DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 0000

# Efficiency and Stability of Hypergraph SAT Algorithms

DANIELE PRETOLANI

ABSTRACT. We discuss some topics related to the practical efficiency of exact methods for satisfiability. We use directed hypergraphs as an algorithmic and modeling tool; in particular, we propose a new relaxation technique, based on a hypergraph depth first search procedure. We discuss our computational experience, comparing the suitability of different approaches for various classes of instances.

## 1. Introduction

In the last years, a growing interest on efficient methods for solving hard satisfiability problems led to the design of many different algorithms, as well as to the implementation of faster and faster codes. There seems to be empirical evidence [4] that the most efficient SAT solvers are specialized versions of a simple branch and bound schema, usually referred to as the *Davis-Putnam-Loveland* algorithm (*DPL*). The efficiency of these methods relies on carefully designed data structures, as well as on simple and effective branching rules. These two aspects are related, since the data structures representing the formula are often chosen according to the implementation of the branching rule.

Specialized DPL algorithms proved to be efficient for randomly generated problems; on the other hand, they show a pathological behaviour on some more structured instances [10, 20]. It is not clear if these algorithms can be optimal for a wider range of problems, e.g. those arising from boolean functions analysis or circuit theory. Many attempts have been made to improve the performance of DPL algorithms, either by further refining the branching rules, or by adding other components, such as pruning methods or local search. Even though some

---

1991 *Mathematics Subject Classification.* Primary 90C09, 90C04; Secondary 05C65.

This work has been supported by a grant from the Government of Québec, Ministère de l'Enseignement Superieur et de la Science.

This paper is in final form and no version of it will be submitted for publication elsewhere.


0000-0000/00 $1.00 + $.25 per page

of these attempts were successful in some cases, they do not increase efficiency for all classes of instances.

Therefore, we are faced with a *stability* issue. In practice, the efficiency of an algorithmic method may be unpredictably affected by the characteristics of the input, resulting in disappointingly poor performances, or unexpected good ones. The analysis of these phenomena may be relevant for two reasons: it can shed light on the properties of the algorithms (see [12] for a discussion of the theoretical relevance of empirical results) and it can provide some guidelines for the development of further algorithms.

In this paper, we propose some methods to improve the efficiency of SAT algorithms, and we make a first attempt to investigate their stability. We use *directed hypergraphs* [9] as a model for the design and implementation of our algorithms; in particular, we propose a new pruning method, based on a *hypergraph depth first search* technique. We compare our hypergraph implementations of DPL to the *B-reduction method* [10, 20], which exploits some basic properties of directed hypergraphs, and uses a quite different enumeration strategy.

The plan of the paper is as follows. In section 2, we introduce the basic terminology, while in section 3 we describe our hypergraph algorithms. Computational results are discussed in section 4, and final remarks are reported in section 5. We assume that the reader is familiar with the basic definitions related to propositional formulas in *Conjunctive Normal Form* (*CNF*) and to the satisfiability problem.

## 2. Satisfiability and Hypergraphs

In this section, we introduce our hypergraph language for satisfiability. We restrict ourselves to the basic definitions used in the paper; see [20] for a more detailed survey of the relations between hypergraphs and satisfiability. A wider introduction to directed hypergraphs can be found in [9].

A *directed hypergraph* (or simply a hypergraph) is a pair $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of nodes, and $\mathcal{E}$ is the set of *hyperarcs*. A hyperarc $a \in \mathcal{E}$ is a pair $(T(a), H(a))$, where $T(a)$ and $H(a)$, respectively the *tail* and the *head* of $a$, are disjoint subsets of $\mathcal{V}$.

We write $u \in a$ if node $u$ belongs to the tail or to the head of $a$. We denote by $BS(v) = \{a \in \mathcal{E} \mid v \in H(a)\}$ and $FS(v) = \{a \in \mathcal{E} \mid v \in T(a)\}$ respectively the *backward star* and the *forward star* of node $v$.

Let $n$ and $m$ be the number of nodes and hyperarcs in a hypergraph $\mathcal{H}$. Denote by $|a| = |T(a)| + |H(a)|$ the *size* of hyperarc $a$; if $|a| = 2$ then $a$ is *binary*. We define the *size* of a hypergraph $\mathcal{H}$ as follows:

$$size(\mathcal{H}) = \sum_{a \in \mathcal{E}} |a|.$$

A hyperarc $a$ is a *B-arc* if $|H(a)| \leq 1$; a *B-graph* is a hypergraph whose hyperarcs are B-arcs. Given a hyperarc $a$, a *B-reduction* $a_B$ of $a$ is a B-arc

obtained by deleting all the nodes in $H(a)$ except one. A *B-reduction* $\mathcal{H}_B$ of a hypergraph $\mathcal{H}$ is the B-graph obtained by replacing each hyperarc in $\mathcal{H}$ by one of its B-reductions. Observe that the number of B-reductions of $\mathcal{H}$ can be exponential in $size(\mathcal{H})$.

A *path* $P_{st}$ in a hypergraph $\mathcal{H}$ is a sequence of nodes and hyperarcs in $\mathcal{H}$:

$$P_{st} = (v_1 = s, e_1, v_2, e_2, \ldots, e_q, v_{q+1} = t)$$

where $v_i \in T(e_i)$ and $v_{i+1} \in H(e_i)$, $i = 1, \ldots, q$. We say that $t$ is *connected* to $s$ in $\mathcal{H}$ if there exists in $\mathcal{H}$ a path $P_{st}$. A path where $s = t$ is a *cycle*; a hypergraph is *acyclic* if it does not contain cycles.

Consider a B-graph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$; a *B-path* $\Pi_{st}$ in $\mathcal{H}$ is a minimal acyclic B-graph $\mathcal{H}_\Pi = (\mathcal{V}_\Pi, \mathcal{E}_\Pi)$ such that:

- $s, t \in \mathcal{V}_\Pi \subseteq \mathcal{V}$;
- $\mathcal{E}_\Pi \subseteq \mathcal{E}$;
- $u \in \mathcal{V}_\Pi$, $u \neq s \implies u$ is connected to $s$ in $\mathcal{H}_\Pi$.

Here, minimality is intended with respect to deletion of hyperarcs and nodes. We say that $t$ is *B-connected* to $s$ in $\mathcal{H}$ if there exists in $\mathcal{H}$ a B-path $\Pi_{st}$.

