International Journal on Artificial Intelligence Tools
Vol. XX, No. X (2014) 1–27
© World Scientific Publishing Company

PREPROCESSING IN PROPOSITIONAL SATISFIABILITY USING BOUNDED (2,K)-CONSISTENCY ON REGIONS WITH A LOCALLY DIFFICULT CONSTRAINT SETUP

PAVEL SURYNEK

Department of Theoretical Computer Science and Mathematical Logic,
Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
Malostranské náměstí 25, 118 00 Praha 1, Czech Republic

Graduate School of Maritime Sciences, Division of Maritime Management Sciences,
Kobe University, Japan
5-1-1 Fukae-minamimachi, Higashinada-ku, Kobe 658-0022, Japan

pavel.surynek@mff.cuni.cz

Initial version submitted: April 4, 2012
Reviews received: November 15, 2012; Revision submitted: February 15, 2013;
Proofread version: June 1, 2013

A new type of partially global consistency derived from $(2, k)$-consistency called bounded $(2, k)$-consistency (B2C-consistency) is presented in this paper. It is designed for application in propositional satisfiability (SAT) as a building block for a preprocessing tool. Together with the new B2C-consistency a special mechanism for selecting regions of the input SAT instance with difficult constraint setup was also proposed. This mechanism is used to select suitable difficult sub-problems whose simplification through consistency can lead to a significant reduction in the effort needed to solve the instance. A new prototype preprocessing tool `preprocessSIGMA` which is based on the proposed techniques was implemented. As a proof of new concepts a competitive experimental evaluation on a set of relatively difficult SAT instances was conducted. It showed that our prototype preprocessor is competitive with respect to the existent preprocessing tools `LiVer`, `NiVer`, `HyPre`, blocked clause elimination (`BCE`), and `Shatter` with `saucy 3.0`.

Keywords: SAT; CSP; SAT preprocessing; local consistency; global consistency; $(2, k)$-consistency; probability; difficult instances; hyper-resolution; blocked clause elimination; symmetry;

1. Introduction and Motivation

Recent works dealing with difficult instances of *propositional satisfiability* (*SAT*) [1, 2, 8, 10, 26] indicate that an intelligent preprocessing focused on the structure of an instance can dramatically reduce the effort needed to solve it. Technically, the preprocessing task is done by transforming the input instance into another one (hopefully simpler), which is subsequently submitted to a general purpose *SAT solver* [8, 13]. It is crucial that the preprocessing step is fast enough relative to the runtime of the SAT solver on the preprocessed instance.

In this work, we further develop ideas from [26] where the input propositional formula is interpreted as a graph, in which graph structures – namely complete sub-graphs – are identified and, after some calculation involving the number and the size of complete sub-graphs, an inference is made. The drawback of the original idea from [26] is that it requires the input instance to be relatively well structured to be able to identify acceptable complete sub-graph decomposition. In this paper, we overcome this major drawback using two new techniques. First, a **new type of consistency** derived from $(2,k)$-*consistency* [11] called *bounded* $(2,k)$-*consistency with complete graphs* (*B2C-consistency*) is proposed. It uses graph interpretation of a sub-problem on which reasoning over its decomposition into complete sub-graphs is performed and can therefore be regarded as a partially global reasoning mechanism. Second, a **new mechanism for selecting** a sub-problem suitable for applying the consistency is proposed. In order to maximize the benefit of inferences made through consistency, we proposed to apply it on regions of the input instance with a locally difficult constraint setup. It means that we are trying to choose such a sub-problem for applying the consistency that, in itself, is difficult in a certain sense (focusing on the difficulty proved to be beneficial in [26] but the previous technique required the whole instance to exhibit a difficult constraint setup). We were primarily inspired by the difficulty of well known problems such as the *pigeon/hole principle* (*P/H principle*) or *FPGA routing* [1, 2] and we are trying to select regions of the instance which, in terms of certain properties, are similar to these difficult instances. To do this, a characteristic called the *expected number of satisfied tuples of values* is used so that regions that have this characteristic similar to difficult instances are used as sub-problems on which B2C-consistency is applied. In this way, we are able to discover sub-problems with a hidden difficulty and simplify them with the proposed consistency reasoning, which provides a faster solution of the output instance.

As a validation of the proposed concepts a prototype SAT preprocessing tool `preprocessSIGMA` [27] based on B2C-consistency and a new sub-problem selection technique have been implemented. The performed experimental evaluation showed that our prototype preprocessing tool is competitive with respect to existent prominent preprocessing tools such as `LiVer` [25], `NiVer` [25], `HyPre` [5], *blocked clause elimination* [16, 20] (`precosat-465`), and `Shatter` with `saucy 3.0` [2, 21] which is so far the latest version.

This work has been iteratively developed and preceding work related to the presented one appeared in [26]. The organization of the paper is as follows: basic concepts from *constraint programming* [11] and SAT are introduced in **Section 2**. The concept of B2C-consistency is subsequently developed (**Section 3**). The following section (**Section 4**) deals with the question of how to build a preprocessing tool exploiting B2C-consistency. Finally (**Section 5**), an extensive experimental evaluation focused on the competitiveness and the investigation of internal properties of the implemented preprocessor is presented.

2. Background from Constraint Programming and Propositional Satisfiability

Let us start with the basic notation and definitions used in the rest of the paper. This section represents the basic background from *constraint programming* [11] and *propositional satisfiability* [8], which the new concepts rely on.

Tuples and *lists* (that is, sequences) consisting of some objects will be denoted using brackets (for example $[x,y]$ denotes an ordered pair consisting of two objects x and y; [] denotes the empty list).

Definition 1 ***(Constraint Satisfaction Problem)*** [11]**.** A *constraint satisfaction problem* (*CSP*) over a given finite universe $\mathbb{D}$ is a triple (X, D, C) where X is a finite set of *variables*, C is a finite set of *constraints*, and $D: X \longrightarrow 2^{\mathbb{D}}$ is a function assigning each variable a finite *domain*. A constraint $c \in C$ is a construct of the form $\langle [x_1^c, x_2^c, \dots, x_{a^c}^c], R^c \rangle$ where $a^c \in \mathbb{N}$ is the *arity* of constraint c, $[x_1^c, x_2^c, \dots, x_{a^c}^c]$ with $x_i^c \in X$ for $i = 1,2, \dots, a^c$ is called a *scope* of c, and $R^c \subseteq D(x_1^c) \times D(x_2^c) \times \dots \times D(x_{a^c}^c)$ is a relation that enumerates a set of tuples of values for which constraint c is satisfied. □

For simplicity, it is sometimes assumed that $D(x) = \mathbb{D}$ for every $x \in X$. We will use this assumption as well in certain cases. Furthermore, it is assumed that we can reorder variables in the scope of a constraint arbitrarily using the above notation. For example, if there is a constraint $c = \langle [x, y], R^c \rangle$ in C, we can suppose that there is also an equivalent formulation of c as a constraint $e = \langle [y, x], R^e \rangle$ in C where relation R^e can be obtained from R^c by swapping its components.

Definition 2 ***(Solution of CSP)*** [11]**.** An assignment $v: X \longrightarrow \mathbb{D}$ such that $v(x) \in D(x)$ for every $x \in X$ is called a solution of a given CSP (X, D, C) if it is defined for every variable in X and all the constraints in C are satisfied by v. That is, it holds that $[v(x_1^c), v(x_2^c), \dots, v(x_{a^c}^c)] \in R^c$ for every constraint $c = \langle [x_1^c, x_2^c, \dots, x_{a^c}^c], R^c \rangle \in C$. □

Regarding constraints, we will sometimes use a formulation that some tuple of values is allowed/forbidden by a constraint, which means exactly that the tuple belongs or does not belong to the defining relation of the constraint.

Closely related to CSP is the *propositional satisfiability problem* (*SAT*) [8, 10]. It is introduced in the following two definitions. Note that in CSP we are trying to find a valuation of variables such that **all** the constraints are satisfied (that is, the conjunction of all the constraints is satisfied). In SAT the task is similar. We are trying to find a propositional valuation that satisfies **all** the clauses of the input formula (the formula has typically the form of a conjunction of clauses – CNF).

Definition 3 ***(Propositional Formula)*** [10]**.** A *propositional formula* in the *conjunctive normal form* (*CNF*) over a given set of propositional variables Ω is a conjunction: $\bigwedge_{i=1}^{n} \Gamma_i$ where $n \in \mathbb{N}_0$ and each Γ_i with $i \in \{1,2, \dots, n\}$ is a *clause* that puts into a disjunction *literals* over variables from Ω. That is, $\Gamma_i = \bigvee_{k=1}^{\alpha^i} \psi_k^i$ for $i = 1,2, \dots, n$ where $\alpha^i \in \mathbb{N}$ is the size of the clause and either $\psi_k^i = \beta$ or $\psi_k^i = \neg\beta$ for some variable $\beta \in \Omega$ for every $k = 1,2, \dots, \alpha^i$. □

Definition 4 ***(Propositional Satisfiability Problem)*** [10]**.** A *valuation* of propositional variables is an assignment $\omega: \Omega \longrightarrow \{FALSE, TRUE\}$. The given valuation of variables ω can be naturally extended to a valuation of formulae over Ω denoted as ω^*. A *propositional satisfiability problem* (*SAT*) with a formula Φ over Ω is the task of determining whether there exists a valuation ω of Ω such that $\omega^*(\Phi) = TRUE$. □

We are about to work with the concept of *consistencies* [11] in SAT which is, however, the concept from constraint programming used over CSPs. Hence, it is convenient to define translation of SAT to CSP so that we are able to work with consistencies in SAT through this translation. For this purpose, we chose the so-called *literal encoding* [6, 27] which provides such a translation in the natural way.

Definition 5 ***(Literal Encoding of SAT)*** [27]**.** Let $\Phi = \bigwedge_{i=1}^{n} \Gamma_i$ with $\bigvee_{l=1}^{\alpha^i} \psi_l^i$ for $i = 1,2,\dots,n$ be a propositional formula in CNF over Ω. A *literal encoding* of Φ is a CSP $E^0(\Phi) = (X_\Phi^0, D_\Phi^0, C_\Phi^0)$ where $X_\Phi^0 = \{\bar{\Gamma}_1, \bar{\Gamma}_2, \dots, \bar{\Gamma}_n\}$, $D_\Phi^0(\bar{\Gamma}_i) = \{\bar{\psi}_l^i | l = 1,2,\dots,\alpha^i\}$ for every $i = 1,2,\dots,n$; and there are constraints between all the pairs of variables as follows: $[\bar{\psi}_l^i, \bar{\psi}_t^j]$ where $\bar{\psi}_l^i \in D_\Phi^0(\bar{\Gamma}_i)$ and $\bar{\psi}_t^j \in D_\Phi^0(\bar{\Gamma}_j)$ is forbidden by relation R^c defining constraint $c = \langle [\bar{\Gamma}_i, \bar{\Gamma}_j], R^c \rangle$ with $i,j \in \{1,2,\dots,n\}$, $l \in \{1,2,\dots,\alpha^i\}$, and $t \in \{1,2,\dots,\alpha^j\}$ if there is $\beta \in \Omega$ such that either $\beta = \psi_l^i$ and $\neg\beta = \psi_t^j$ or $\neg\beta = \psi_l^i$ and $\beta = \psi_t^j$. □

The stripe above the generic symbols is used to distinguish constant symbols (with the stripe) which do not evaluate from variables (without the stripe) which do evaluate (down to other constants). Note that literal encoding is a *binary CSP*; that is, all the constraints have arity of at most 2.

For our purposes, literal encoding is further processed to capture constraints imposed by the original formula more explicitly (note that there is an incompatibility between complementary literals only at this stage). A new incompatibility is introduced as a constraint between every two literals ψ_l^i and ψ_t^j with $i,j \in \{1,2,\dots,n\}$ such that $i \neq j$, $l \in \{1,2,\dots,\alpha^i\}$ and $t \in \{1,2,\dots,\alpha^j\}$ if the *singleton unit propagation* [12, 26] with the setting $\psi_l^i = TRUE$ infers that $\psi_t^j = FALSE$ with respect to Φ (that is, it is set that $\psi_l^i = TRUE$; all the other variables are left unassigned and unit propagation follows). Let this modification of literal encoding be called an *explicit literal encoding* and it will be denoted as $E^1(\Phi) = (X_\Phi^1, D_\Phi^1, C_\Phi^1)$ (the upper index implies that the first stage of inference has been made).

