neg l_6), (\neg l_5 \lor l_6), (\neg l_5 \lor \neg l_6)\} \)
- \( \Sigma|_{\{l_1, -l_2, l_5\}} = \{(l_6), (\neg l_5 \lor l_6), (\neg l_5 \lor \neg l_6)\} \)
- \( \Sigma|_{\{l_1, -l_2, l_6\}} = \{(l_6), (\neg l_5 \lor l_6), (\neg l_5 \lor l_5 \lor \neg l_6)\} \)

In Figure 1, the search tree and local autarkies are illustrated. Clearly, the local autarky pruning rule is a generalization of the monotone literal pruning one. Indeed, if \( l \) is a monotone literal, then \( \Sigma|_l \) is the formula obtained from \( \Sigma \) where all clauses containing \( l \) are removed. In consequence \( \Sigma|_l \subseteq \Sigma \) and \( \{l\} \) is a local autarky for \( \Sigma \). Using the pruning rule based on local autarky allows the simplification by means of the monotone literal rule. Let us note that monotone literals are not exploited in modern SAT solvers that exploit lazy data structures like watched literals. In other words, the occurrence number of a given literal at each node of the search tree is not recorded or updated.

At a given level \( i \) of the search tree, finding the smallest level \( j < i \) such that \( \Sigma|_{\{l_0, \ldots, l_j\}} \) is included in \( \Sigma|_{\{l_0, \ldots, l_i\}} \) is computationally hard. However the following property allows an incremental detection that ensures the best pruning.

**Property 2 ([20])** If \( \Sigma|_{\{l_0, \ldots, l_i\}} \subseteq \Sigma|_{\{l_0, \ldots, l_j\}} \) with \( j < i \), then \( \Sigma|_{\{l_0, \ldots, l_j\}} \) is also included in \( \Sigma|_{\{l_0, \ldots, l_{j+1}\}} \), \( \Sigma|_{\{l_0, \ldots, l_{j+2}\}} \), \ldots , and \( \Sigma|_{\{l_0, \ldots, l_{i-1}\}} \).

To avoid useless inclusion tests, it is more convenient to use the following corollary.

**Corollary 1** If \( \Sigma|_{\{l_0, \ldots, l_i\}} \not\subseteq \Sigma|_{\{l_0, \ldots, l_j\}} \), then \( \Sigma|_{\{l_0, \ldots, l_i\}} \) is not included in all the following formulas: \( \Sigma|_{\{l_0, \ldots, l_{j+1}\}} \), \( \Sigma|_{\{l_0, \ldots, l_{j+2}\}} \), \ldots , and \( \Sigma|_{\{l_0, \ldots, l_{i-1}\}} \) and \( \Sigma \).

The efficiency of the detection of the greatest value \( i - j \), called a partition, depends only on the efficiency of the inclusion test between two formulas associated to two consecutive nodes in the current branch of the search tree. This test can be performed efficiently using adapted data structures.

Finally, to ensure that all inclusions are detected, it is necessary to remove subsumed clauses during the DPLL procedure, as illustrated by the following example.

**Example 2** Let \( \Sigma = \{\neg l_1 \lor l_2, l_2 \lor l_4, \neg l_2 \lor l_3 \lor l_4\} \), and \( \Sigma|_{\{l_1\}} = \{(l_3), (l_2 \lor l_4), (\neg l_2 \lor l_3 \lor l_4)\} \), and \( \Sigma|_{\{l_1, l_2\}} = \{(l_3), (l_3 \lor l_4)\} \).
In this case no inclusion is detected. But if the subsumed clauses are removed from $\Sigma_{\{i_1,i_2\}}$, then $\Sigma_{\{i_1,i_2\}} = \{(l_0)\} \subseteq \Sigma_{\{i_1\}}$.

The subsumption test can be achieved efficiently using for example the recent approach by Lintao Zhang [22].

In the next section, a generalization of the (local) autarky assignments is introduced.