Hypergraphs provide a natural way of representing CNF formulas. We associate a node with each variable; for each clause $C$, we define a hyperarc $a_C$ whose tail and head respectively contain the nodes corresponding to negated and non-negated variables in $C$. As an example, a clause:

$$C = u_1 \vee \neg u_2 \vee u_3 \vee \neg u_4$$

is represented by the hyperarc:

$$a_C = (\{u_2, u_4\}, \{u_1, u_3\}).$$

More formally, given a CNF formula $\sigma = (\mathcal{P}, \mathcal{M})$, where $\mathcal{P}$ and $\mathcal{M}$ are the sets of variables and clauses respectively, we define the hypergraph $\mathcal{H}_\sigma = (\mathcal{V}_\sigma, \mathcal{E}_\sigma)$ *associated with* $\sigma$ as follows:

$$\begin{aligned} \mathcal{V}_\sigma &= \mathcal{P}; \\ \mathcal{E}_\sigma &= \{a_C : C \in \mathcal{M}\}. \end{aligned}$$

Observe that $size(\mathcal{H}_\sigma)$ is equal to the *length* of the formula $\sigma$, that is, the total number of occurrences of variables. From now on, we will identify variables and nodes; for example, we will say that a node is assigned a value *true* or *false* if the corresponding variable is.

Denote by $\mathcal{H}(\sigma)$ the hypergraph obtained from $\mathcal{H}_\sigma$ by adding two nodes $F$ and $T$, and replacing each empty head (tail) by the set $\{F\}$ (respectively $\{T\}$). Clearly, $size(\mathcal{H}(\sigma))$ is linear in the length of $\sigma$. Observe that $\mathcal{H}(\sigma)$ and $\mathcal{H}_\sigma$ are B-graphs when $\sigma$ is a *Horn formula*, in which each clause contains at most one non-negated variable.

The following proposition relates B-paths in B-graphs and satisfiability of Horn formulas:

PROPOSITION 1. [9] *A Horn formula $\sigma$ is satisfiable if and only if $\mathcal{H}(\sigma)$ does not contain a B-path $\Pi_{TF}$.*

Let us say that a B-reduction of $\mathcal{H}(\sigma)$ is *feasible* if it does not contain a B-path $\Pi_{TF}$; the next proposition relates B-reductions and satisfiability:

PROPOSITION 2. [9] *A formula $\sigma$ is satisfiable if and only if there exists a feasible B-reduction of $\mathcal{H}(\sigma)$.*

In practice, Proposition 2 means that a formula $\sigma$ is satisfiable if and only if there exists a satisfiable Horn formula obtained from $\sigma$ by deleting some ocurrences of non-negated variables.

## 3. Hypergraph SAT Algorithms

We will derive our hypergraph SAT algorithms from a simplified version of DPL [18]. In practice, we consider a general enumeration schema, where *Unit Resolution* is used to simplify a sub-problem $P$, and a binary branching technique is adopted: two sub-problems $P_T$ and $P_F$ are generated from $P$ by setting a *branching variable* $p$ to *true* and *false*, respectively. Two further steps can be added to the above schema. First, one may try to prove that $P$ is not satisfiable; this step usually corresponds to solving a *relaxation* of $P$, i.e. a problem $P_r$ which is guaranteed to be satisfiable when $P$ is. In addition, one may try to prove that $P$ is satisfiable, and this can be done by solving a *restriction*, i.e. a problem $P_R$ which is satisfiable only if $P$ is. A formal description of our enumeration schema follows:

*Algorithm* DPL
*Input*: a CNF formula $\sigma$;
*Output*: "yes" if $\sigma$ is satisfiable; "no" otherwise;
*Step* 0: set $Q = \{\sigma\}$;
*Step* 1: if $Q = \emptyset$ return "no"; otherwise, remove $P$ from $Q$;
*Step* 2: simplify $P$ by Unit Resolution; if $P$ becomes empty, return "yes"; if a contradiction arises, go to step 1;
*Step* 3: solve a relaxation $P_r$ of $P$; if $P_r$ is not satisfiable, go to step 1;
*Step* 4: solve a restriction $P_R$ of $P$; if $P_R$ is satisfiable, return "yes";
*Step* 5: select a branching variable $p$; set $Q = Q \cup \{P_T, P_F\}$; go to step 1.

We assume that the sub-problems are examined in *Last In - First Out* order, i.e. DPL generates a binary *enumeration tree* in a depth first fashion. Therefore, $Q$ is implemented as a *stack*; when a contradiction arises, in step 2 or 3, DPL *backtracks* to the unsolved sub-problem on top of $Q$. A *branching rule* specifies the criteria used to select the branching variable and the sub-problem $P_T$ or $P_F$ to be solved first.

The input formula $\sigma$ and each sub-problem will be represented by means of hypergraphs; this allows for a quite efficient implementation of the basic steps, i.e. Unit Resolution and backtracking. A linear time hypergraph implementation of Unit Resolution was given in [10]. In fact, in our implementation, Unit Resolution and backtracking have a linear *branch complexity*, that is, the overall time they spend in all the problems along a branch in the enumeration tree is linear in $size(\mathcal{H}_\sigma)$ [20].

We will discuss branching rules and relaxations in the following; although not used in our algorithms, restrictions will be briefly described to introduce the B-reduction method.

**3.1. Branching Rules.** Many different branching rules for DPL have been proposed in the literature, see e.g. [4, 7, 16]. However, the relations between the analytical and the empirical properties of these rules are not yet clear, even if some attempts in this direction have been made [14]. In the following, we devise two different branching rules by combining various known techniques.

A measure of the impact on the problem of selecting a variable $p$ for branching can be obtained as a function of the clauses in which $p$ appears. A quite simple approach consists in considering only the clauses of minimum length: intuitively, these clauses can be considered as more crucial, in particular if they are binary (recall that the formula does not contain unit clauses when a branching is performed).

Let $\mathcal{H}$ be the hypergraph associated with the current sub-problem. Denote by $FS_k(u)$ and $BS_k(u)$ the sets of hyperarcs of size $k$ in $FS(u)$ and $BS(u)$, respectively. For each $k \geq 2$ define the node weighting function $W_k$ as follows:

$$W_k(u) = |FS_k(u)| + |BS_k(u)| + \alpha \min\{|FS_k(u)|, |BS_k(u)|\}.$$

Let $k$ be the minimum size of hyperarcs in $\mathcal{H}$; we select for branching the node with the larger weight $W_k$. Here, the parameter $\alpha = 1.5$ is introduced to give higher priority to a node $u$ if the difference between $|FS_k(u)|$ and $|BS_k(u)|$ is small. This is done in order to obtain two subroblems $P_T$ and $P_F$ with similar size, and therefore generate a more balanced enumeration tree. A similar technique has been used e.g. by Böhm and Speckenmeyer [3].