We are now ready to define the so-called $(2,k)$-consistency [11]. It is a generalization of k-consistency [24] which checks whether a value is supported by a k-tuple of values from the domains of other variables. Within $(2,k)$-consistency, it is checked whether a pair of consistent values has a supporting k-tuple of values. If there is no such supporting k-tuple of values the value or the pair of values respectively can be ruled out from further consideration by an additional constraint.

An auxiliary operation of *projection* denoted as $T|_{A \to B}$ will be used to transform a tuple T into another tuple with respect to patterns A and B. Tuple T and pattern A are of the same size and B is contained by A. The result of the projection is obtained by matching pattern A on T followed by selecting components of T associated with their counterparts in A that correspond to B (for instance, $[1,2,3]|_{a,b,c \to c,b} = [3,2]$).

Definition 6 ***($(\mathbf{2},\boldsymbol{k})$-Consistency)*** [11]**.** Let $k \in \mathbb{N}$ be a natural number, (X,D,C) be a CSP, and $x_0, x_1, \dots, x_k, x_{k+1} \in X$ be a $(k+2)$-tuple of distinct variables. A pair of values $d_0 \in D(x_0)$ and $d_{k+1} \in D(x_{k+1})$ with $[d_0, d_{k+1}] \in R^c$ for every binary constraint $c = \langle [x_0, x_{k+1}], R^c \rangle$ in C is called to be $(2,k)$-*consistent* with respect to k-list of variables $x_1, x_2, \dots, x_k$ if there exists a k-tuple of values $d_1 \in D(x_1)$, $d_2 \in D(x_2)$,..., $d_k \in D(x_k)$ such that for every constraint $e = \langle [z_1^e, z_2^e, \dots, z_{a^e}^e], R^e \rangle$ in C with $\{z_1^e, z_2^e, \dots,$

$z^e_{a^e}\} \subseteq \{x_0, x_1, \dots, x_{k+1}\}$ it holds that $[d_0, d_1, \dots, d_{k+1}]|_{x_0, x_1, \dots, x_{k+1} \rightarrow z^e_1, z^e_2, \dots, z^e_{a^e}} \in R^e$. The pair of values $d_0 \in D(x_0)$ and $d_{k+1} \in D(x_{k+1})$ is called to be $(2,k)$-*consistent* if it is $(2,k)$-consistent with respect to all the k-tuples of variables $x_1, \dots, x_k \in X$. Finally, CSP (X, D, C) is called to be $(2,k)$-*consistent* if all the pairs of values from domains of every two distinct variables are $(2,k)$-consistent. □

It is not difficult to see that checking whether there exists a supporting k-tuple of values with respect to a fixed k-list of variables of unbounded size k is an NP-*complete problem* [22] in both k-consistency and $(2,k)$-consistency (for example, the *graph coloring problem* can be reduced to the task of searching for a supporting k-tuple). Hence, unless $P = NP$, the support cannot be found in polynomial time.

Another simple observation is that a support with respect to a fixed list of variables can be found in $\mathcal{O}(|\mathbb{D}|^k)$ by traversing all the involved k-tuples of values. This is also the currently best known upper bound of the time complexity of the search for a support within $(2,k)$-consistency enforcing algorithms [11].

Both the discussed higher level consistencies represent powerful techniques when k is bounded by the number of variables only. After enforcing k-consistency/$(2,k)$-consistency with k high enough it is possible to obtain a solution of a problem in a backtrack-free manner [11]. Without providing more details, the high enough k means that it is at least the *width of the constraint graph* of the given CSP which does not exceed the number of variables [14].

3. Bounded $(2,k)$-Consistency with Complete Graphs – B2C Consistency

Our new concept of the so-called *bounded* $(2,k)$-*consistency with complete graphs* (*B2C-consistency*) combines the inference strength of $(2,k)$-consistency with graph-based global reasoning. The global oriented reasoning in SAT which is of our interest was first introduced in [26]. Particularly, the idea of exploiting global information reflected in complete sub-graphs in a certain graph interpretation of the problem has been taken from the previous work and further elaborated. However, global reasoning itself turned out to be unilateral and hence not ideally suitable for using in SAT preprocessing. Therefore, it is suggested in this work to enhance global reasoning with $(2,k)$-consistency, which is quite universal and is supposed to help in cases where global reasoning alone is unsuitable. If both the approaches – global and $(2,k)$-consistency – are applied together a synergic effect is produced in certain situations.

Local consistencies such as k-consistency and related consistencies in SAT have been studied in several works [7, 23, 29]. The common approach in these works is to encode a given task so that a local consistency of interest is simulated by *unit propagation* [12]. Our approach takes an instance of SAT problem as a list of clauses (constraints) and applies the consistency directly without caring about the way how the original task has been encoded into the instance. The result is a set of forbidden value assignments in the case of *B2C-consistency* which is subsequently submitted to a SAT solver together with the original instance as a list of additional clauses.

The major obstacle with $(2,k)$-consistency is that it is difficult to enforce because it is necessary to search for a consistent k-tuple of values, which means to traverse the search space of the size of $|\mathbb{D}|^k$ in the worst case (supposed that all the variables have an identical domain of $\mathbb{D}$). Hence, to preserve low computation costs of the consistency enforcing algorithm we suggest to bound the consistency in some way. It has been chosen to bound the number of steps of the search for a consistent k-tuple by constant Λ.

B2C-consistency is again defined with respect to a $(k+2)$-tuple of distinct variables. Again, it checks whether a given pair of values from domains of two distinct variables have a supporting k-tuple in domains of remaining k variables. The following sections describe how the new consistency is enforced supposed that $(k+2)$-list of variables has been already determined. The process of selecting a promising $(k+2)$-tuple is discussed later.

3.1. *A Graph Derived from SAT – Graph Interpretation*

Let $E^1(\Phi) = (X^1_\Phi, D^1_\Phi, C^1_\Phi)$ be an explicit literal encoding of a given propositional formula Φ. Next, let us have $k \in \mathbb{N}$ and an ordered $(k+2)$-tuple of selected variables $K^2_+ = [\bar{\Gamma}_{i_0},\ \bar{\Gamma}_{i_1}, \bar{\Gamma}_{i_2}, \dots, \bar{\Gamma}_{i_k}, \bar{\Gamma}_{i_{k+1}}] \subseteq X^1_\Phi$ with $i_0, i_1, \dots, i_{k+1} \in \{1,2,\dots,n\}$ where $i_\zeta \neq i_\xi$ for $\zeta, \xi \in \{0,1,\dots,k+1\}$ with $\zeta \neq \xi$.

Propositional Formula Φ

$\Omega = \{\beta_1, \beta_2, \beta_3\}$

$\Phi =$
$\Gamma_1: (\ \beta_1 \vee \ \beta_2) \quad \wedge$
$\Gamma_2: (\neg\beta_1 \vee \neg\beta_2) \quad \wedge$
$\Gamma_3: (\ \beta_1 \vee \neg\beta_3) \quad \wedge$
$\Gamma_4: (\neg\beta_1 \vee \ \beta_2 \vee \beta_3)$

Explicit Literal Encoding $E^1(\Phi)$

$E^1(\Phi) = (X^1_\Phi, D^1_\Phi, C^1_\Phi) X^1_\Phi = \{\bar{\Gamma}_1, \bar{\Gamma}_2, \bar{\Gamma}_3, \bar{\Gamma}_4\}$

$D^1_\Phi(\bar{\Gamma}_1) = \{\bar{\beta}_1, \bar{\beta}_2\} \qquad D^1_\Phi(\bar{\Gamma}_2) = \{\neg\bar{\beta}_1, \neg\bar{\beta}_2\}$
$D^1_\Phi(\bar{\Gamma}_3) = \{\bar{\beta}_1, \neg\bar{\beta}_3\} \qquad D^1_\Phi(\bar{\Gamma}_4) = \{\neg\bar{\beta}_1, \bar{\beta}_2, \bar{\beta}_3\}$

$C^1_\Phi = \{c_{\{1,2\}}, c_{\{1,3\}}, c_{\{1,4\}}, c_{\{2,3\}}, c_{\{2,4\}}, c_{\{3,4\}}\}$

$$
\begin{aligned}
c_{\{1,2\}} &= \langle [\bar{\Gamma}_1, \bar{\Gamma}_2], D^1_\Phi(\bar{\Gamma}_1) \times D^1_\Phi(\bar{\Gamma}_2) \setminus \{[\bar{\beta}_1, \neg\bar{\beta}_1], [\bar{\beta}_2, \neg\bar{\beta}_2]\}\rangle \\
c_{\{1,3\}} &= \langle [\bar{\Gamma}_1, \bar{\Gamma}_3], D^1_\Phi(\bar{\Gamma}_1) \times D^1_\Phi(\bar{\Gamma}_3) \setminus \{[\bar{\beta}_1, \neg\bar{\beta}_3], [\bar{\beta}_2, \bar{\beta}_1]\}\rangle \\
c_{\{1,4\}} &= \langle [\bar{\Gamma}_1, \bar{\Gamma}_4], D^1_\Phi(\bar{\Gamma}_1) \times D^1_\Phi(\bar{\Gamma}_4) \setminus \{[\bar{\beta}_1, \neg\bar{\beta}_1], [\bar{\beta}_1, \bar{\beta}_2], [\bar{\beta}_2, \bar{\beta}_3]\}\rangle \\
c_{\{2,3\}} &= \langle [\bar{\Gamma}_2, \bar{\Gamma}_3], D^1_\Phi(\bar{\Gamma}_2) \times D^1_\Phi(\bar{\Gamma}_3) \setminus \{[\neg\bar{\beta}_1, \bar{\beta}_1], [\neg\bar{\beta}_2, \neg\bar{\beta}_3]\}\rangle \\
c_{\{2,4\}} &= \langle [\bar{\Gamma}_2, \bar{\Gamma}_4], D^1_\Phi(\bar{\Gamma}_2) \times D^1_\Phi(\bar{\Gamma}_4) \setminus \{[\neg\bar{\beta}_1, \bar{\beta}_3], [\neg\bar{\beta}_2, \neg\bar{\beta}_1], [\neg\bar{\beta}_2, \bar{\beta}_2]\}\rangle \\
c_{\{3,4\}} &= \langle [\bar{\Gamma}_3, \bar{\Gamma}_4], D^1_\Phi(\bar{\Gamma}_3) \times D^1_\Phi(\bar{\Gamma}_4) \setminus \{[\neg\bar{\beta}_1, \bar{\beta}_1], [\bar{\beta}_1, \bar{\beta}_2]\}\rangle
\end{aligned}
$$

Graph Interpretation $I(K^2_+)$

$K^2_+ = [\bar{\Gamma}_1, \bar{\Gamma}_2, \bar{\Gamma}_4] \qquad I(K^2_+) = (I_V, I_E)$

$I_V = \{\bar{\psi}^1_1, \bar{\psi}^1_2, \bar{\psi}^2_1, \bar{\psi}^2_2, \bar{\psi}^3_1, \bar{\psi}^3_2, \bar{\psi}^3_3\}$
$I_E =$
$\{\{\bar{\psi}^1_1, \bar{\psi}^2_1\}, \{\bar{\psi}^1_1, \bar{\psi}^4_1\}, \{\bar{\psi}^1_1, \bar{\psi}^4_2\},$
$\{\bar{\psi}^1_2, \bar{\psi}^2_2\}, \{\bar{\psi}^1_2, \bar{\psi}^4_3\},$
$\{\bar{\psi}^2_1, \bar{\psi}^4_3\},$
$\{\bar{\psi}^2_2, \bar{\psi}^4_1\},\{\bar{\psi}^2_2, \bar{\psi}^4_2\},\{\bar{\psi}^2_2, \bar{\psi}^4_3\}\}$

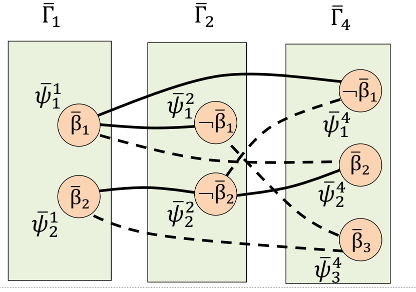


Figure 1. *Graph interpretation.* An original input Propositional formula Φ with four clauses is shown (upper left). Then a corresponding explicit literal encoding (upper right – that is, a literal encoding after singleton unit propagation) – the CSP model consisting of four variables is provided. The lower part depicts a graph interpretation over three variables selected in the CSP model. Dotted edges represent binary clauses that come from singleton unit propagation.