### 4. Generalized (local) autarky

The generalization is obtained by substituting the syntactical inclusion operation ($\subset$) by the semantic logical consequence relationship ($\models$) in Definitions 1 and 2.

**Proposition 1 (Generalized (local) autarky)** Let $\Sigma$ be a CNF formula, $\rho = \{l_0, \ldots, l_j, \ldots, l_i\}$ is a generalized local autarky if there exists $\rho' = \{l_0, \ldots, l_j\}$ such that $\Sigma|_{\rho'} \models \Sigma|_{\rho}$. If $\rho' = \emptyset$ then $\rho$ is a generalized autarky.

This is clearly a generalization of the previous concept, because if $\Omega \subset \Sigma$ then $\Sigma \models \Omega$ while the converse is false, as illustrated by the following example.

**Example 3** Let $\Sigma = \{(a \lor b \lor c), (\neg a \lor b)\}$ and $\Omega = \Sigma|_{\neg a} = \{(b \lor c)\}$, clearly $\Sigma \models \Omega$ but $\Omega \not\subset \Sigma$.

The most important feature of local autarkies is the incremental detection which ensures the greatest inclusion or partition (see Property 2). This property is preserved for the generalized autarky assignments.

**Property 3** If $\Sigma|_{\{l_0, \ldots, l_i\}} \models \Sigma|_{\{l_0, \ldots, l_j, \ldots, l_i\}}$, then $\Sigma|_{\{l_0, \ldots, l_j, \ldots, l_i\}}$ is a logical consequence of all the following formulae: $\Sigma|_{\{l_0, \ldots, l_{j+1}\}}, \Sigma|_{\{l_0, \ldots, l_{j+2}\}}, \ldots$ and $\Sigma|_{\{l_0, \ldots, l_{j-1}\}}$.

As before, the following corollary avoids useless logical consequence tests (CoNP-Complete) in order to compute the greatest partition.

**Corollary 2** If $\Sigma|_{\{l_0, \ldots, l_j\}} \not\models \Sigma|_{\{l_0, \ldots, l_j, \ldots, l_i\}}$, then $\Sigma|_{\{l_0, \ldots, l_j, \ldots, l_i\}}$ is not a logical consequence of the following formulae: $\Sigma|_{\{l_0, \ldots, l_{j-1}\}}, \Sigma|_{\{l_0, \ldots, l_{j-2}\}}, \ldots, \Sigma|_{\{l_0\}}$ and $\Sigma$.

We have seen that the propagation of monotone literals is a special case of the local autarky detection. The following property shows that the classical unit propagation is also a special case of the generalized local autarkies.

**Property 4** Let $\Sigma$ be a CNF formula and $l$ a literal occurring in a unit clause of $\Sigma$, we have $\Sigma \models \Sigma|_{\{l\}}$.

By extension, it can be shown that:

**Corollary 3** Let $\Sigma$ a CNF formula, $\Sigma \models \Sigma^*$.

More generally, if $\Sigma \models l$ with $l$ a literal of the CNF formula $\Sigma$, then $\Sigma \models \Sigma|_{\{l\}}$. This means that the detection of generalized local autarkies includes the propagation due to any implied literal.

The use of generalized local autarkies within a DPLL algorithm can be achieved in the same way than for local autarkies. Unfortunately, if the test of inclusion can be computed efficiently for local autarkies, the detection of generalized autarkies requests a CoNP-complete test.

#### 4.1. Generalized (local) autarky modulo unit propagation

In order to practically exploit the generalized local autarkies, it is necessary to weaken the deductive relation that is used. Accordingly, the full consequence relationship ($\models$) is weakened down to the mere unit propagation rule ($\models^*$), which can be computed in linear-time.

**Definition 3** Let $\Sigma$ and $\Omega$ be two sets of clauses, $\Omega$ is a deductive consequence of $\Sigma$ restricted to unit propagation, noted $\Sigma \models^* \Omega$, if and only if $\forall c \in \Omega$, $\Sigma \models^* c$.