If $k = 2$, we solve first the sub-problem in which more unit clauses appear. Otherwise, we choose the one in which more clauses are deleted. We summarize the above criteria in the following branching rule:

*Branching Rule* 1

(i) $k := \min\{|a| : a \in \mathcal{H}\}$; $p := \arg\max\{W_k(u) : u \in \mathcal{H}\}$;

(ii) $k > 2$: if $|BS_k(p)| > |FS_k(p)|$ then solve $P_T$ first, else solve $P_F$ first;
$k = 2$: if $|BS_k(p)| > |FS_k(p)|$ then solve $P_F$ first, else solve $P_T$ first.

In practice, Rule 1 requires a small amount of computation, even if in the worst case it takes $O(size(\mathcal{H}_\sigma))$ time for each node in the enumeration tree. It is worth noting that the sets $FS_2(u)$ and $BS_2(u)$ can be maintained for each node

$u$ with an overall $O(size(\mathcal{H}_\sigma))$ branch complexity; thus, computation becomes much faster as soon as binary hyperarcs appear.

We devise a more sophisticated rule using the following techniques. First, we introduce two *static* node weights $S_F(u)$ and $S_B(u)$, which give a measure of the overall relevance of a node. These weights are computed at the beginning of the algorithm, considering all the non-binary hyperarcs in the original problem:

$$S_B(u) = \frac{\beta}{\mu} \sum_{a \in BS(u),\, |a|>2} 2^{-|a|}; \qquad S_F(u) = \frac{\beta}{\mu} \sum_{a \in FS(u),\, |a|>2} 2^{-|a|};$$

where:

$$\mu = \frac{1}{n} \sum_{a \in \mathcal{A}_\sigma,\, |a|>2} |a|\, 2^{-|a|}.$$

These weights are normalized to an expected value $S_B(u) + S_F(u) = \beta$. The parameter $\beta$ has been set to 6 in our computational experience.

In addition, we adapt a method proposed by Dubois et al. [7], based on the following observation: in order to obtain a better evaluation of the relevance of a node $u$, one could take into account all the nodes (if any) that are assigned a value, by Unit Resolution, as a consequence of setting $u$. In practice, we partially exploit this idea, that is, we consider only those nodes appearing in binary hyperarcs together with $u$. Therefore, we introduce the following node weights, for each $k \geq 2$:

$$F_k(u) = f_k(u) + \sum_{(\{u\},\{v\}) \in FS(u)} f_2(v) + \sum_{(\{u,v\},\emptyset) \in FS(u)} b_2(v);$$

$$B_k(u) = b_k(u) + \sum_{(\emptyset,\{u,v\}) \in BS(u)} f_2(v) + \sum_{(\{v\},\{u\}) \in BS(u)} b_2(v);$$

where $f_k(u) = |FS_k(u)| + S_F(u)$ and $b_k(u) = |BS_k(u)| + S_B(u)$. Finally, we define a new weighting function:

$$W'_k(u) = F_k(u) + B_k(u) + \alpha \min\{F_k(u), B_k(u)\}.$$

Our improved branching rule is as follows:

*Branching Rule* 2

(i) $k := \min\{|a| : a \in \mathcal{H}\}$; $p := \arg\max\{W'_k(u) : u \in \mathcal{H}\}$;

(ii) $k > 2$: if $B_k(p) > F_k(p)$ then solve $P_T$ first, else solve $P_F$ first;
$k = 2$: if $B_k(p) > F_k(p)$ then solve $P_F$ first, else solve $P_T$ first.

Observe that, in each node of the enumeration tree where binary clauses appear, Rule 2 requires an extra $O(m)$ computation with respect to Rule 1.

**3.2. Relaxations.** The idea of solving relaxations to improve DPL algorithms has been exploited in different ways. In general, *logical inference* methods are used in order to derive a contradiction or assign a value to some variables.

Some algorithms use powerful inference methods, based e.g. on resolution [2] or cutting plane generation [13]. In practice, these methods turn out to be too expensive – even though they lead to a significant reduction in the size of the enumeration tree.

Other relaxations are based on a simple *guess and deduce* technique. Unit Resolution is used to derive the consequences of assigning a value to a certain variable. If a contradiction arises, clearly the variable must be set to the opposite value. In principle, this method is simple and fast, but requires to be carefully implemented, in order to limit the time spent in performing Unit Resolution (see e.g. [7, 8]).

A more sophisticated technique is based on the solution of a *binary relaxation*, i.e. a 2-SAT instance $\sigma_2$ given by the binary clauses in the current sub-problem. It is well known that 2-SAT can be solved in linear time; moreover, all the logical conclusions implied by a 2-SAT instance (such as assignments of values to variables, or equivalence of variables) can be found in linear expected time [11]. Binary relaxations have been exploited in several different ways (see e.g. [4, 15]). In general, these techniques have a main drawback, i.e. the number of binary clauses appearing in the sub-problems may be too small to provide relevant information.

A hybrid of the above two approaches has been used in the *Extended 2-SAT Relaxation* [20], where $\sigma_2$ is solved using a guess and deduce technique, but applying Unit Resolution to the whole formula.

Here, we propose a different guess and deduce relaxation, where Unit Resolution is replaced by a more sophisticated inference method. This method is based on *depth first search* in hypergraph [20], and uses a *hyperarc shrinking* technique, previously applied to B-graphs to obtain a linear time algorithm for the *Unique Horn Satisfiability* problem [19].

Procedure Deduce$(u, l)$, given below, finds the consequences of assigning the value $l \in \{true, false\}$ to node $u$. Boolean values are represented by a node label $L$, initially set to *null* for each node. We say that $u$ is *visited* when Deduce$(u, l)$ begins, i.e. $L(u)$ is set to $l$, and remains *active* until Deduce$(u, l)$ ends.

Given a hyperarc $a$ and a node $u \in a$, let $tvalue(u, a) = true$ if $u \in H(a)$, and $tvalue(u, a) = false$ if $u \in T(a)$. For each visited node $u$, the set $AD(u)$ contains each hyperarc $a$ such that $u \in a$ and $tvalue(u, a) = \neg L(u)$. Binary hyperarcs in the sets $AD$ are used to set node labels. If a *conflict* arises, i.e. if Deduce tries to assign a label $l$ to a node $v$ such that $L(v) = \neg l$, then the procedure ends returning $v$. Binary hyperarcs are also used to build (in a depth first fashion) a tree $\mathcal{T}$, defined by the predecessor function $P$.

For each non-binary hyperarc $a$ we introduce a counter $V(a)$, initially set to $|a|$, which is decreased each time a node $u$ such that $a \in AD(u)$ is visited.