It is more convenient to define consistency with respect to an undirected graph derived from the constraint network. A target undirected graph will be represented by the so-called *graph interpretation* in the given context. It is defined with respect to K_{+}^{2} as an undirected graph $I(K_{+}^{2}) = (I_V, I_E)$ where a set of vertices I_V consists of $\bigcup_{\zeta=0}^{k+1}\{\bar{\psi}_l^{i_\zeta} | l = 1,2,\dots,\alpha^{i_\zeta}\}$ and a set of edges I_E contains edge $\{\bar{\psi}_l^{i_\zeta}, \bar{\psi}_t^{i_\xi}\}$ with $\zeta, \xi \in \{0,1,\dots,k+1\}$ such that $\zeta \neq \xi$, $l \in \{1,2,\dots,\alpha^{i_\zeta}\}$, and $t \in \{1,2,\dots,\alpha^{i_\xi}\}$ if it holds that $[\bar{\psi}_l^{i_\zeta}, \bar{\psi}_t^{i_\xi}] \notin R^c$ for some constraint $c = \langle[\bar{\Gamma}_{i_\zeta}, \bar{\Gamma}_{i_\xi}], R^c\rangle$ in C_{Φ}^{1} (edges stand for forbidden pairs of values; that is, an edge represents a *conflict*).

3.2. *Initial Setup of B2C-Consistency*

We are about to utilize structural information contained in the graph interpretation. It has been shown in the previous work [26] that useful structural information is constituted by the knowledge of complete constraint sub-graphs. Regarding the given context, we can observe that **at most one literal** can be satisfied in a complete sub-graph in the graph interpretation of a literal encoding of a SAT instance. If a large enough complete sub-graph is detected in the graph interpretation, its knowledge can be used for an efficient search space pruning or a strong global inference. The exact process of doing so will be explained in detail in the following text.

A decomposition into complete sub-graphs of a given graph interpretation $I(K_{+}^{2}) = (I_V, I_E)$ is constructed first. It is a task of finding number $\delta \in \mathbb{N}$ and sets I_V^1, I_V^2,..., $I_V^\delta \subseteq I_V$ called decomposition sets that satisfy the following conditions:

(i) $\bigcup_{i=1}^{\delta} I_V^i = I_V$; that is, all the vertices are covered by the decomposition;

(ii) $I_V^i \nsubseteq I_V^j$ for any two $i, j \in \{1,2,\dots,\delta\}$ such that $i \neq j$; that is, the decomposition is not allowed to contain redundancies;

(iii) I_V^i induces a complete sub-graph over edges from I_E for every $i \in \{1,2,\dots,\delta\}$;

(iv) $\forall u, v \in I_V$ with $\{u, v\} \in I_E$ there exists $i \in \{1,2,\dots,\delta\}$ such that $\{u, v\} \subseteq I_V^i$; that is, all the edges are covered by complete sub-graphs.

Observe that if no further objective is imposed on the decomposition into complete sub-graphs, it can be easily constructed by setting $\delta = |I_E|$ and putting endpoints of each edge into its own decomposition vertex set. On the other hand, the construction of decomposition with respect to any reasonable objective (such as maximizing the size of complete sub-graphs or minimizing number δ) is a difficult task [15, 22].

In our approach we try to obtain large complete sub-graphs. However, this requirement is not that strict so we have settled for a greedy approach for the construction of decomposition. The greedy algorithm used in our work is shown using a pseudo-code as Algorithm 1 ($\deg_{(V,E)}(v)$ denotes the number of edges from E adjacent to $\in V$).

The algorithm always prefers a vertex with the highest degree with respect to the remaining set of edges. Such a vertex is included into the constructed complete graph and the task is reduced to its neighborhood. This is repeated until the neighborhood of the currently constructed complete sub-graph is empty (a neighborhood of a complete sub-graph is a set of vertices that are connected to all of the vertices of the sub-graph). Once the complete sub-graph is finished its edges are removed from the original graph and the process continues until there are no edges.

The construction of a decomposition as shown in Algorithm 1 heuristically prefers a construction of a large complete sub-graph at the beginning. This strategy proved to produce decompositions of acceptable quality for sub-sequent usage within the B2C-consistency enforcing algorithm.

Proposition 1 (Greedy Time/Space Complexity). A greedy algorithm for the decomposition of a graph interpretation $I(K_+^2) = (I_V, I_E)$ into complete sub-graphs can be implemented to have the worst case time complexity of $\mathcal{O}(|I_E||I_V|^2)$. The corresponding worst case space complexity is of $\mathcal{O}(|I_V| + |I_E|)$. ■

Commentary: Observe that there may be up to $|I_E|$ complete sub-graphs in the decomposition (each edge constitutes a decomposition set). All the edges of the input graph interpretation may be investigated within the construction of an individual complete sub-graph which adds $|I_E|$ steps (which is $\mathcal{O}(|I_V|^2)$. Adding a vertex with the maximum degree into a complete sub-graph consumes $|I_V|$ steps while it may be repeated up to $|I_V|$ times. Altogether, we have $|I_V|^2$ steps for one complete sub-graph.

Regarding the space complexity it can be argued that several copies of the input graph need to be stored, which makes $\mathcal{O}(|I_V| + |I_E|)$ if the neighborhood of a vertex is represented using linked lists. ■

Algorithm 1. *Greedy algorithm for decomposing a graph interpretation into complete sub-graphs.* The output decomposition is returned as a sequence of decomposition sets of vertices where each of them induces a complete sub-graph.

```
function Decompose-Graph-Interpretation($I(K_+^2) = (I_V, I_E)$): sequence
     /* Parameters:    $I(K_+^2)$   - a graph interpretation for decomposing */
1:   $\delta \leftarrow 1$
2:   while $I_E \neq \emptyset$ do
3:        $I_V^\delta \leftarrow \emptyset$
4:        $(T_V, T_E) \leftarrow (I_V, I_E)$ /* an auxiliary graph for gradual dismantling */
5:        while $T_V \neq I_V^\delta$ do
6:             let $v_{max} \in T_V \setminus I_V^\delta$ be a vertex such that $\deg_{(T_V,T_E)}(v_{max}) =$
7:                  $\max\{\deg_{(T_V,T_E)}(v) \mid v \in T_V \setminus I_V^\delta\}$
8:             $I_V^\delta \leftarrow I_V^\delta \cup \{v_{max}\}$
9:             $T_V \leftarrow T_V \setminus \{u \mid \{v_{max}, u\} \notin T_E\}$)
10:            $T_E \leftarrow T_E \cap \binom{T_V}{2}$
11:       $I_E \leftarrow I_E \setminus \binom{I_V^\delta}{2}$
12:       $I_V \leftarrow I_V \setminus \{v \in I_V \mid \deg_{(I_V,I_E)}(v) = 0\}$
12:       $\delta \leftarrow \delta + 1$
13:  return $[I_V^1, I_V^2, \ldots, I_V^\delta]$
```

There are some more properties of the decomposition into complete sub-graphs. Note that decomposition sets intersect vertices corresponding to a domain of a single variable at most once. This is due to the fact that there are no edges between vertices corresponding to a single domain and due to condition (iii). On the other hand, a single vertex may be included in several decomposition sets.

3.3. *B2C-Consistency Enforcing Algorithm*

B2C-consistency will be defined algorithmically as this is the most natural way to do that. Suppose that a decomposition into complete sub-graphs of a given graph interpretation has already been constructed. The basic idea is to enforce bounded $(2,k)$-consistency using only Λ steps in the search for a supporting k-tuple. This search will be accompanied by a special pruning which will use the decomposition into complete sub-graphs to obtain more global reasoning. It is supposed that the search is done in a some systematic way by extending a partial selection of a supporting tuple of values. Regardless of the exact process of the search for the support, we can assume that some values/vertices are selected into the partial supporting tuple at every step of the process. The selection automatically rules out several other values/vertices – more precisely, values/vertices that are present together with the selected ones in some complete sub-graph are ruled out (this is due to the condition that no more than one literal can be selected in a complete sub-graph).

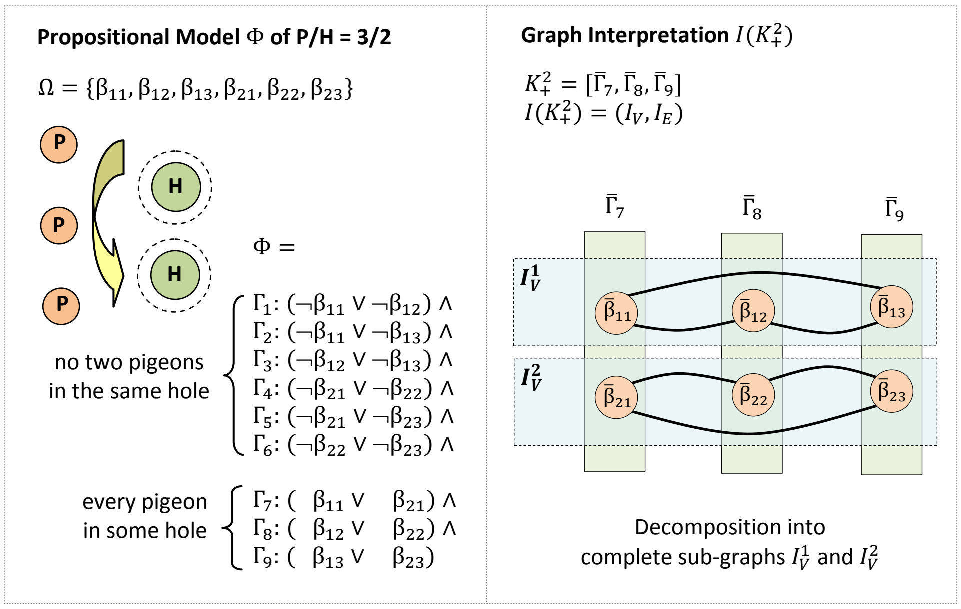

Propositional Model Φ of P/H = 3/2
$\Omega = \{\beta_{11}, \beta_{12}, \beta_{13}, \beta_{21}, \beta_{22}, \beta_{23}\}$
P
P
P
H
H
$\Phi =$
no two pigeons in the same hole
$\Gamma_1: (\neg\beta_{11} \vee \neg\beta_{12}) \wedge$
$\Gamma_2: (\neg\beta_{11} \vee \neg\beta_{13}) \wedge$
$\Gamma_3: (\neg\beta_{12} \vee \neg\beta_{13}) \wedge$
$\Gamma_4: (\neg\beta_{21} \vee \neg\beta_{22}) \wedge$
$\Gamma_5: (\neg\beta_{21} \vee \neg\beta_{23}) \wedge$
$\Gamma_6: (\neg\beta_{22} \vee \neg\beta_{23}) \wedge$
every pigeon in some hole
$\Gamma_7: (\beta_{11} \vee \beta_{21}) \wedge$
$\Gamma_8: (\beta_{12} \vee \beta_{22}) \wedge$
$\Gamma_9: (\beta_{13} \vee \beta_{23})$
Graph Interpretation $I(K_+^2)$
$K_+^2 = [\bar\Gamma_7, \bar\Gamma_8, \bar\Gamma_9]$
$I(K_+^2) = (I_V, I_E)$
$\bar\Gamma_7$
$\bar\Gamma_8$
$\bar\Gamma_9$
I_V^1
I_V^2
$\bar\beta_{11}$
$\bar\beta_{12}$
$\bar\beta_{13}$
$\bar\beta_{21}$
$\bar\beta_{22}$
$\bar\beta_{23}$
Decomposition into complete sub-graphs I_V^1 and I_V^2

Figure 2. *Pigeon hole (P/H) principle – graph interpretation with complete sub-graphs.* The standard propositional model of the P/H principle Φ for $p = 3$ and $h = 2$ is shown in the left part. A graph interpretation over the explicit literal encoding of Φ with selected variables $\bar{\Gamma}_7$, $\bar{\Gamma}_8$, and $\bar{\Gamma}_9$ is shown in the right part together with its decomposition into complete sub-graphs (notice that the decomposition shown here can be found by the presented greedy algorithm – Algorithm 1).

Nevertheless, the main innovative reasoning mechanism uses the decomposition in a different way. At every point of the process there are still some candidate values/vertices for selection into the final supporting k-tuple. Each one is included in some decomposition sets from which no value/vertex has been selected yet. Let $\mathcal{L}$ be a set of such not yet used decomposition sets and let S be a set of already selected vertices. As only one value/vertex can be selected from each complete sub-graph we can make the following pruning: if it happens that $|\mathcal{L}| + |S| < k$, the search in the current branch of the support search tree can be terminated as it is not possible to extend the partial selection so that it will finally consist of k elements. This kind of reasoning is especially useful for problems with **non-local** properties such as the P/H principle or FPGA Switch-Box routing [1]. For illustration see Figure 2 (if $\bar{\beta}_{11}$ and $\bar{\beta}_{23}$ have been already selected, then $\mathcal{L} = \emptyset$, $k = 1$, and $S = \{\bar{\beta}_{11}, \bar{\beta}_{23}\}$ and hence we can conclude that $\bar{\beta}_{11}$ and $\bar{\beta}_{23}$ are inconsistent).