Clearly, if $\Sigma \models^* \Omega$ then $\Sigma \models \Omega$. In consequence, Proposition 1 can be adapted to introduce the generalized local autarky modulo unit propagation.

**Proposition 2 (Generalized (local) autarky modulo unit propagation)** Let $\Sigma$ be a CNF formula, $\rho = \{l_0, \ldots, l_j, \ldots, l_i\}$ is a generalized local autarky modulo unit propagation if there exists $\rho' = \{l_0, \ldots, l_j\}$ such that $\Sigma|_{\rho'} \models^* \Sigma|_{\rho}$. If $\rho' = \emptyset$ then $\rho$ is a generalized autarky modulo unit propagation.

Earlier in the paper, it has been shown that the propagations of monotone, unit and more generally implied literals are special cases of local autarky detection. Likewise, the detection of local autarky modulo unit propagation automatically computes the propagations of monotone and unit literals, too. The following property proves useful in that respect:

**Property 5** Let $\Sigma$ and $\Omega$ be two CNF formulae. If $\Omega \subseteq \Sigma$, then $\Sigma \models^* \Omega$.

This property is easily proved. Indeed, each clause $c$ of $\Omega$ is present in $\Sigma$, also $(\Sigma|_{\neg c})^* = \bot$ because the clause $c$ of $\Sigma$ is reduced to an empty clause by the propagation of $\neg c$. In consequence, for each clause $c$ of $\Omega$ $\Sigma \models^* c$ and by definition $\Sigma \models^* \Omega$. The converse is false, as illustrated by the following example.

**Example 4** Let $\Sigma = \{(a \lor b \lor c), (a \lor \neg b), (b \lor \neg c), (c \lor \neg a)\}$ and $\Omega = \Sigma|_{\{b\}} = \{(a), (c \lor \neg a)\}$, we have $\Sigma \models^* \Omega$ but $\Omega \not\subseteq \Sigma$. 
So, Property 5 ensures that all classical local autarkies are also detected by generalized local autarkies modulo unit propagation. In consequence, the simplification of monotone literals is implicitly included during the detection of generalized local autarkies modulo unit propagation. Similarly, the simplification by unit propagation is a special case of generalized local autarkies modulo unit propagation (Property 6).

**Property 6** For any set of clauses \( \Sigma \), \( \Sigma \models^* \Sigma^* \).

Compared with generalized autarkies, only the simplification by the implied literals are not included in the generalized autarkies modulo unit propagation, as illustrated by the following example:

**Example 5** Let \( \Sigma = \{ (a \lor b \lor c), (a \lor b \lor \neg c), (a \lor \neg b \lor c), (\neg a \lor b \lor \neg c), (\neg a \lor \neg b \lor c), (a \lor \neg b \lor c) \} \). We have \( \Sigma \models a \text{ but } \Sigma \not\models^* a \).

However, the restriction has been introduced to ensure that generalized (local) autarkies modulo unit propagation can be computed in polynomial time. Its implementation in a DPLL-like procedure is discussed in the next section.

4.2. Some useful properties and implementation

In this section, some practical properties about the generalized (local) autarkies modulo unit propagation are presented. More precisely, it is shown how the detection of these kinds of autarkies can be computed efficiently during the computation of a DPLL-like algorithm.

The first property is about the time complexity of deduction restricted to unit propagation.

**Property 7** Let \( \Sigma \) and \( \Omega \) be two sets of clauses. Checking \( \Sigma \models^* \Omega \) is in \( \mathcal{O}(C_3 \times (|\Sigma| + |c|)) \) where \( c \) is the longest clause of \( \Omega \).

Unfortunately a polynomial-time complexity does not guarantee practical efficiency. In this respect, we also reduce in a significant way the number of unit propagations that are needed to detect local autarkies modulo unit propagation.

**Property 8** Let \( \Sigma \) a set of clauses and \( l \) a literal. \( \Sigma \models^* \Sigma |_l \) if and only if \( \forall c \in \Sigma \text{ such that } l \in c, \Sigma \models^* c \).