If $V(a) = 0$, Deduce ends returning $root(a)$, that is, the common ancestor in $\mathcal{T}$ of the visited nodes in $a$. If $V(a) = 1$, and there is a not yet visited node $u \in a$, a binary *shrunken hyperarc* $a_S$ is added to $AD(w)$, where $w = root(a)$; $a_S$ contains the nodes $w$ and $u$, with $tvalue(w, a_S) = \neg L(w)$ and $tvalue(u, a_S) = tvalue(u, a)$.

In the formal description of Deduce, function $last(a)$ returns a not yet visited node in $a$, if one exists, and *null* otherwise; function $shrink(a)$ returns the shrunken hyperarc $a_S$ described above.

**procedure** $Deduce(u, l)$
  $L(u) := l$;
  **if** $l = true$
  **then** $AD(u) := FS(u)$;
  **else** $AD(u) := BS(u)$;
  **for_each** $\{a \in AD(u) : \mid a \mid > 2\}$ **do**
    $V(a) := V(a) - 1$;
    **if** $V(a) = 1$ **and** $last(a) \neq null$
    **then** $w := root(a)$; $AD(w) := AD(w) \cup shrink(a)$;
    **else** **if** $V(a) = 0$ **then return**$(root(a))$;
  **end_for_each**;
  **for_each** $\{a \in AD(u) : \mid a \mid = 2\}$ **do**
    **select** $v \in a, v \neq u$;
    **if** $L(v) = null$
    **then** $P(v) := u$; $Deduce(v, tvalue(v, a))$;
    **else** **if** $L(v) \neq tvalue(v, a)$ **then return**$(v)$;
  **end_for_each**;
**end_procedure**;

In the following, we point out the basic properties of procedure Deduce.

LEMMA 1. *The addition of shrunken hyperarcs is correct.*

PROOF. Let $a_S = shrink(a)$ be the first shrunken hyperarc added, containing nodes $w = root(a)$ and $u = last(a)$. It is easy to see that if $w$ is set to $L(w)$ then each node $v \in a$, $v \neq u$, must be set to $L(v) \neq tvalue(v, a)$; as a consequence, node $u$ must be set to $tvalue(u, a)$. Therefore, the clause corresponding to $a_S$ is implied by the current formula. Then, the lemma can be easily proved by induction. □

LEMMA 2. *When a hyperarc $a_S = shrink(a)$ is selected from $AD(w)$ a label conflict cannot arise.*

PROOF. Let $w \neq u \in a_S$; the property is obvious if $u$ has not yet been visited when $a_S$ is selected. If $u$ has been visited, then $a \notin AD(u)$, otherwise it would be $V(a) = 0$, and the procedure would have terminated returning $root(a)$. Therefore, $L(u) = tvalue(u, a_S)$, and no conflict can arise. □

THEOREM 1. *If Deduce returns a node $v$ then $v$ and its ancestors in $\mathcal{T}$ must be set to the value opposite to their labels.*

PROOF. First, observe that shrunken hyperarcs can be used to generate $\mathcal{T}$; this is correct, as follows from Lemma 1. Suppose that a conflict arises while considering a binary hyperarc $a \in AD(u)$, containing node $v$. By Lemma 2, $a$ cannot be a shrunken hyperarc, thus $a \in AD(v)$ and node $v$ must be active when the hyperarc $a$ is considered. It follows that $v$ is an ancestor of $u$ in the tree $\mathcal{T}$, therefore, setting $v$ to $L(v)$ leads to a contradiction. Suppose now that for a non-binary hyperarc $a$ it is $V(a) = 0$ and $v = root(a)$: again, setting $v$ to $L(v)$ leads to a contradiction. The theorem follows immediately from the above observations. □

Theorem 1 implies that Deduce is an implementation of Unit Resolution. In practice, within a guess and deduce approach procedure Deduce should be preferred, since it might allow to set the value of several nodes after each guess. Moreover, we can take advantage of the following property, which might allow to assign values even if Deduce does not return a node:

PROPERTY 1. *If a node $v$ is assigned the same label $L(v)$ by Deduce$(u, false)$ and Deduce$(u, true)$ then $v$ must be set to $L(v)$.*

In addition, the following property allows us to skip some guesses, avoiding unnecessary calls to Deduce:

PROPERTY 2. *If Deduce$(u, l)$ does not return a node then, for each node $v$ in the tree $\mathcal{T}$, Deduce$(v, L(v))$ does not return a node.*

Our relaxation is formally described by algorithm Relax, which takes as input the hypergraph representing the current sub-problem, and returns "backtrack" if a contradiction is found by Unit Resolution, which is applied each time Deduce returns a node.

*Algorithm* Relax
*Input*: a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$;
*Output*: "backtrack" if a contradiction is detected;
*Step* 0 $\forall u \in \mathcal{V}$ set $B(u) := \{true, false\}$; $S := \mathcal{V}$;
*Step* 1 if $S = \emptyset$ then stop; else, remove a node $u$ from $S$;
*Step* 2 for each $l \in B(u)$ perform the following step:
if Deduce$(u, l)$ returns a node, set values according to Theorem 1, and go to step 4; otherwise, set $B(v) := B(v) \setminus L(v)$ for each visited node $v$;
*Step* 3 if Deduce$(u, l)$ was performed for each $l \in \{true, false\}$, set values according to Property 1, and go to step 4; otherwise, go to step 1;
*Step* 4 apply Unit Resolution; if a contradiction arises, return "backtrack"; otherwise, delete from $S$ the nodes set to a value, and go to step 1.

Using proper data structures to implement function *root*, procedure Deduce takes an overall $O(size(\mathcal{H}_\sigma))$ time for each built tree (see [19]); this gives an $O(n\ size(\mathcal{H}_\sigma))$ complexity for algorithm Relax. In the actual implementation of Deduce, we use a simpler method, which is faster in practice even though its worst case bound is $O(n\,m\ +\ size(\mathcal{H}_\sigma))$.

**3.3. Restrictions.** Restrictions have been included in SAT algorithms in different ways. In *branch and cut* methods the solution of the linear relaxation is actually used in a restriction step (see e.g. [13]). A heuristic procedure for searching a solution is proposed in [16]. A different approach is followed in [15], where *tabu search* is used before starting enumeration.

The *B-reduction method* reduces SAT to the search of a feasible B-reduction, as suggested by Proposition 2. Its basic restriction step consists in the generation of a B-reduction $\mathcal{H}_B$ of the hypergraph $\mathcal{H}(\sigma)$; this step is implemented as a particular visit procedure, which checks the feasibility of $\mathcal{H}_B$ while building it. If $\mathcal{H}_B$ is not feasible, a B-path $\Pi_{TF}$ is selected among those contained in $\mathcal{H}_B$ (see Proposition 1); the (unique) hyperarc $a = (\{u_1, u_2, \ldots, u_k\}, \{F\})$ incident with $F$ in $\Pi_{TF}$ is used in the branching step. The following *multiple branching* technique is adopted: $k$ sub-problems $P_1, \ldots, P_k$ are generated, in sub-problem $P_i$ one has $u_i = false$ and $u_j = true$, $\forall j < i$.