Algorithm 2. Search for a supporting k-tuple of values within B2C-consistency. It is supposed that a decomposition into complete sub-graphs $\mathcal{J}$ of a given graph interpretation $I(K_+^2)$ with respect to a $(k+2)$-list of variables K_+^2 has already been calculated.

```
function Check-B2C-Consistency($\bar{\psi}_{t_0}^{i_0}, \bar{\psi}_{t_{k+1}}^{i_{k+1}}, I(K_+^2) = (I_V, I_E), \mathcal{J}, \Lambda$): propositional
        /* Parameters: $\bar{\psi}_{t_0}^{i_0}, \bar{\psi}_{t_{k+1}}^{i_{k+1}}$   - a pair of values for consistency checking
                    $I(K_+^2)$          - a graph interpretation for decomposing,
                    $\mathcal{J}$                - a decomposition of $I(K_+^2)$ into
                                        complete sub-graphs,
                    $\Lambda$                - the number of allowed search steps. */
1:      $(\omega, \Lambda) \leftarrow$ Search-B2C-Support($\bar{\psi}_{t_0}^{i_0}, \bar{\psi}_{t_{k+1}}^{i_{k+1}}, \emptyset, I(K_+^2) = (I_V, I_E), \mathcal{J}, \Lambda$)
2:      return $\omega$

function Search-B2C-Support($\bar{\psi}_{t_0}^{i_0}, \bar{\psi}_{t_{k+1}}^{i_{k+1}}, S, I(K_+^2) = (I_V, I_E), \mathcal{J}, \Lambda$): pair
        /* Parameters: $S$          - a set of already selected supports. */
1:      if $|S| = k$ then return $(TRUE, \Lambda)$
2:      let $[\bar{\psi}_{t_1}^{i_1}, \bar{\psi}_{t_2}^{i_2}, \dots, \bar{\psi}_{t_l}^{i_l}] = S$
3:      for each $\bar{\psi}_{t_{l+1}}^{i_{l+1}} \in D(\bar{\Gamma}_{i_{l+1}})$ do
4:        if $\Lambda \leq 0$ then return $(TRUE, \Lambda)$ /* all the steps were consumed */
5:        $\alpha \leftarrow TRUE$
6:        for each $I_V \in \mathcal{J}$ do /* check of constraints */
7:            if $|I_V \cap (\cup [\bar{\psi}_{t_0}^{i_0}].S.[\bar{\psi}_{t_{l+1}}^{i_{l+1}}, \bar{\psi}_{t_{k+1}}^{i_{k+1}}])| > 1$ then $\alpha \leftarrow FALSE$
8:        let $\mathcal{L} = \{I_V \in \mathcal{J} | I_V \cap (\cup S.[\bar{\psi}_{t_{l+1}}^{i_{l+1}}]) = \emptyset\}$
9:        if $|\mathcal{L}| + |S| < k$ then $\alpha \leftarrow FALSE$ /* global check */
10:       if $\alpha$ then
11:           if $l + 1 < k$ then /* some supports still remain to be found */
12:               $(\omega, \Lambda) \leftarrow$ Search-B2C-Support($\bar{\psi}_{t_0}^{i_0}, \bar{\psi}_{t_{k+1}}^{i_{k+1}}, S.[\bar{\psi}_{t_{l+1}}^{i_{l+1}}]$,
13:                                                   $I(K_+^2), \mathcal{J}, \Lambda$)
14:               if $\omega$ then return $(TRUE, \Lambda)$
15:           else /* all the supports have been found */
16:               return $(TRUE, \Lambda)$
17:       $\Lambda \leftarrow \Lambda - 1$
18:     return $(FALSE, \Lambda)$
```

The process of B2C-consistency enfording for a pair of values and a fixed list of variables $K_+^2 = [\bar{\Gamma}_{i_0}, \bar{\Gamma}_{i_1}, \bar{\Gamma}_{i_2}, \dots, \bar{\Gamma}_{i_k}, \bar{\Gamma}_{i_{k+1}}]$ is shown as Algorithm 2. The algorithm searches for a supporting k-tuple of values for a given pair of values $\bar{\psi}_{t_0}^{i_0} \in D(\bar{\Gamma}_{i_0})$ and $\bar{\psi}_{t_{k+1}}^{i_{k+1}} \in D(\bar{\Gamma}_{i_{k+1}})$ in domains of $\bar{\Gamma}_{i_1}, \bar{\Gamma}_{i_2}, \dots, \bar{\Gamma}_{i_k}$. The search is done through a systematic extension of the current partial selection of supporting values/vertices. This functionality is implemented using recursive calls, which simulates chronological backtracking search.

The algorithm for enforcing B2C-consistency for a pair of values should be regarded as an incomplete proof of non-existence of a support. That is, if the algorithm finds the given pair of values to be inconsistent then there is actually no support for them (that is, it managed to prove that there is no support using Λ search steps and other techniques; $FALSE$ is returned by *Check-B2C-Consistency* in this case). However, if it does not find the given pair of values to be inconsistent, one of the following cases might happen: a supporting k-tuple of values was found or the algorithm ran out of the allowed number of search steps Λ ($TRUE$ is returned in this case).

Proposition 2 (B2C Time/Space Complexity). If $\Lambda = \infty$ then the algorithm for enforcing B2C-consistency with a decomposition into complete sub-graphs $\mathcal{J}$ of a graph interpretation $I(K_+^2) = (I_V, I_E)$ of a $(k+2)$-list of variables K_+^2 can be implemented to have the worst case time complexity of $\mathcal{O}(k|\mathcal{J}||\mathbb{D}|^{\mathrm{k}})$; otherwise, the worst case time complexity is $\mathcal{O}(|\mathcal{J}|\Lambda)$. The corresponding worst case space complexity is $\mathcal{O}(|I_V| + |I_E|)$. ■

Commentary: It is not difficult to observe that the algorithm needs to go through all the $|\mathbb{D}|^k$ k-tuples in the worst case if the number of the allowed search steps Λ is unbounded. Checking a k-tuple may consume up to $k|\mathcal{J}|$ constraint checks (namely checks against complete sub-graphs). If Λ is bounded then obviously at most Λ steps are done while each step consumes up to $|\mathcal{J}|$ constraint checks.

As all the data elements are accessed sequentially no extra data structures are needed. Hence, we need to store graph interpretation and its decomposition into complete sub-graphs, which we already know to be of $\mathcal{O}(|I_V| + |I_E|)$. The space needed to store the resulting k-tuple is again of $\mathcal{O}(|I_V| + |I_E|)$. ■

Here it depends on our perception of k. It is natural to perceive it as a part of the input and hence the complexity of search for a support is exponential with unbounded Λ. Therefore, the time consumption represents the main bottleneck of the method. However, having the global reasoning based on complete sub-graphs, still much can be done in Λ steps while Λ is bounded.

4. Building a SAT Preprocessing Tool

We intended to use B2C-consistency as a basis for a SAT preprocessing tool. As we have seen, it may not be simply used for that task in its raw form due to its time complexity. A good compromise between the computational effort and strength of the inference has to be found. This section describes how a list of variables should be chosen and how to set particular parameters of B2C-consistency to make it suitable for the intended preprocessing tool.

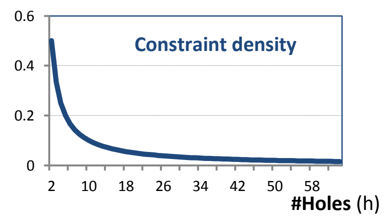
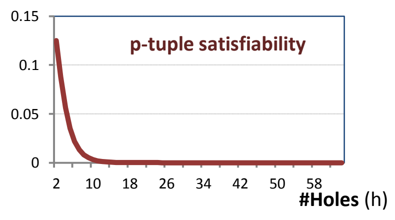
4.1. *Selection of k-tuples of CSP Variables*

As it is computationally infeasible to achieve B2C-consistency with respect to all the k-tuples of variables and pairs of values in their domains in a non-trivially large SAT instance, some selection of promising subsets of variables on which the consistency will be applied has to be done. The selection is considered to be promising if there is a chance that the consistency rules out some pair of values (that is, the ruled out pair of values cannot be a part of any solution). At the same time, the information captured in the fact that a given pair of values is incompatible should be valuable for a SAT solver in a certain sense. This requirement is imposed by the intention to use B2C-consistency as a preprocessing tool. Hence, the information should not be easily derivable by the SAT solver itself since informing the SAT solver about the inconsistency between a pair of trivially incompatible values is not helpful.

Our approach is to select k-tuples of variables induced by a region of the instance with difficult constraint setup that however can be tackled by the consistency. Such a setup provides a chance to extract valuable information by B2C-consistency. The well known SAT model of the *pigeon hole principle* (P/H principle) – more precisely its explicit literal encoding – is a representative of such a setup which is well known for resisting from being handled by SAT solvers [1]. All the instances of the P/H principle are unsatisfiable. Having a suitable graph interpretation for the P/H principle as it is shown in Figure 2 (that is, clauses modeling that each pigeon is placed in some hole are selected as a $(k+2)$-tuple of CSP variables for the graph interpretation) we are able to calculate various useful probabilistic characteristics.

Let p be the number of pigeons and let h be the number of holes where it holds that $h = p - 1$. A *constraint tightness* ρ in a binary CSP will be defined as the ratio of the number of allowed pairs of values to the number of all the possible pairs of values. Particularly in the case of the P/H principle it holds that $\rho = \frac{\binom{p}{2}h}{\binom{p}{2}h^2} = \frac{1}{h}$ in the graph interpretation as described above.

Table 1. Probabilistic characteristics of the graph interpretation in the P/H principle.

Configuration: **pigeons** (p) $\times$ **holes** ($h = p - 1$)	Constraint **tightness ρ** $\frac{\binom{p}{2}h}{\binom{p}{2}h^2} = \frac{1}{h}$	Probability of **satisfiability** of a random p-tuple $\boldsymbol{\sigma}$ $(1-\frac{1}{h})^{\binom{h+1}{2}}$	**Expected number** of satisfied p-tuples ε $h^{h+1}(1-\frac{1}{h})^{\binom{h+1}{2}}$	 
3×2	0.5	0.125	1	
4×3	0.333333	0.087791	7.111111	
5×4	0.25	0.056314	57.66504	
6×5	0.2	0.035184	549.7558	
7×6	0.166667	0.021737	6084.888	
8×7	0.142857	0.01335	76961.62	

Another interesting characteristic is the probability of a randomly selected assignment of values to p variables σ calculated from the constraint tightness. It is a reasonable assumption that the satisfaction of individual pairs of values within the assignment is independent of each other. Then it holds for the *probability of satisfaction* of a random p-tuple of values that $\sigma = (1-\rho)^{\binom{p}{2}}$ which is $(1-\frac{1}{h})^{\frac{(h+1)h}{2}}$ in the case of the P/H principle.

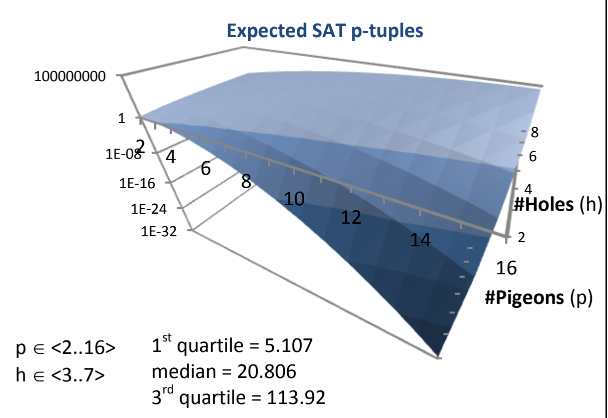
Finally, we will investigate the expected number of satisfied p-tuples of values ε, which will be defined as the total number of possible p-tuples multiplied by ρ. It holds that $\varepsilon = h^p\sigma = h^{h+1}(1-\frac{1}{h})^{\frac{(h+1)h}{2}}$ in the case of the P/H principle. Several examples of probabilistic characteristics are shown in Table 1. The limit behavior of the above characteristics with $h \to \infty$ is summarized in the following easy-to-prove proposition.