The above property suggests that after the assignment of a literal \( l \), clauses containing \( \neg l \) need to be checked, only.

From properties 7 and 8, the following corollary is easily obtained:

**Corollary 4** Let \( \Sigma \) be a set of clauses and \( l \) a literal. Checking \( \Sigma \models^* \Sigma |_l \) is in \( \mathcal{O}(\text{Occ}(l) \times C_3) \) where \( \text{Occ}(l) \) is the number of occurrences of \( l \) in \( \Sigma \).

Algorithm 1: DPLL-autark

```plaintext
input : A CNF formula \( \Sigma \)
output: true if \( \Sigma \) is satisfiable, false otherwise
1.1 \( \Sigma^* \leftarrow \text{UnitPropagation}(\Sigma); \)
1.2 if \( \Sigma^* = \perp \) then return false;
1.3 else if \( \Sigma^* = \emptyset \) then return true;
1.4 else
1.5 \( l \leftarrow \text{DecisionHeuristic}(\Sigma^*); \)
1.6 if \( \text{DPLL-autark}(\Sigma^* \land (l)) \) then return true;
1.7 else return \( \text{AutarkyModuloUP}(\Sigma^*, l); \)
end
```

This corollary shows that detecting a generalized local autarky modulo unit propagation between two consecutive nodes of a DPLL-like algorithm is a simple task that is very cheap w.r.t. time-complexity. However, there can exist generalized local autarkies modulo unit propagation between two non-consecutive nodes. The following property and its corollary show how useless tests can be avoided and that an incremental detection is possible.

**Property 9** Let \( \Sigma \) be a set of clauses and \( \{l_0, \ldots, l_j, l_i\} \) a set of literals. If \( \Sigma |_{\{l_0, \ldots, l_j\}} \models^* \Sigma |_{\{l_0, \ldots, l_j, l_{j+1}\}} \models^* \Sigma |_{\{l_0, \ldots, l_j, l_{j+2}\}} \models^* \Sigma |_{\{l_0, \ldots, l_j, l_{j+3}\}} \) and \( \Sigma |_{\{l_0, \ldots, l_j, l_{j+1}\}} \models^* \Sigma |_{\{l_0, \ldots, l_j, l_{j+2}\}} \models^* \Sigma |_{\{l_0, \ldots, l_j, l_{j+3}\}} \).

**Corollary 5** Let \( \Sigma \) be a set of clauses and \( \{l_0, \ldots, l_j, l_i\} \) a set of literals. If \( \Sigma |_{\{l_0, \ldots, l_j\}} \not\models^*, \Sigma |_{\{l_0, \ldots, l_j, l_{j+1}\}} \not\models^* \Sigma |_{\{l_0, \ldots, l_j, l_{j+2}\}}, \) and \( \Sigma \not\models^* \Sigma |_{\{l_0, \ldots, l_j, l_{j+3}\}} \).

Detecting generalized local autarkies modulo unit propagation between two non-consecutive nodes of a DPLL-like algorithm depends on such a detection between two consecutive nodes. Accordingly, we insert this detection process within a DPLL-like procedure. It is performed during the backtrack step. Compared with a forward detection, this method avoids the useless unit propagations when the developed branch is proved unsatisfiable. The following algorithms describe this detection process (see Algorithm 2) and its integration within a DPLL-like procedure (see Algorithm 1).

Algorithm 1 details the DPLL-autark function which is a DPLL-like algorithm using the simplification by the generalized (local) autarkies modulo unit propagation. In this algorithm, the UnitPropagation and DecisionHeuristic functions are the DPLL-like solvers usual ones. All kinds of decision heuristics (MOMS [11], VSIDS [19], etc.) can be used within DPLL-autark. When a call to the AutarkyModuloUP function is made, the \( \Sigma |_l \) branch has been already explored and proved unsatisfiable. If \( \Sigma \models^* \Sigma |_l \) then the \( \Sigma |_l \) branch can be pruned since...
an autarky is detected. The AutarkyModuloUP function computes this test and, depending on the result, it either explores or prune the branch. This function is detailed in Algorithm 2.