In our computational experience, we used a version of the B-reduction method which includes the Extended 2-SAT Relaxation. The resulting algorithm, called BRR, shows a good efficiency, as well as a rather stable behaviour on many classes of problems [20]. However, it is slower than specialized DPL versions on random problems [4].

**3.4. Implementation Details.** We implemented four hypergraph versions of our enumeration schema. Algorithms H1 and H2 correspond to basic DPL, not including any relaxation or restriction step; they use branching rules 1 and 2, respectively. Algorithms H1R and H2R are the versions of H1 and H2 that include algorithm Relax.

In the actual implementation of the relaxation step we introduced some limitations. Indeed, Relax is applied only at a sufficient depth in the enumeration tree, and guesses are made only for nodes appearing in sufficiently many binary hyperarcs. Moreover, only hyperarcs of size 2 or 3 are considered by Deduce.

It is worth noting that the same data structures are used for all the versions. Hypergraphs are represented by means of backward and forward stars; $FS(u)$ and $BS(u)$ are implemented as linked lists, whose items represent occurrences of node $u$ in tails and heads of hyperarcs. A hyperarc is represented as a set of the above list items. When comparing the results of our DPL algorithms to the ones of algorithm BRR, it should be remarked that BRR uses a slightly different representation of hypergraphs (in fact, a less efficient one).

## 4. Computational Results

In this section we discuss the results of our computational experience. We solved several randomly generated problems, as well as some instances with a special structure, obtained by encoding combinatorial problems in terms of satisfiability. For each type of problems, we report the average CPU time and the average number of enumerated nodes; in some cases (i.e. for reasonably large sample sets) we also report the estimated *standard deviation* of the observed results. Times are given in seconds on a SUN SPARCstation 2.

First we consider $k$-SAT problems, with $k \in \{3,5\}$; we chose a ratio $m/n$ close to the *crossover point* [6], where satisfiable and unsatisfiable instances are expected to be sufficiently hard and to occur with similar frequency. In table 1 and 2 we report the results for problems of smaller size; standard deviations are given within parentheses.

| $n = 200$, $m = 860$ | | H1 | H2 | H1R | H2R | BRR |
|---|---|---|---|---|---|---|
| *time* | (10 Yes) | 9.42 | 6.55 | 8.71 | 5.80 | 44.26 |
| | | (7.06) | (6.74) | (8.99) | (6.19) | (45.05) |
| | (10 No) | 28.72 | 16.54 | 23.21 | 13.40 | 132.62 |
| | | (8.35) | (4.00) | (6.85) | (3.30) | (55.21) |
| *nodes* | (10 Yes) | 10,345 | 5,121 | 1,791 | 1,016 | 6,119 |
| | | (7,985) | (5,430) | (1,888) | (1,112) | (6,209) |
| | (10 No) | 31,297 | 13,054 | 5,129 | 2,544 | 18,452 |
| | | (10,181) | (3,213) | (1,461) | (604) | (7,453) |

TABLE 1. 3-SAT problems (easy).

| $n = 50$, $m = 1058$ | | H1 | H2 | H1R | H2R | BRR |
|---|---|---|---|---|---|---|
| *time* | (10 Yes) | 12.76 | 13.32 | 11.03 | 12.07 | 75.20 |
| | | (12.86) | (11.92) | (10.50) | (10.27) | (71.66) |
| | (10 No) | 37.84 | 41.29 | 37.53 | 35.78 | 233.08 |
| | | (2.77) | (3.10) | (1.61) | (2.03) | (14.56) |
| *nodes* | (10 Yes) | 13,424 | 13,308 | 6,167 | 6,379 | 9,349 |
| | | (14,018) | (12,070) | (6,038) | (5,509) | (8,876) |
| | (10 No) | 40,086 | 41,228 | 21,165 | 18,971 | 28,815 |
| | | (3,106) | (3,134) | (922) | (1,082) | (1,546) |

TABLE 2. 5-SAT problems (easy).

As one might expect, both Rule 2 and algorithm Relax are less effective when the length of the clauses increases. Indeed, for 3-SAT, H1 is outperformed by all other DPL algorithms, and H2R gives the best results. On the contrary, for 5-SAT, H2 is worse than H1, and the improvements for H1R and H2R are

almost negligible. Here, an interesting interaction effect can be pointed out: Rule 2 gives poor results in H2, but it is worth applying if combined with the relaxation. Similar effects are also being observed elsewhere, but we do not know exactly how to explain this phenomenon. Finally, BRR is much slower than DPL algorithms, even if it enumerates less nodes than H1.

Note that satisfiable and unsatisfiable instances show a quite different distribution. On average, satisfiable instances are easier to solve, but the observed results (both time and nodes) are spread in a much wider range; indeed, the standard deviation is greater than the mean in many cases. It must be remarked that the above behaviour does not seem to be related to some specific algorithm. In particular, for each subset of instances, the ratio of the standard deviation to the mean is roughly the same for all the algorithms. We might conclude that improvements in efficiency do not come at the expense of stability, at least as long as 3-SAT and 5-SAT problems are considered.

In tables 3 and 4 we consider problems with larger size. The results substantially confirm our previous observations on the relative efficiency of DPL algorithms. It must be remarked that improvements, in particular those obtained by H2R, are more impressive for problems of larger size.

| $n = 300, m = 1275$ | | H1 | H2 | H1R | H2R |
|---|---|---|---|---|---|
| *time* | (5 Yes) | 1,361 | 525 | 650 | 403 |
| | (5 No) | 2,492 | 1,076 | 1,759 | 789 |
| *nodes* | (5 Yes) | 1,090,935 | 289,251 | 77,016 | 41,594 |
| | (5 No) | 1,940,120 | 582,219 | 220,844 | 86,121 |

TABLE 3. 3-SAT problems (hard).

| $n = 70, m = 1500$ | | H1 | H2 | H1R | H2R |
|---|---|---|---|---|---|
| *time* | (4 Yes) | 243 | 319 | 162 | 234 |
| | (6 No) | 1,218 | 1,387 | 1,169 | 1,074 |
| *nodes* | (4 Yes) | 199,346 | 245,575 | 68,104 | 90,156 |
| | (6 No) | 998,115 | 1,051,175 | 464,839 | 408,093 |

TABLE 4. 5-SAT problems (hard).