Proposition 3 (Limit P/H Characteristics). The probability of satisfiability of a random p-tuple of values ρ in a graph interpretation of the P/H principle converges to 0 for h → ∞; that is, $\lim_{h\to\infty}(1-\frac{1}{h})^{\binom{h+1}{2}} = 0$. The expected number of satisfied p-tuples of values ε in the P/H principle is $\mathcal{O}(e^{(h+1)(-\frac{1}{2}+\ln h)})$ which is $\mathcal{O}\left(\left(\frac{h}{\sqrt{e}}\right)^{(h+1)}\right)$ and blows up to $+\infty$ for $h \to \infty$; that is, $\lim_{h\to\infty} h^{h+1}(1-\frac{1}{h})^{\binom{h+1}{2}} = +\infty$. ■

We will generalize the P/H principle so that there will be strictly less holes than pigeons but not necessarily one fewer. The generalized P/H principle is unsatisfiable as well. A sample of probabilistic characteristics of the model of the generalized P/H principle is shown in Table 2.

Table 2. Expected number of satisfied tuples of values in the generalized P/H principle.

	Expected number of satisfied p-tuples $\varepsilon = h^p(1-\frac{1}{h})^{\binom{p}{2}}$		
	Number of **holes** h		
Number of **pigeons** p	***3***	***4***	***5***
2	6.0	12.0	20.0
3	8.0	27.0	64.0
4	7.111	45.563	163.84
5	4.213	57.665	335.544
6	1.664	54.737	549.756
7	0.438	38.968	720.576



Our aim is to select $(k+2)$-tuples of CSP variables for B2C-consistency in the explicit literal encoding $E^1(\Phi) = (X_\Phi^1, D_\Phi^1, C_\Phi^1)$ which has similar probabilistic characteristics that are exhibited by the model of the (generalized) P/H principle. This selection is supposed to ensure the required properties – that is, a similar level of difficulty as the P/H principle and the similar constraint setup. The following incremental mechanism for selecting the next variable based on estimating probabilistic characteristics from the currently selected variables will be used.

The requirement which is specified as a part of the input together with k is the interval for the expected number of satisfied $(k+2)$-tuples of values. Let ε_L and ε_U be the lower and upper bound for this interval respectively. The first CSP variable into the $(k+2)$-tuple is supposed to be selected using some specific process (randomly or systematically; actually a systematic process is used within the experimental implementation). Other CSP variables are selected incrementally; suppose that $K_+^2 = [\bar{\Gamma}_{i_0}, \bar{\Gamma}_{i_1}, \bar{\Gamma}_{i_2}, \dots, \bar{\Gamma}_{i_\kappa}]$ is a tuple of the already selected CSP variables (if $\kappa = k$ then the process is finished). Let $\bar{\Gamma}_{i_{\kappa+1}}$ be a candidate CSP variable.

Algorithm 3. *Process of selecting a suitable $(k+2)$-tuple of CSP variables.* Variables are heuristically selected to prefer the resulting expected number of satisfied $(k+2)$-tuples of values in the interval of $\langle \varepsilon_L, \varepsilon_U \rangle$ or near this interval from below or above.

```
function Select-CSP-Variables$(k, \bar{\Gamma}_{i_0}, E^1(\Phi) = (X_\Phi^1, D_\Phi^1, C_\Phi^1), \varepsilon_L, \varepsilon_U)$: tuple
    /* Parameters: $k$            - size of the tuple of CSP variables,
                   $\bar{\Gamma}_{i_0}$          - the first CSP variable,
                   $E^1(\Phi)$     - explicit literal encoding,
                   $\varepsilon_L, \varepsilon_U$       - lower and upper bounds for the expected number of
                                  satisfied $(k+2)$-tuples of values. */
1:for $\kappa = 0,1,\dots,k$ do
2:      for each $\bar{\Gamma}_{i_{\kappa+1}} \in X_\Phi^1$ do
3:          let $\rho(\bar{\Gamma}_{i_{\kappa+1}})$ is the constraint tightness in $\bigcup[\bar{\Gamma}_{i_0}, \bar{\Gamma}_{i_1}, \bar{\Gamma}_{i_2}, \dots, \bar{\Gamma}_{i_{\kappa+1}}]$
4:          $\varepsilon(\bar{\Gamma}_{i_{\kappa+1}}) \leftarrow \left(\sqrt[\kappa+1]{\prod_{l=0}^{\kappa+1} |D(\bar{\Gamma}_{i_l})|}\right)^{k+2} \left(1 - \rho(\bar{\Gamma}_{i_{\kappa+1}})\right)^{\binom{k+2}{2}}$
      /* the following let form assigns $\perp$ if undefined */
5:      let $\bar{\Gamma}_\alpha \in X_\Phi^1$ such that $\varepsilon(\bar{\Gamma}_\alpha) \leq \varepsilon_L \wedge (\forall \bar{\Gamma}_l \in X_\Phi^1) \varepsilon(\bar{\Gamma}_\alpha) \geq \varepsilon(\bar{\Gamma}_l)$
6:      let $\bar{\Gamma}_\beta \in X_\Phi^1$ such that $\varepsilon(\bar{\Gamma}_\beta) \geq \varepsilon_U \wedge (\forall \bar{\Gamma}_l \in X_\Phi^1) \varepsilon(\bar{\Gamma}_\beta) \leq \varepsilon(\bar{\Gamma}_l)$
7:      let $\bar{\Gamma}_\mu \in X_\Phi^1$ such that $\varepsilon_L \leq \varepsilon(\bar{\Gamma}_\mu) \leq \varepsilon_U$
8:      if $\bar{\Gamma}_\mu \neq \perp$ then
9:              $i_\kappa \leftarrow \mu$
10:         else
11:             if $\bar{\Gamma}_\alpha = \perp$ then
12:                 $i_\kappa \leftarrow \beta$
13:             else
14:                 if $|\varepsilon(\bar{\Gamma}_\alpha) - \varepsilon_L| < |\varepsilon(\bar{\Gamma}_\beta) - \varepsilon_U|$ then
15:                     $i_\kappa \leftarrow \alpha$
16:                 else
17:                     $i_\kappa \leftarrow \beta$
18:     return $[\bar{\Gamma}_{i_0}, \bar{\Gamma}_{i_1}, \bar{\Gamma}_{i_2}, \dots, \bar{\Gamma}_{i_\kappa}]$
```

The expected number of the satisfied $(k+2)$-tuples with $\bar{\Gamma}_{i_{\kappa+1}}$ denoted as $\varepsilon(\bar{\Gamma}_{i_{\kappa+1}})$ is estimated as follows: let $\rho(\bar{\Gamma}_{i_{\kappa+1}})$ be the constraint tightness among variables from the set $\bigcup K_+^2 \cup \{\bar{\Gamma}_{i_{\kappa+1}}\}$ (already selected variables together with the new candidate) then $\varepsilon(\bar{\Gamma}_{i_{\kappa+1}}) = \left(\sqrt[\kappa+1]{\prod_{l=0}^{\kappa+1} |D(\bar{\Gamma}_{i_l})|}\right)^{k+2} \left(1 - \rho(\bar{\Gamma}_{i_{\kappa+1}})\right)^{\binom{k+2}{2}}$. That is, the product of sizes of domains of the final $(k+2)$-tuple is estimated as $(k+2)$th power of the geometric mean of sizes of the domain of already selected variables. The constraint tightness is supposed to be preserved for the final $(k+2)$-tuple. If $\varepsilon_L \leq \varepsilon(\bar{\Gamma}_{i_{\kappa+1}}) \leq \varepsilon_U$ then $\bar{\Gamma}_{i_{\kappa+1}}$ may be used as the next CSP variable for the $(k+2)$-tuple. If there are multiple variables satisfying this condition any of them may be selected (in the implementation that one with ε closest to $\frac{\varepsilon_L+\varepsilon_U}{2}$ is selected). The whole process of selection of CSP variables for B2C-consistency is formalized as Algorithm 3.

Proposition 4 (Selection Time/Space Complexity). The algorithm for selecting CSP variables can be implemented to have the worst case time complexity of $\mathcal{O}(k^2|X_\Phi^1||\mathbb{D}|^2)$. A space of $\mathcal{O}(k+|X_\Phi^1|)$ is needed in addition to the space necessary for storing CSP $E^1(\Phi)$. ■

Commentary: Each new CSP variable is selected for the resulting tuple out of at most $|X_\Phi^1|$ CSP variables for which estimation of the expected number of satisfied $(k+2)$-tuples must be calculated. Calculating this estimation with respect to a single variable consumes $\mathcal{O}(k|\mathbb{D}|^2)$ steps as it is necessary to calculate constraint tightness relatively to all the already selected variables. A new variable is included exactly k times.

Additional space is needed for storing probabilistic characteristics for CSP variables, which consumes the space of $\mathcal{O}(|X_\Phi^1|)$. A space of $\mathcal{O}(k)$ is needed to store the resulting tuple of CSP variables. ■

It is infeasible in large SAT instances to compute and to store constraint tightness between all the pairs of variables on the current commodity hardware because there are too many such pairs (notice that there may be more than $1.0\mathrm{E}+6$ clauses in large SAT instances which makes more than $\binom{1.0\mathrm{E}+6}{2} \approx 1.0\mathrm{E}+12$ pairs of variables; that would require approximately several terabytes of memory). Hence, it is necessary to compute constraint tightness on demand.

4.2. *SAT Preprocessing with B2C-Consistency*

An experimental SAT preprocessing tool based on B2C-consistency called `preprocessSIGMA` [27] was implemented in C++ in order to conduct an experimental evaluation and to provide proof of the concept. To achieve the best inference strength of preprocessing, $(k+2)$-tuples are selected according to the theory in the previous section so that the expected number of satisfied tuples of values belongs into the interval typical for the model of the generalized P/H principle. We select k uniformly from the interval $\langle 2..10\rangle$ as it experimentally proved to be computationally manageable in reasonable time.

In typical SAT instances arity of clauses ranges from 2 to 10 [18] while the most common are small clauses with arities 3, 4, and 5 – domain sizes in the corresponding literal encoding are exactly the same. The expected number of satisfied tuples of values for a setup of the P/H principle with corresponding $p \in \langle 2..10\rangle$ and $h \in \langle 3..5\rangle$ belongs into the interval $\langle 0.001,755.579\rangle$ while the 1st quartile, median, and 3rd quartile are equal to 5.107, 20.806, 113.92, respectively. Taking into account that we are preferring the non-existence of satisfied tuple of values, it is advisable to select the preferred interval for the expected number of satisfied tuples of values $\langle\varepsilon_L,\varepsilon_U\rangle$ with ε_L low below the median and slightly below the 1st quartile and ε_U slightly above the median. A preliminary experimental evaluation with SAT instances containing mainly small clauses showed that the best setting is $\langle\varepsilon_L,\varepsilon_U\rangle = \langle 3.0,32.0\rangle$ which well correlates with the above probabilistic estimations. The use of different bounds resulted in deriving less valuable forbidden pairs of values in the preprocessing step (that is, explicit forbidding of such pairs by adding new clauses had a limited positive effect).

As the computation of B2C-consistency is a time consuming operation it is done only for a certain number of tuples of variables. More precisely, small formulae with less than or equal to 2048 variables are allowed 16 times the number variables B2C-consistency checks. Large formulae (that is, those with more than 2048 variables) are allowed 4 times square root of the number of variables B2C-consistency checks (currently, there is no smooth transition between these two rates as it was not necessary to be implemented for experimental evaluation). In both cases, the number of steps of the search for a consistent k-tuple was bounded by the constant $\Lambda = 4096$. This setup of Λ was manually tailored during the development of the method.

We are aware that the presence of several parameters in the method may be problematic since the user is required to set them. However, in our analysis we provide some ideas for their setting and, most importantly, the parameters can be regarded as an opportunity for further optimization by methods for automated parameter tuning (*programming by optimization*) [17].

5. Experimental Evaluation

The experimental evaluation of our prototype SAT preprocessor `preprocessSIGMA` focused on discovering the benefit of B2C-consistency in the context of other existent preprocessing techniques and on the evaluation of internal properties of the experimental implementation. It also should provide a justification for the theory we have discussed earlier.

5.1. *Basic Competitive Experimental Evaluation*

The experimental implementation of B2C-consistency within our prototype tool `preprocessSIGMA` has been competitively evaluated with respect to the most prominent existing tools for SAT preprocessing. Particularly, the following preprocessing tools have been evaluated: `LiVer` [25], `NiVer` [25], `HyPre` [5], `Shatter` with `Saucy` version `3.0` [2, 21] (here abbreviated as `saucy-3`), and the technique of *blocked clause elimination* [16, 20] (here abbreviated as `BCE`) implemented within `precosat-465` [20] (here abbreviated as `BCE`). As the reference SAT solver `MiniSAT` version `2.2` [13] with an built-in `SatElite` preprocessing step has been used.