For efficiency reasons, Algorithm 2 begins with the unit propagation of \( \bar{l} \). Indeed, each clause tested during the verification of \( \Sigma \models^* \Sigma_l \) implies the unit literal \( \bar{l} \). In order to avoid several propagations of \( \bar{l} \), the propagation of \( \bar{l} \) is done once at the beginning of the procedure (see line 2.1).

When a clause \( c' = (l_1 \lor l_2 \lor \cdots \lor l_n) \) is proved to be a logical consequence through unit propagation of \( (\Sigma_l)^* \), it is added to \( (\Sigma_l)^* \) (see line 2.8). These additional clauses speed up the computation of all unit propagations done during the rest of the procedure and reduce the size of the subtree built during the exploration of the branch \( \Sigma_l \) when no autarky has been detected. Note that these additions are performed in constant space. Indeed, the \( c' \) clause is obtained from a \( c \) clause of \( \Sigma \) that contains \( \bar{l} \). This \( c \) clause is satisfied during the unit propagation of \( \bar{l} \) at the beginning of the procedure. The addition of \( c' \) can be done with a simple removal of the \( \bar{l} \) literal from \( c \). This process can be achieved in constant time and space with an adequate data structure.

Finally, even if the generalized local autarky modulo unit propagation is not detected, this approach is able to produce several sub-clauses. It is also possible to stop the detection as soon as \( \Sigma \not\models^* \Sigma|_{l} \) is proved (at the line 2.10). We believe that the spent time to produce sub-clauses is offset by the time saved to explore a smaller sub-tree. Two versions can be derived, one that only checks for generalized local autarkies modulo unit propagation and an other one that additionally produces sub-clauses. In the future, we plan to implement these two versions in order to measure and compare their practical strengths.

4.3. Local autarkies and implication levels

In the previous algorithm, only the current branch of DPLL can be cut when an autarky is found. It could be interesting to cut several branches to process a non-chronological backtracking like it is done in modern implementations of DPLL. In this section, it is shown that this goal can be reached by computing the largest jump with respect to an order of unit propagations. The standard concepts of level for a literal (section 2.2) and implication graph (Definition 5) are used in that respect.

We recall that \( \delta(l) \) the level of a given literal \( l \) is given by the number of branches not explored before its assignment. All assigned literals between two calls of the decision heuristics have the same level, as illustrated in the next example. An unassigned literal has an infinite level. In Table 1, the level of all literals at each node of the search tree depicted in Fig. 2 is detailed.

<table>
<thead>
<tr>
<th>Literal</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l_1 )</td>
<td>0</td>
</tr>
<tr>
<td>( l_2 )</td>
<td>1</td>
</tr>
<tr>
<td>( l_3 )</td>
<td>2</td>
</tr>
<tr>
<td>( l_4 )</td>
<td>3</td>
</tr>
<tr>
<td>( l_5 )</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 2. Levels of literals in DPLL
Definition 4 Let \( \Sigma \) be a CNF formula, \( \{l_1, l_2, \ldots, l_i\} \) a partial assignment of \( \Sigma \) and \( c \) a clause of \( \Sigma \setminus \{l_1, l_2, \ldots, l_i\} \). The level of \( c \), noted \( \delta(c) \), is defined in the following way:

- If \( c \in \Sigma \), \( \delta(c) = 0 \);
- Else \( \delta(c) = \max\{\delta(l) \mid l \in c \text{ et } l \in \{l_1, l_2, \ldots, l_i\}\} \).

The clause level matches the level of the last literal of the clause assigned to false.

The fundamental process to detect a generalized autarky modulo unit propagation consists in checking whether a clause is implied by unit propagation. The sequence of generated implications can be depicted by an acyclic oriented graph, called implication graph. Usually such a graph is used in all CDCL (conflict-driven clauses learning) approaches to learn clauses. It was first introduced in [17].