Even though we do not expect Rule 2 and Relax to be effective for $k$-SAT problems with large $k$, better results can be obtained when the length of the clauses is spread in a wider interval. In particular, we considered the case in which the clause length varies between 3 and 7 with a discrete uniform distribution. In table 5 we report the results for a set of 10 (reasonably large) instances. Observe that algorithm H1 is outperformed by the other versions of DPL; in particular, Rule 2 seems to be quite effective. Even though the average clause

length is the same as in 5-SAT problems, the improvements are comparable to the ones obtained for 3-SAT.

| $n = 150, m = 1900$ | | H1 | H2 | H1R | H2R |
|---|---|---|---|---|---|
| *time* | (5 Yes) | 229 | 99 | 229 | 102 |
| | (5 No) | 746 | 471 | 568 | 421 |
| *nodes* | (5 Yes) | 126,833 | 47,891 | 37,858 | 14,062 |
| | (5 No) | 396,653 | 211,493 | 87,764 | 57,678 |

TABLE 5. Discrete Uniform Distribution in [3..7].

We consider next two classes of instances encoding *inductive inference* problems, where an unknown boolean function must be (approximately) identified given its results on a set of input samples.

In the *parity learning* problems [5] the function to identify is the parity of a suitable (unknown) subset of the input bits. There exists a *standard* encoding of these instances, as well as a *compressed* one, the latter being a more compact version of the former (e.g. not containing unit clauses). In table 6 we report the results for a set of five satisfiable instances, in both formats, corresponding to boolean functions of 16 bits.

| | | H1 | H2 | H1R | H2R | BRR |
|---|---|---|---|---|---|---|
| *standard* | *time* | 92.11 | 114.95 | 91.01 | 95.27 | 620 |
| | *nodes* | 52,435 | 52,412 | 4,105 | 3,966 | 29,484 |
| *compressed* | *time* | 225.47 | 82.92 | 1,784 | 44.04 | 401 |
| | *nodes* | 169,237 | 53,359 | 50,283 | 4,223 | 30,968 |

TABLE 6. Parity learning problems.

The results for the two encodings show a quite remarkable difference. For the standard format, Rule 2 obtains worse results than Rule 1, and Relax does not lead to significant improvements, even though it enumerates much less nodes. On the contrary, for compressed problems, Rule 2 is quite effective, while algorithm H1R is more than 8 times slower than H1; nevertheless, the best results are obtained by H2R. Again, we can see here a relevant interaction effect. Observe that BRR is much slower than the other algorithms, and works better on the compressed format.

The above results show that the efficiency of algorithms can be deeply affected by the encoding of the input. Clearly, a better knowledge of the structure of parity learning problems might give an insight of this behaviour.

We consider next the *boolean function synthesis* problems introduced in [17]. Here, the function to be identified must correspond to a two-level AND/OR logical circuit, with a fixed maximum number of AND gates. We consider two sets of problems, corresponding to functions of 16 and 32 bits respectively. Remark that

H1R and H2R did not solve all the instances, thus we consider only a subset of 15 out of the 17 32-bits problems. Instead, BRR solved both the 32-bits and the (quite larger) 16-bits problems. Results are reported in table 7; standard deviations are written within parenthesis.

| 32-bits | | H1 | H2 | H1R | H2R |
|---|---|---|---|---|---|
| *time* | (15 Yes) | 47.61 (87.01) | 25.99 (60.80) | 1,520 (2,581) | 744 (1,708) |
| *nodes* | (15 Yes) | 15,466 (45,717) | 1,778 (3,962) | 9,593 (26,335) | 1,613 (3,570) |

| 32-bits | | BRR | 16-bits | | BRR |
|---|---|---|---|---|---|
| *time* | (17 Yes) | 1.75 (1.73) | *time* | (10 Yes) | 3.01 (1.70) |
| *nodes* | (17 Yes) | 76 (230) | *nodes* | (10 Yes) | 49 (87.12) |

TABLE 7. Inductive Inference problems.

For this class of problems, BRR finds feasible B-reductions quite easily, and thus greatly outperforms our DPL algorithms. Observe that Rule 2 is effective, but not enough to close the gap. On the contrary, the relaxation gives quite poor results; this behaviour can be explained considering the structure of the input. These instances contain a large amount of binary clauses, and a smaller amount of much larger clauses. However, only binary clauses of the following types appear:

$$x_i \vee x_j, \qquad x_i, x_j \in X;$$
$$x \vee \neg y, \qquad x \in X,\ y \in Y;$$

where $X$ and $Y$ give a partition of the variables. Therefore, the relaxation examines a large number of binary clauses in almost all the nodes of the enumeration trees, but can detect contradictions only if further clauses of length 2 or 3 are generated, and this happens quite deep in the enumeration tree. Indeed, Deduce returns nodes in less than 1/1000 of the cases, and almost always only one node is assigned a value.

In table 8 we consider some 3-SAT problems in the class defined by Urquhart [21]; these instances encode a particular graph coloring problem with parity constraints. We solved a set of 5 unsatisfiable instances, obtained from the same graph of 40 nodes introducing different parity constraints; here, we have $n = 60$ and $m = 160$. For these problems, our DPL algorithms show a deterministic and rather pathological behaviour. In each instance, both H1 and H2 enumerate exactly $2^{22} - 1$ nodes; this number reduces to $2^{20} - 1$ when the relaxation is used, but execution times increase. Algorithm BRR shows an opposite behaviour: it is

faster than DPL algorithms for all instances except one, where it is about twice slower. Nevertheless, it obtains the best average results.

| | H1 | H2 | H1R | H2R | BRR |
|---|---|---|---|---|---|
| *time* | 264 | 308 | 283 | 308 | 234 |
| *nodes* | 4,194,303 | 4,194,303 | 1,048,575 | 1,048,575 | 392,040 |

TABLE 8. Urquhart's 3-SAT problems.

In table 9 we considered some examples of AIM problems [1]. Even though they do not show the same deterministic behaviour as above, our DPL algorithms are much slower than BRR. Note that Rule 2 is useful, at least for unsatisfiable instances, while the relaxation gives worse results.

| | | H1 | H2 | H1R | H2R | BRR |
|---|---|---|---|---|---|---|
| *time* | (4 Yes) | 3.57 | 4.62 | 4.17 | 4.88 | 2.62 |
| | (4 No) | 904.77 | 598.04 | 1,106.85 | 680.54 | 46.45 |
| *nodes* | (4 Yes) | 38,550 | 45,628 | 26,009 | 33,916 | 2,616 |
| | (4 No) | 13,433,872 | 6,307,424 | 11,425,563 | 5,169,668 | 51,208 |

TABLE 9. AIM problems.