`LiVer` and `NiVer` use resolution-based variable elimination for preprocessing; `LiVer` allows a bounded increase in the total number of literals in the resulting formula while `NiVer` does not allow any increase in this number. The `HyPre` preprocessing tool is based on binary hyper-resolution and equivalence reasoning. `Shatter` represents a tool most akin to our `preprocessSIGMA` as it employs a certain kind of global reasoning as well. Symmetries in the input formula are detected and symmetry-breaking clauses are added by `Shatter` into the output formula. To detect symmetries, the *graph isomorphism* problem [28] needs to be solved during the preprocessing process which is done by the `Saucy` module. The performance of the `Saucy` module is crucial in `Shatter`.

Table 3. *Listing of results for a fraction of the set of testing instances used in our experimental evaluation.* The number of conflicts `MiniSAT 2.2` encountered on the original instances and on those preprocessed by `HyPre`, `LiVer`, `NiVer`, `saucy-3`, and our `preprocessSIGMA` are shown. The best performing preprocessors on each instance are marked in bold (the timeout for both preprocessing and `MiniSAT` was set to 256 seconds). Observe the large differences among individual preprocessors.

Conflicts	*Variables*	*Clauses*	*C/V Ratio*	*Original*	*HyPre*	*BCE*	*LiVer*	*NiVer*	*saucy-3*	*sigma*	*SAT/UNSAT*
bart12.shuffled	*180*	*820*	4.555	**105**	212	**105**	118	118	603	**105**	SAT
bart14.shuffled	*195*	*905*	4.641	104	402	104	**100**	**100**	102	104	SAT
bart16.shuffled	*210*	*990*	4.714	**103**	106	**103**	**103**	**103**	215	**103**	SAT
bart20.shuffled	*270*	*1476*	5.466	121	127	206	**103**	**103**	160	121	SAT
ca004.shuffled	*80*	*168*	2.1	43	**29**	48	32	**29**	42	43	UNSAT
ca008.shuffled	*130*	*370*	2.846	145	117	175	**102**	150	151	145	UNSAT
ca016.shuffled	*272*	*780*	2.867	449	**293**	465	416	326	357	433	UNSAT
ca032.shuffled	*558*	*1606*	2.878	943	752	1103	739	**657**	901	790	UNSAT
difp_19_99_arr_rcr	*1201*	*6563*	5.464	209417	141649	343814	58305	304092	209417	**92754**	SAT
difp_19_99_wal_rcr	*1775*	*10446*	5.885	134284	**31031**	92343	108681	158235	N/A	15245	SAT
difp_21_1_arr_rcr	*1453*	*7967*	5.483	191884	63546	126655	538426	427292	191884	**45453**	SAT
difp_21_99_arr_rcr	*1453*	*7967*	5.483	190663	97408	66191	249983	350142	190663	**35704**	SAT
dp04u03.shuffled	*1017*	*2411*	2.370	70	**26**	77	72	63	N/A	61	UNSAT
dp05s05.shuffled	*1885*	*4818*	2.555	90	138	80	116	100	N/A	**46**	SAT
ezfact32_6.shuffled	*769*	*4777*	6.211	422	33088	**169**	32957	32957	422	209	SAT
ezfact32_7.shuffled	*769*	*4777*	6.211	5744	29574	**173**	46659	46659	5744	836	SAT
ezfact32_9.shuffled	*769*	*4777*	6.211	1181	47191	218	64056	64056	1181	**160**	SAT
ezfact32_10.shuffled	*769*	*4777*	6.211	1990	1988	**406**	22500	22500	1990	448	SAT
fpga10_11_uns_rcr	*220*	*1122*	5.1	4935017	8315862	4935017	4866421	4866421	548002	**2**	UNSAT
fpga10_12_uns_rcr	*240*	*1344*	5.6	7209341	7219129	7140410	7183640	7218248	645603	**1**	UNSAT
fpga10_13_uns_rcr	*260*	*1586*	6.1	6466487	6511919	5963904	6497147	6497268	264637	**1**	UNSAT
fpga10_15_uns_rcr	*300*	*2130*	7.1	5390760	5401469	5361172	5387934	5405715	91837	**1**	UNSAT
fpga10_8_sat	*120*	*448*	3.733	201	163	201	201	201	**65**	201	SAT
fpga10_9_sat	*135*	*549*	4.066	202	168	202	202	202	**100**	202	SAT
fpga12_11_sat	*198*	*968*	4.888	200	405	200	200	200	**54**	200	SAT
fpga12_12_sat	*216*	*1128*	5.222	208	102	208	208	208	**36**	208	SAT
homer06.shuffled	*180*	*830*	4.611	272019	209811	272019	258487	258487	39341	**1**	UNSAT
homer10.shuffled	*360*	*3460*	9.611	641132	502279	641132	464639	464639	144	**2**	UNSAT
homer16.shuffled	*264*	*1476*	5.590	6525641	6527180	6484372	6766937	6682636	3195152	**3**	UNSAT
homer20.shuffled	*440*	*4220*	9.590	3230156	3249156	3043099	3265756	3207038	87585	**2**	UNSAT
lisa19_0_a.shuffled	*1201*	*6563*	5.464	235824	117828	209804	381242	108878	235824	**15534**	SAT
lisa19_1_a.shuffled	*1201*	*6563*	5.464	445563	439709	688143	**208567**	528589	445563	320076	SAT
lisa21_1_a.shuffled	*1453*	*7967*	5.483	328846	121498	67873	**4841**	309122	328846	93629	SAT
med11.shuffled	*341*	*5556*	16.293	**41**	197	102	101	101	**41**	**41**	SAT
med17.shuffled	*782*	*18616*	23.805	**106**	151	4599	808	808	**106**	**106**	SAT
qg1-7.shuffled	*686*	*6816*	9.935	**49**	115	**44**	67	67	242	49	SAT
term1_gr_2pin_w3.shuffled	*746*	*3517*	4.714	52	69	27	21	124	52	**9**	UNSAT
term1_gr_rcs_w3.shuffled	*606*	*2518*	4.155	7	7	7	7	7	11	**1**	UNSAT

The experimental evaluation was done with a set of 344 difficult SAT instances (mixture of satisfiable and unsatisfiable) taken from the *Satisfiability Library* (SATLib – only structured instances have been taken) [18] and from the crafted category of the *SAT Competitions 2002/2003* and *2007/2009* (all the problems from the crafted category of a size up to 600kB have been taken). The complete set of instances used in the experimental evaluation can be found at the website: http://ktiml.mff.cuni.cz/

~surynek/research/j-preprocess-2011. This website also contains experimental data in the raw form and the complete source code in C++ necessary to reproduce all the presented experiments.

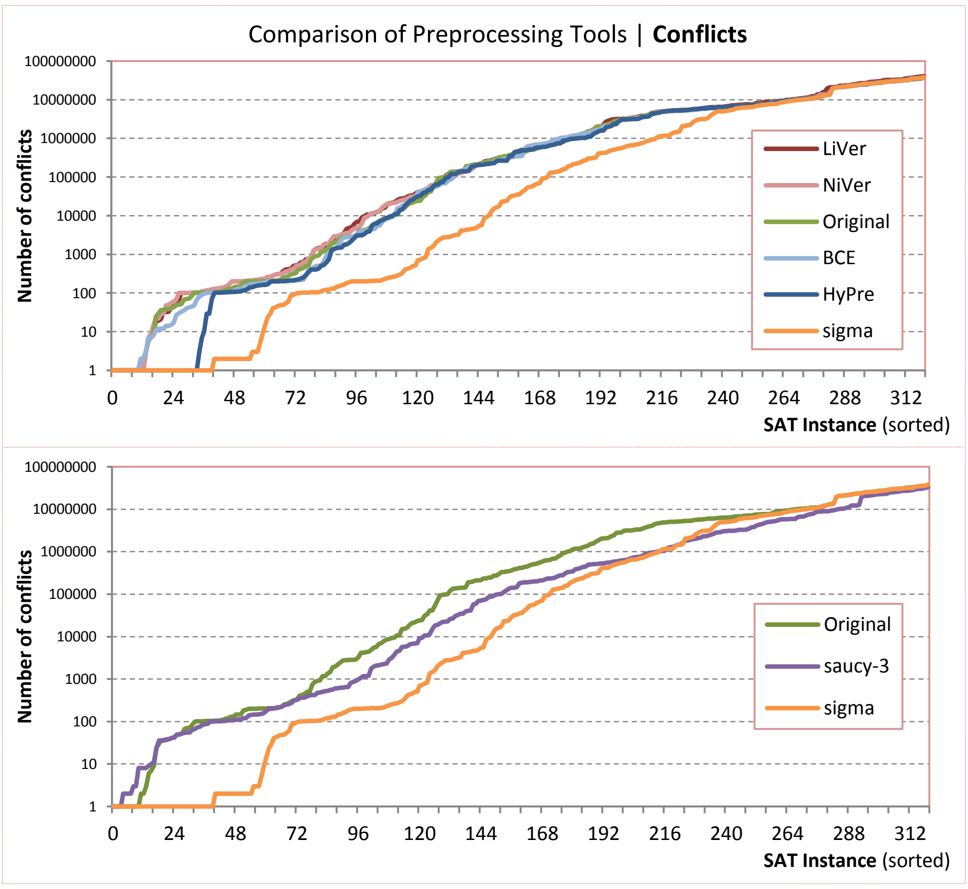


Figure 3. *Competitive comparison of preprocessing tools (conflicts).* The number of conflicts that occurred when solving the original and preprocessed SAT instances by `MiniSAT 2.2` are shown (instances are sorted for each preprocessor to get increasing sequences – easier instances tend to be on the left while hard instances tend to be on the right). Our `preprocessSIGMA` is compared with `HyPre`, `LiVer`, `NiVer`, `BCE`, and `saucy-3` on a set of SAT instances from SATLib and SAT Competitions 2003/2004 and 2007/2009. It can be observed that `HyPre`, `LiVer`, `NiVer`, and `BCE` have only marginal effect (upper part) compared to `preprocessSIGMA` and `saucy-3` (lower part) which both deliver significant improvements. If timeout was reached the instance was excluded from the figure.

Several characteristics were measured during the evaluation process. The most informative characteristic is the number of *conflicts* that occurred during the process of solving. The conflicts can be regarded as a dead-end in the backtracking-based search process. The number of conflicts has been measured for the original instances and for instances processed by individual SAT preprocessors from our test suite. The number of conflicts corresponds well with the overall runtime. The CPU time[*] has been measured as well to obtain the complete picture of performance of all the SAT preprocessors.

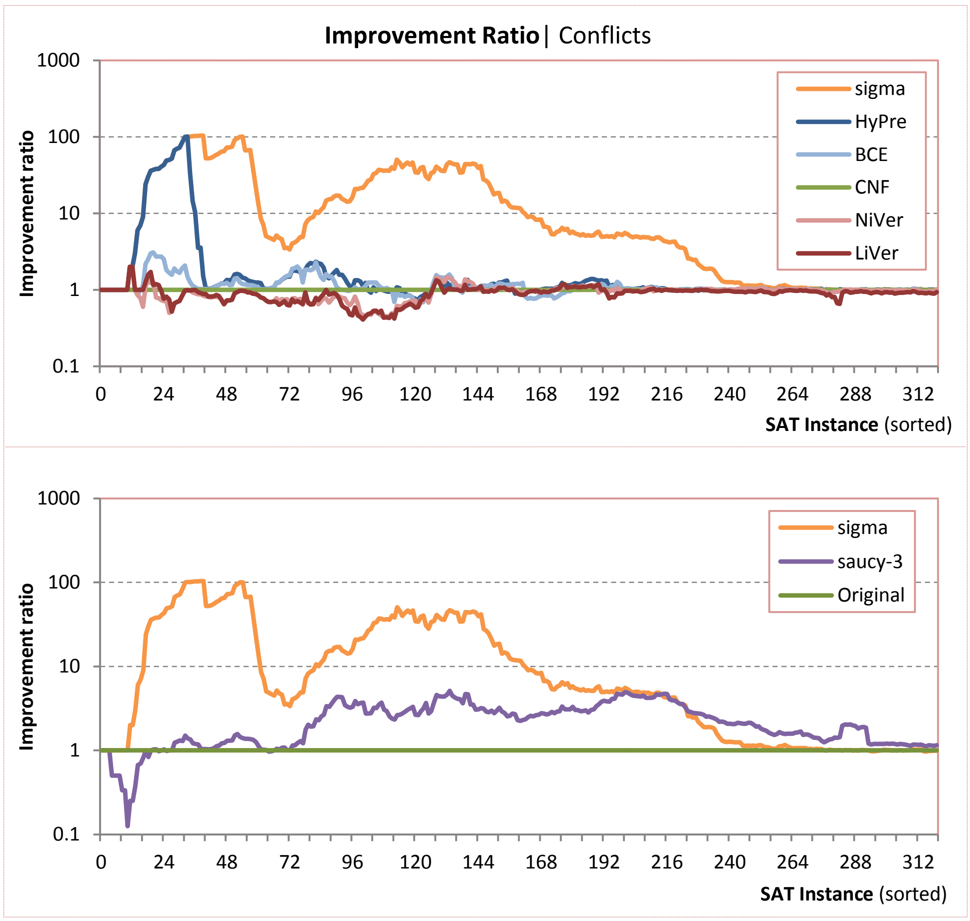


Figure 4. *Improvement ratio in the number of conflicts gained by the application of preprocessor.* The ordering of instances per preprocessor is the same as in Figure 3. It can be observed that `NiVer` and `LiVer` cause worsening in a significant number of instances. `HyPre` is particularly successful on easier instances. The best improvement can be achieved by `saucy-3` and `preprocessSIGMA` while `saucy-3` has an advantage in large instances and `preprocessSIGMA` dominates in easier nes.