Definition 5 An implication graph, noted \( \mathcal{G} = (\mathcal{V}, \mathcal{A}) \) is a graph such that:

1. \( \mathcal{V} \) is the set of vertices where each vertex, noted \( s(l) \), is an assigned literal during unit propagation;
2. \( \mathcal{A} \) is the set of edges. For a vertex \( s(l) \) the set of predecessors is defined by the vertices \( \{s(l_1), \ldots, s(l_{i-1}), s(l_{i+1}), \ldots, s(l_n)\} \) if there exists a clause \( c = (l_1, \ldots, l_{i-1}, l_i, l_{i+1}, \ldots, l_n) \) and a current partial assignment \( \rho \) such that \( \{l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n\} \subseteq \rho \). Each arc \( \langle s(l), s(l_i) \rangle \) is labeled by \( c \).

The roots of the implication graph representing the implication of a given clause by unit propagation are the vertices labelled by the negation of each literal of the clause. By construction, it is straightforward that this graph is acyclic.

Example 6 Let \( \Sigma \) be the CNF formula built on the following 14 clauses:

- \( c_1 : (\neg a \lor c) \)
- \( c_2 : (\neg a \lor d) \)
- \( c_3 : (\neg b \lor d) \)
- \( c_4 : (\neg d \lor f) \)
- \( c_5 : (\neg d \lor g) \)
- \( c_6 : (\neg b \lor e) \)
- \( c_7 : (\neg c \lor i) \)
- \( c_8 : (\neg c \lor h) \)
- \( c_9 : (\neg a \lor f) \)
- \( c_{10} : (\neg h \lor k) \)
- \( c_{11} : (\neg i \lor l \lor \neg k) \)
- \( c_{12} : (\neg i \lor \neg l) \)
- \( c_{13} : (\neg k \lor \neg l) \)
- \( c_{14} : (\neg c \lor e) \)

The implication graph corresponding to the implication by unit propagation of the clause \( (a \lor b) \) is depicted in Fig. 3.

The goal is to compute the minimum levels of implication of the clauses that are used during the detection of the generalized autarkies modulo unit propagation. A level can be assigned to the detected autarky. It is defined as the greatest implication level computed for each implied clause. In consequence, if the level of the autarky is \( j \), it is possible to backjump safely up to this decision level. Indeed, for a sub-formula \( \Sigma_j \) obtained from \( \Sigma \) and a partial assignment \( \{l_1, \ldots, l_j, \ldots, l_i\} \), if all the clauses of \( \Sigma_j \) are implied modulo unit propagation at level \( k \leq j \), then \( \Sigma_j =^* \Sigma_i \). All branches between \( j \) and the current level \( i \) are pruned.

Furthermore, even when no autarky is found, we have seen that the clauses that are proved to be implied during the autarky detection, are added to the formula. These clauses are added only until the current decision level is achieved. Since its minimal implication level has been computed, it is possible to keep the implied clauses until backtracking to their corresponding levels. This will reduce the size of search tree even more.

5. Conclusion

This paper presents a formal framework for the detection and use of autarky assignments, with the goal to im-
prove the efficiency of SAT solvers. Several contributions about the generalization of classical autarkies have been provided. The first one introduces the concept of autarky modulo unit propagation, which extends the usual definition by substituting the syntactical inclusion operation by a semantic-one based on unit propagation. It has also been proved that propagating unit literals are special cases of this generalization. A second extension introduces the concept of implication level, aiming at detecting larger partitions. Interestingly enough, when no generalized local autarky can be found, the approach is able to produce several sub-clauses from the formula. Each implied clause can be exploited in order to reduce the search space until backtracking to its corresponding implication level. Finally, it is shown how these different approaches could be integrated in DPLL-like SAT solvers.

As future works, we plan to investigate how to integrate and implement the approach in modern SAT solvers. This might prove very successful for efficiently solving industrial-related SAT instances. It is well-known that many of these instances can be partitioned into several disconnected components after assigning a small set of variables. Another interesting path of research would be to design powerful clauses ordering heuristics for searching for local autarkies modulo unit propagation.

References