Admittedly, Urquhart's and AIM problems have a quite artificial structure, thus the obtained results should be taken with a grain of salt. Nevertheless, a careful analysis of these problems might give an insight of the actual behaviour of algorithms, possibly pointing out some of their weaknesses.

Finally, in table 10 we report some statistics describing the behaviour of the relaxation within algorithm H2R. The first row (*rate*) gives the rate of success of procedure Deduce, that is, the percentage of trees in which some nodes to be set where found. The second row (*size*) gives the average number of nodes that are set to a value in case of success. In table 10, we denote by DUD and U40 the problems in tables 5 and 8, respectively.

| | 3-SAT | | 5-SAT | | DUD | parity | | U40 | AIM |
|---|---|---|---|---|---|---|---|---|---|
| | *easy* | *hard* | *easy* | *hard* | | *unc.* | *comp.* | | |
| *rate* | 19.05 | 13.31 | 69.33 | 58.74 | 23.94 | 2.31 | 9.47 | 22.32 | 59.68 |
| *size* | 2.46 | 2.63 | 1.67 | 1.81 | 2.20 | 6.7 | 7.13 | 1.00 | 1.36 |

TABLE 10. Algorithm Relax.

The results obtained on parity problems are quite impressive, and shed light on the different behaviour of H2R for compressed and uncompressed format. Note also that *size* was equal to 1 for all Urquhart's problems, in accordance with the deterministic behaviour of DPL algorithms for this class. For 5-SAT

problems, the high value of *rate* may be due to the fact that Deduce is used quite seldom, and rather deep in the enumeration tree.

In general, we may argue that a high rate of success not necessarily corresponds to an improvement in the efficiency; the best results are obtained when both *rate* and *size* are sufficiently high.

## 5. Conclusions and future work

Even though our computational experience is necessarily limited, there seems to be enough evidence for some general remarks.

A first observation concerns the stability of DPL algorithms. On one side, our experience confirms the relevance of branching rules, and the necessity of further investigations on this topic. On the other side, DPL methods with sophisticated branching rules do not seem to be always optimal; as soon as the structure of the instances becomes more deterministic, a different enumerative approach can give better results. This may provide some guidelines for a further development of algorithms. For example, the results obtained for the boolean function synthesis problems may suggest a combination of *deterministic* local search and specialized enumeration strategies.

We can derive some general guidelines also for our relaxation technique: good results can be obtained when instances contain a sufficient percentage of small clauses with a sufficiently random distribution. We may argue that our relaxation works fine when also DPL does: it can take advantage of a sophisticated branching rule, but it does not help when the DPL approach is not suitable. Nevertheless, some of the results are encouraging; we conclude that our relaxation may become a useful algoritmic tool, at least for some classes of problems.

Clearly, our methods can be extended and improved. For example, procedure Deduce can be easily modified in order to find sets of variables that must take the same value (a formal analysis of some related techniques can be found in [20]). In addition, it is possible to add the shrunken hyperarcs generated by Deduce to the current hypergraph, thus obtaining a tighter representation. Clearly, the above extensions require a careful analysis of data structures and implementation details. In conclusion, hypergraph methods for SAT may represent a challenging topic for future theoretical and practical developments.

**Acknowledgements.** This work has been written during a stage at the CRT, Centre de Recherche sur les Transports, Université de Montréal. I wish to thank all the members of the CRT for their constant help and friendship. I am also particularly indebted to Mihnea Stan for several helpful discussions.

## REFERENCES


1. Y. Asahiro, K. Iwama, and E. Miyano, *Random generation of test instances with controlled attributes*, this volume.
2. A. Billionnet and A. Sutter, *An efficient algorithm for the 3-satisfiability problem*, Oper. Res. Lett. **12** (1992), 29–36.
3. M. Böhm and E. Speckenmeyer, *A fast parallel SAT-solver — efficient workload balancing*, Tech. Rep. 94-159, Informatik, Universität zu Köln, Germany, 1994, to appear in Ann. Math. Artificial Intelligence.
4. H. Kleine Büning and M. Buro, *Report on a SAT competition*, research report 110, FB 17 (Mathematik/Informatik) Universität Paderborn, Germany, November 1992.
5. J. Crawford, contributed to the Second DIMACS Algorithm Implementation Challenge, 1993.
6. J. Crawford and L. Auton, *Experimental results on the cross-over point in satisfiability problems*, Proceedings of the $11^{th}$ AAAI Spring Symposium, 1993, pp. 22–28.
7. O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier, *SAT versus UNSAT*, this volume.
8. J. W. Freeman, *Failed literals in the Davis-Putnam procedure for SAT*, contributed to the Second DIMACS Algorithm Implementation Challenge, 1993.
9. G. Gallo, G. Longo, S. Nguyen, and S. Pallottino, *Directed hypergraphs and applications*, Discrete Appl. Math. **42** (1993), no. 2-3, 177–201.
10. G. Gallo and D. Pretolani, *A new algorithm for the propositional satisfiability problem*, research report TR-18/92, Computer Science Department, University of Pisa, Italy, June 1992, to appear on Discrete Appl. Math.
11. P. Hansen, B. Jaumard, and M. Minoux, *A linear expected time algorithm for deriving all the logical conclusions implied by a set of boolean inequalities*, Math. Programming **34** (1986), 223–231.
12. J. N. Hooker, *Needed: an empirical science of algorithms*, Oper. Res. **42** (1994), 201–212.
13. J. N. Hooker and C. Fedjki, *Branch and cut solution of inference problems in propositional logic*, Ann. Math. Artificial Intelligence **1** (1990), 123–140.
14. J. N. Hooker and V. Vinay, *An empirical study of branching rules for satisfiability*, Presented at the $3^{rd}$ Sympos. on Artificial Intelligence and Math., January 1994.
15. B. Jaumard, M. Stan, and J. Desrosiers, *Tabu search and a quadratic relaxation for the satisfiability problem*, this volume.
16. R. G. Jeroslow and J. Wang, *Solving propositional satisfiability problems*, Ann. Math. Artificial Intelligence **1** (1990), 167–187.
17. A. P. Kamath, N. K. Karmarkar, K. G. Ramakrishnan, and M. G. C. Resende, *A continuous approach to inductive inference*, Math. Programming **57** (1992), 215–238.
18. D. Loveland, *Automated theorem proving: a logical basis*, North Holland, 1978.
19. D. Pretolani, *A linear time algorithm for unique Horn satisfiability*, Inform. Process. Lett. **48** (1993), no. 2, 61–66.
20. ———, *Satisfiability and Hypergraphs*, Ph.D. thesis, Department of Computer Science, University of Pisa, Italy, March 1993, TD-12/93.
21. A. Urquhart, *Hard examples for resolution*, J. Assoc. Comput. Mach. **34** (1987), 209–219.