A small fraction of the set of instances (38 out of 344) used in the experimental evaluation together with their characteristics and results regarding the number of conflicts after preprocessing is shown in Table 3. In all the tests presented in this paper, preprocessing and solving by `MiniSAT` was run for at most 256 seconds of CPU time; that is, the total runtime per instance is limited to 512 seconds of CPU time. The full competitive

[*] All the tests were run on a machine with Intel Xeon 2.0GHz CPU, 12 GB of RAM, under Ubuntu Linux version 8.04, Kernel 2.6.24-19 SMP.

comparison of the number of conflicts that `MiniSAT` encountered when solving the original instances and preprocessed ones is shown in Figure 3.

There are large improvements and large differences among individual SAT preprocessors observable in Table 3 and Figure 3. Hence, preprocessing seems to be a powerful tool and the choice of the right preprocessor is a crucial decision point with an important performance impact.

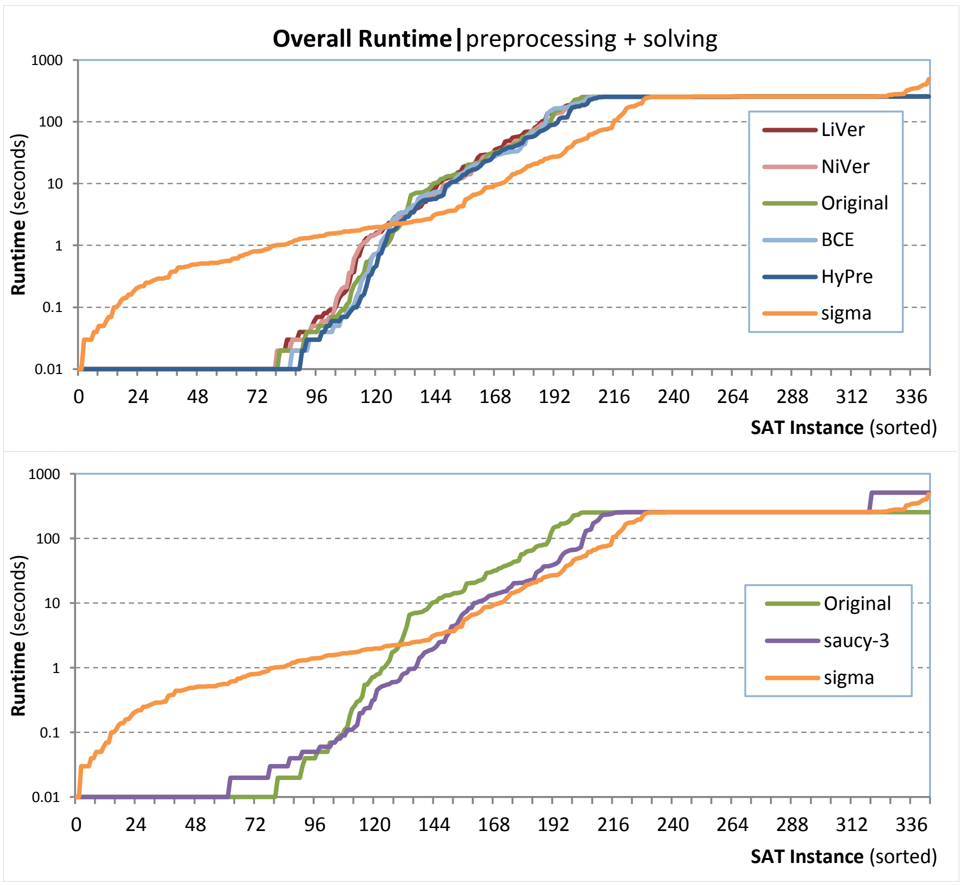


Figure 5. *Competitive comparison of preprocessing tools according to runtime*[†]. The runtime of preprocessing plus the runtime of solving preprocessed instances are shown. Instances are sorted according to the increasing runtime (each preprocessor has its own sorting of instances). The advantage of `preprocessSIGMA` and `saucy-3` is slightly reduced as they both require longer runtime for preprocessing than other preprocessors. However, they are still dominant on medium hard to very hard instances. On easier instances `saucy-3` prevails over `preprocessSIGMA` but the difference is narrowing towards harder instances where `saucy-3` often did not finish in the given timeout and `preprocessSIGMA` became better option.

[†] Notice there are more instances in runtime figures. This is due to the fact that instances where preprocessing did not finish are included in runtime figures but they are not included in figures regarding conflicts.

The evaluation implies that preprocessors solely relying on simplification through local inferences such as resolution, hyper-resolution, and blocked clause elimination – that is `HyPre`, `LiVer`, `Niver`, and `BCE` – deliver almost no improvement on the evaluated set of difficult SAT instances (even worsening in a significant number of instances appeared). These results indicate that preprocessing employing local inference rules only is unable to discover and exploit a higher level structure encoded in the instance.

On the other hand, `saucy-3` as well as `preprocessSIGMA`, which both employ global reasoning, deliver significant improvements in terms of the number of conflicts on preprocessed instances. Hence, global reasoning seems to be beneficial in instances encoding a certain kind of a high level structure.

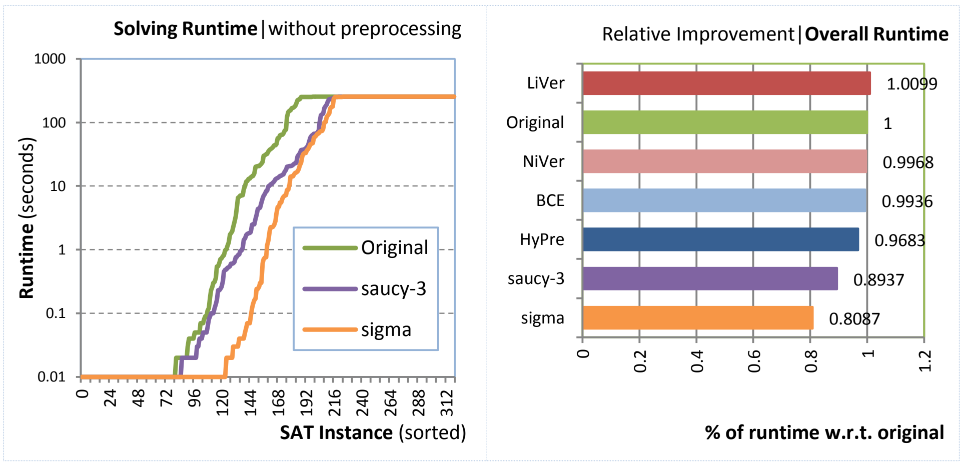


Figure 6. *Additional results regarding runtime – runtime without preprocessing/aggregated improvement.*
Left: If merely the SAT solving runtime is accounted then `preprocessSIGMA` delivers better performance in easier to moderately difficult instances than `saucy-3`. In difficult instances the performances of `preprocessSIGMA` and `saucy-3` are matched.
Right: The improvement of the overall solving time over the whole evaluated set of instances. Only preprocessors based on global reasoning – `saucy-3` and `preprocessSIGMA` – deliver considerable improvement. Approximately 20% of the original runtime is saved in the case of `preprocessSIGMA`.

In instances of easy to medium difficulty, `preprocessSIGMA` delivers a better positive effect in preprocessing than `saucy-3` – up to 100 times less conflicts are encountered in instances preprocessed with `preprocessSIGMA` than in the original ones. The difference between `preprocessSIGMA` and `saucy-3` diminishes in instances of top difficulty (`saucy-3` becomes marginally better in several instances).

The results, however, should not be interpreted as that preprocessing by resolution/hyper-resolution is useless. In simpler instances it is typically more beneficial [4] if we take into account a tradeoff between the benefit and computational costs. Moreover, we need to consider that the version of `MiniSAT` we used has its own built-in preprocessor `SatElite`. The results may thus show that simple resolution-based preprocessing is not enough to outperform the benefit of the use of `SatElite` (although this claim may require further investigation).

An experimental evaluation regarding the runtime is shown in Figure 5 and Figure 6. It can be observed that if merely solving runtime is measured, then the picture is almost the same is in the case of conflicts – preprocessSIGMA and saucy-3 clearly outperforms the others (BCE, HyPre, LiVer, and Niver). The situation changes if the time for preprocessing is accounted (that is, total runtime = preprocessing runtime + solving runtime is taken into account). Here preprocessSIGMA starts lagging behind all others in easier instances due to its long runtime.

A similar phenomenon but not that profound can be observed for saucy-3, which loses against BCE, HyPre, LiVer, and NiVer in easier instances. The situation changes in more difficult instances where saucy-3 and preprocessSIGMA perform better than others. Even preprocessSIGMA matches saucy-3 on yet more difficult instances.

If the total runtime for the whole testing suite is considered, we get an interesting comparison: both saucy-3 and preprocessSIGMA save up to 20% of the total runtime compared to the situation without preprocessing while the other tools (BCE, HyPre, LiVer, and Niver) provide no or marginal improvement only.

Note that the match in overall runtime with saucy-3 in more difficult instances has been achieved despite the not well optimized implementation of preprocessSIGMA (this is also the reason why we need to limit the size of the tested instances). Regarding the preprocessing time with preprocessSIGMA there is a great potential for further improvement.

5.2. *B2C-Consistency on Integer Factorization*

An especially good performance was exhibited by our preprocessing tool based on B2C-consistency in instances encoding *integer factorization problem* [3] (satisfiable instances). The first observation made in these instances is that B2C-consistency is able to make many inferences of inconsistent pairs of values that can be ruled out in the preprocessed instance afterwards.

An additional experimental evaluation showed that the more inconsistent pairs of values are inferred, the greater the reduction of the number of conflicts (as well as runtime) can be achieved on the resulting instance. However, this property contradicts the requirement of bounding the number of B2C-consistency checks which is needed to be low to preserve reasonable time consumption (if we want to infer as many inconsistent pairs of values as possible we should perform as many consistency checks as possible). Hence, there is still room for improvement on integer factorization problems using fine tuning of the parameters of B2C-consistency such as the allowed number of constraint checks.

The competitive results regarding the integer factorization problem are shown in Figure 7. Clearly, preprocessSIGMA is the best for almost all the instances in terms of the number of conflicts it can save. Surprisingly, saucy-3 did not finish preprocessing for approximately half of the instances in the given timeout of 256 seconds. Regarding relative improvement, it rarely happens that the tested preprocessors cause worsening (only LiVer and NiVer exhibited this behavior marginally).

If we look at the overall runtime, saucy-3 loses due to its frequent depleting the timeout. Another observation is that accounting preprocessing time does not change the picture of relative performance so much as the solving time for the instances is quite long compared to the preprocessing time.