**Second DIMACS Challenge**
Satisfiability Benchmark Results

*GENERAL INFORMATION*

*Author:* Daniele Pretolani
*Title:* Efficiency and Stability of Hypergraph SAT Algorithms.

*Name of Algorithm:* H2R
*Brief Description of Algorithm:*
Complete; DPL method with pruning procedure.

*Name of Algorithm:* BRR
*Brief Description of Algorithm:*
Complete; B-Reduction method with pruning procedure.

*Type of Machine:* SUN SPARCstation 2
*Compiler and flags used:* `cc`

*MACHINE BENCHMARKS*

*User time for instances:*

| r100.5 | r200.5 | r300.5 | r400.5 | r500.5 |
|---|---|---|---|---|
| 0.20 | 5.53 | 48.19 | 297.45 | 1138.35 |

*ALGORITHM BENCHMARKS*

*Authors' Comments:*
Algorithm BRR reads input from strings, as described in [4]. The time needed to translate the input file into a string is not included in the reported times. In some cases (e.g. for problems ii32$xx$) translating the file took more time than solving the instance.

*Results on Benchmark Instances: Algorithm H2R*

| Name | Runs (Fail) | Time Min | Time Avg (Std. Dev.) | Time Max | Result |
|---|---|---|---|---|---|
| aim-100-2_0-no-1 | | | 0.06667 | | NO |
| aim-100-2_0-no-2 | | | 567.717 | | NO |
| aim-100-2_0-no-3 | | | 1663.63 | | NO |
| aim-100-2_0-no-4 | | | 513.767 | | NO |
| aim-100-2_0-yes1-1 | | | 1.78333 | | YES |
| aim-100-2_0-yes1-2 | | | 4.91667 | | YES |
| aim-100-2_0-yes1-3 | | | 12.6833 | | YES |
| aim-100-2_0-yes1-4 | | | 0.03333 | | YES |
| bf0432-007.cnf | | | 43.4167 | | NO |
| bf2670-001.cnf | | | 54.4833 | | NO |
| dubois20.cnf | | | 375.100 | | NO |
| dubois21.cnf | | | 768.133 | | NO |
| f400.cnf | | | 5727.32 | | YES |
| f800.cnf | DNR | | | | |
| f1600.cnf | DNR | | | | |
| f3200.cnf | DNR | | | | |
| f6400.cnf | DNR | | | | |
| g125.17.cnf | DNR | | | | |
| g125.18.cnf | DNR | | | | |
| g250.15.cnf | DNR | | | | |
| g250.29.cnf | DNR | | | | |
| ii32b3.cnf | | | 1064.9 | | YES |
| ii32c3.cnf | | | 313.53 | | YES |
| ii32d3.cnf | DNR | | | | |
| ii32e3.cnf | | | 84.383 | | YES |
| par16-2-c.cnf | | | 48.05 | | YES |
| par16-4-c.cnf | | | 115.93 | | YES |
| par32-2-c.cnf | DNR | | | | |
| par32-4-c.cnf | DNR | | | | |
| par8-2-c.cnf | | | 0.0833 | | YES |
| par8-4-c.cnf | | | 0.0833 | | YES |
| pret150_25.cnf | DNR | | | | |
| pret150_75.cnf | DNR | | | | |
| pret60_25.cnf | | | 310.53 | | NO |
| pret60_75.cnf | | | 309.65 | | NO |
| ssa0432-003.cnf | | | 0.833 | | NO |
| ssa2670-141.cnf | DNR | | | | |
| ssa7552-038.cnf | | | 3.367 | | YES |
| ssa7552-158.cnf | | | 2.283 | | YES |
| ssa7552-159.cnf | | | 2.683 | | YES |
| ssa7552-160.cnf | | | 2.800 | | YES |

*Results on Benchmark Instances: Algorithm BRR*

| Name | Runs (Fail) | Time Min | Avg (Std. Dev.) | Max | Result |
|---|---|---|---|---|---|
| aim-100-2_0-no-1 | | | 76.45 | | NO |
| aim-100-2_0-no-2 | | | 7.5 | | NO |
| aim-100-2_0-no-3 | | | 38.73 | | NO |
| aim-100-2_0-no-4 | | | 63.18 | | NO |
| aim-100-2_0-yes1-1 | | | 3.817 | | YES |
| aim-100-2_0-yes1-2 | | | 5.617 | | YES |
| aim-100-2_0-yes1-3 | | | 1.033 | | YES |
| aim-100-2_0-yes1-4 | | | 0.033 | | YES |
| bf0432-007.cnf | | | 518.317 | | NO |
| bf2670-001.cnf | DNR | | | | |
| dubois20.cnf | | | 667.95 | | NO |
| dubois21.cnf | | | 1314.7 | | NO |
| f400.cnf | DNR | | | | |
| f800.cnf | DNR | | | | |
| f1600.cnf | DNR | | | | |
| f3200.cnf | DNR | | | | |
| f6400.cnf | DNR | | | | |
| g125.17.cnf | DNR | | | | |
| g125.18.cnf | DNR | | | | |
| g250.15.cnf | DNR | | | | |
| g250.29.cnf | DNR | | | | |
| ii32b3.cnf | | | 1.150 | | YES |
| ii32c3.cnf | | | 4.767 | | YES |
| ii32d3.cnf | | | 2.983 | | YES |
| ii32e3.cnf | | | 0.8 | | YES |
| par16-2-c.cnf | | | 338.233 | | YES |
| par16-4-c.cnf | | | 211.883 | | YES |
| par32-2-c.cnf | DNR | | | | |
| par32-4-c.cnf | DNR | | | | |
| par8-2-c.cnf | | | 0.03 | | YES |
| par8-4-c.cnf | | | 0.05 | | YES |
| pret150_25.cnf | DNR | | | | |
| pret150_75.cnf | DNR | | | | |
| pret60_25.cnf | | | 159.7 | | NO |
| pret60_75.cnf | | | 533.3 | | NO |
| ssa0432-003.cnf | | | 0.85 | | NO |
| ssa2670-141.cnf | | | 19881.6 | | NO |
| ssa7552-038.cnf | | | 1.55 | | YES |
| ssa7552-158.cnf | | | 0.50 | | YES |
| ssa7552-159.cnf | | | 0.50 | | YES |
| ssa7552-160.cnf | | | 0.67 | | YES |

DIPARTIMENTO DI INFORMATICA, UNIVERSITÀ DI PISA, CORSO ITALIA 40, 56125 PISA, ITALY.
*E-mail address*: pretola@di.unipi.it