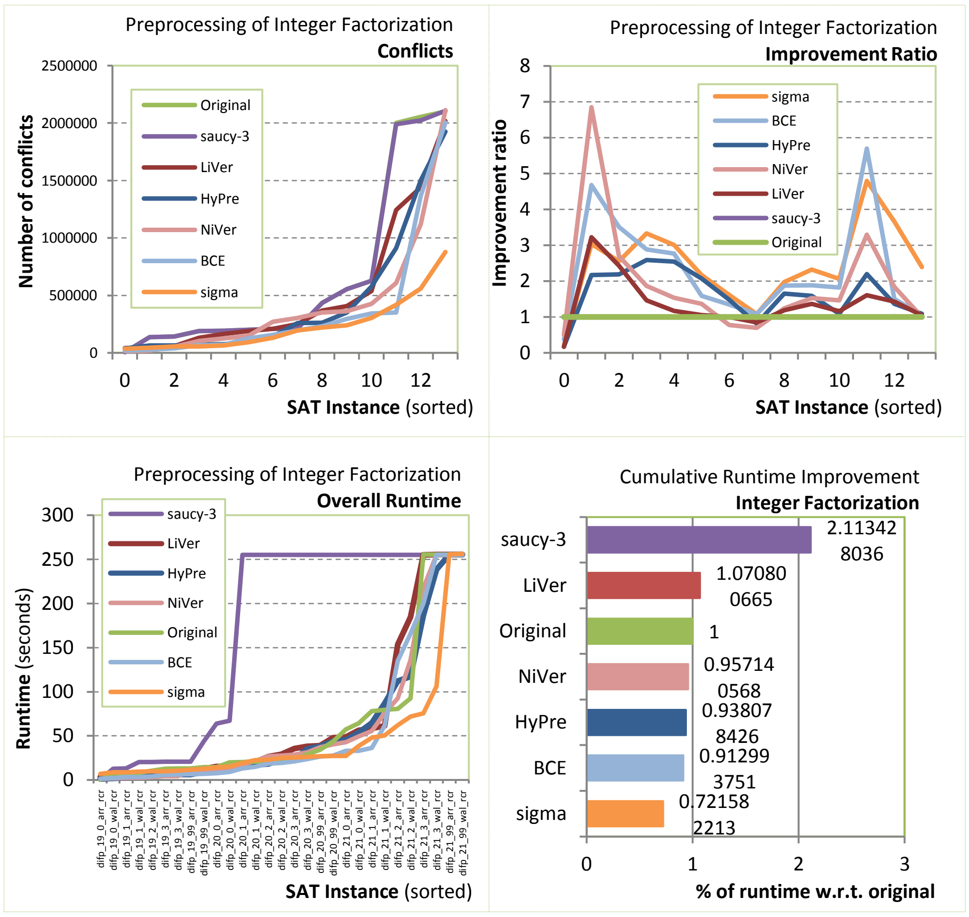


Figure 7. *Competitive evaluation in an instance encoding the integer factorization problem.*
Upper left: The absolute number of conflicts that `MiniSAT 2.2` has encountered in instances encoding integer factorization [3] after preprocessing by tested SAT preprocessors is shown. Clearly, `preprocessSIGMA` provides the best performance while `saucy-3` surprisingly lost to all the preprocessors. `BCE` seems to be a good option on integer factorization although it delivers mediocre performance on other instances from the tested set.
Upper right: Improvement ratio in terms of the number of conflicts is shown. Instances are sorted in the same order as in the previous figure.
Lower left: Runtime measurement also includes instances where `saucy-3` did not finish in the given timeout of 256.0 seconds which is approximately half of the instances encoding integer factorization.
Lower right: The aggregated improvement achieved by `preprocessSIGMA` in the overall runtime is $100.00 - 72.16 = 27.84\%$ on integer factorization. The second best `BCE` lost by a significant margin of almost 20% to the winner.

The cumulative runtime improvement achieved by `preprocessSIGMA` on integer factorization instances is 27.84% compared to 19.13% on the complete set of testing instances.

A surprising result has been obtained for `saucy-3` which was unexpectedly outperformed by all the local inference based preprocessors `BCE`, `HyPre`, `LiVer`, and `NiVer`.

5.3. *Experimental Evaluation of the Variables Selection Process*

The last part of the experiments was devoted to an evaluation of the selection of variables for consistency checks. This evaluation is important in order to verify whether all the internal processes of B2C-consistency worked as expected. This aspect concerns mainly the selection of a list of variables for the consistency check.

The expected number of satisfied tuples of values over the variables selected by Algorithm 3 with the setup of $\langle \varepsilon_L, \varepsilon_U \rangle = \langle 3.0, 32.0 \rangle$ over all the consistency checks on the tested instances has the following probabilistic characteristics – minimum, first quartile, median, third quartile, maximum equal to 2.131, 14.899, 27.562, 130.149, and 1,141,710.567 respectively (in this test only instances from SATLib were used). A more detailed insight into the distribution of the expected number of satisfied tuples of values over selected variables is provided in a partial histogram shown in

Figure ***8***.

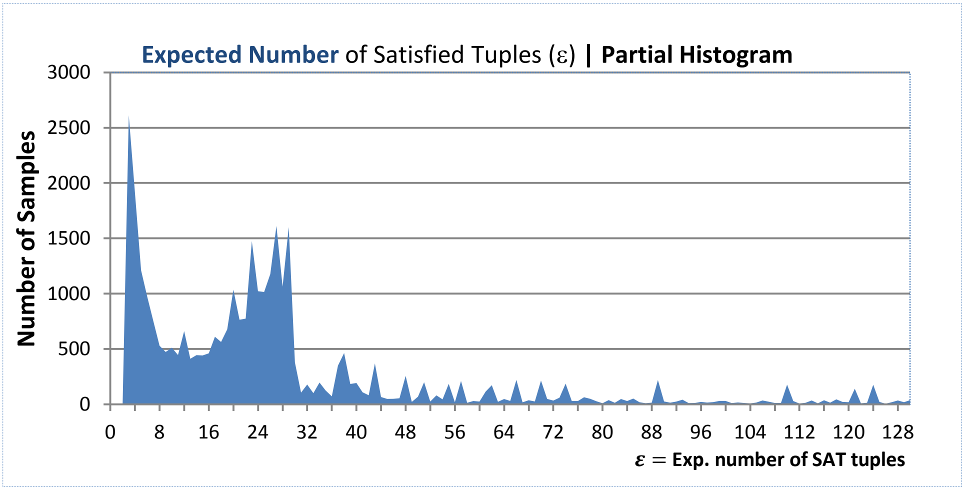


Figure 8. *Partial histogram of the expected number of satisfied tuples (ε).* The histogram characterizes the selection of variables made by Algorithm 3 over all the testing SAT instances and all the B2C-Consistency checks. Only the part up to the 3rd quartile is shown. It can be observed that most of the selections of tuples of variables have the expected number of satisfied tuples of values within the interval $\langle \varepsilon_L, \varepsilon_U \rangle = \langle 3.0, 32.0 \rangle$ as it was required.

5.4. *Summary of Experimental Evaluation*

If we summarize the results of the experimental evaluation we can state that B2C-consistency with the proposed process for the selection of variables represents a powerful technique that can be used as a basis of a SAT preprocessing tool. Our experimental evaluation has proven that prototype preprocessing tool `preprocessSIGMA` based on B2C-consistency is fully competitive with respect to the existent prominent SAT preprocessing tools in terms of saving the number of conflicts as well as in terms of the overall runtime. Competitiveness in terms of runtime has been achieved despite the not well optimized implementation of the prototype.

An especially good performance was exhibited by `preprocessSIGMA` in instances encoding integer factorization problems where there is still room for fine tuning the parameters of B2C-consistency to achieve yet better performance.

The evaluation of the internal characteristics of our prototype preprocessing tool – namely the evaluation of the process of selection of the list of variables for consistency check – indicates a good match with theoretical expectations.

6. Conclusion and Future Work

In this paper, a new type of consistency called B2C-consistency (bounded $(2,k)$-consistency) for use in Propositional satisfiability (SAT) has been presented. This new consistency has been inspired by both global constraints and local consistency. Basically, it is $(2,k)$-consistency with the bounded number of search steps for proving inconsistency enriched by reasoning over complete sub-graphs of pair-wise conflicting literals. Reasoning over complete sub-graphs introduces a global aspect into proving inconsistency and it can improve the consistency enforcing process significantly especially in SAT instances encoding the well known P/H principle (pigeon/hole principle) and similar principles which are known to be difficult for a standard solving process based on search.

The whole design of new consistency is explained in the context of modeling SAT as a constraint satisfaction problem (CSP) using the so-called explicit literal encoding (that is, literal encoding with explicit clauses obtained by singleton unit propagation).

Next we investigated probabilistic properties of the so-called generalized P/H principle – particularly the expected number of satisfied (consistent) tuples of values with respect to a tuple of the selected variables for consistency check. The investigation showed that a certain distribution of the expected number of satisfied tuples is characteristic for the P/H principle where many inconsistent tuples of values can be found. Therefore we proposed a process for the selection of variables which is trying to select variables so that the corresponding expected number of satisfied tuples of variables has a similar probabilistic distribution as in the case of the P/H principle. Using this process, we are trying to identify difficult sub-problems (such as the P/H principle) that can be yet resolved by B2C-consistency.

To evaluate our proposal we implemented B2C-consistency and the process of selection of variables within the prototype SAT preprocessing tool `preprocessSIGMA`. The experiments have confirmed that B2C-consistency and the variable selection process are beneficial and that we are able to select variables for consistency checks with similar probabilistic characteristics as in the case of the generalized P/H principle. The competitive evaluation on a set of 344 SAT instances from SATLib, SAT Competition 2003/2004 and 2007/2009 (mixture of satisfiable and unsatisfiable) showed that `preprocessSIGMA` delivers better results than the existent preprocessing tools `BCE`, `HyPre`, `LiVer`, and `Niver` which are based on local reasoning and comparable results to `saucy-3` based on symmetry breaking. In instances encoding the integer factorization problem `preprocessSIGMA` performed as far the best of all the tested preprocessing tools. Moreover, `preprocessSIGMA` has some advantages with respect to the comparable `saucy-3`. It is easier to implement – in `saucy-3`, graph isomorphism which, in itself, is a difficult problem needs to be solved – and it has many parameters that can be further fine tuned. Note that we have achieved a competitive performance despite the not well optimized implementation of `preprocessSIGMA`.

There are several interesting questions for future work. At present, we used a characterization of the distribution of an expected number of satisfied tuples of values with two parameters – the lower and the upper bound. It would be interesting to use more parameters to control the shape of the resulting distribution over all the consistency checks more precisely.

Another interesting investigation may be done with a repeated use of B2C-consistency. Consider a preprocessed instance to be preprocessed once again. Unfortunately, this approach is impractical at the current implementation stage as the setup of preprocessing is relatively time-consuming, and in order to preserve relatively acceptable competitiveness we cannot afford to run the process more than once. However, a more efficient implementation may change the situation.

References

1. Aloul, F. A., Ramani, A., Markov, I. L., Sakallah, K. A., *Solving Difficult SAT Instances in the Presence of Symmetry.* Proceedings of the 39th Design Automation Conference (DAC 2002), pp. 731-736, USA, (ACM Press, 2002, http://www.aloul.net/benchmarks.html, [March 2011]).
2. Aloul, F. Markov, I. L., Sakallah, K., Shatter: *Efficient Symmetry-Breaking for Propositional Satisfiability.* Proceedings of the Design Automation Conference (DAC 2003), pp. 836-839, (ACM Press, 2003, http://www.aloul.net/Tools/shatter/, [July 2011]).
3. Aloul, F. A., *SAT Benchmarks, Difficult Integer Factorization Problems.* Research web page, (http://www.aloul.net/benchmarks.html, [March 2011]).
4. Anbulagan, Slaney, J., *Multiple Preprocessing for Systematic SAT Solvers*. Proceedings of the 6th International Workshop on the Implementation of Logics, (CEUR-WS.org, 2006).
5. Bacchus, F., Winter, J., *Effective Preprocessing with Hyper-Resolution and Equality Reduction.* Proceedings of the Theory and Applications of Satisfiability Testing, 6th International Conference, (SAT 2003), pp. 341-355, LNCS 2919, (Springer 2004, http://www.cs.toronto.edu/~fbacchus/sat.html, [July 2011]).
6. Bennaceur, H., *The satisfiability problem regarded as constraint-satisfaction problem.* Proceedings of the 12th European Conference on Artificial Intelligence (ECAI 1996), pp. 155-159, (John Wiley and Sons, 1996).
7. Bessière, C., Hebrard, E., Walsh, T., *Local Consistencies in SAT.* Proceedings of the Theory and Applications of Satisfiability Testing, 6th International Conference (SAT 2003), pp. 299-314, LNCS 2919, (Springer, 2004).
8. Biere, A., Heule, M., van Maaren, H., Walsh, T., *Handbook of Satisfiability*. IOS Press, 2009.
9. Bomze, I. M., Budinich, M., Pardalos, P. M., Pelillo, M., *The maximum clique problem, Handbook of Combinatorial Optimization.* (Kluwer Academic Publishers, 1999).
10. Cook, S. A., *The Complexity of Theorem Proving Procedures.* Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971), pp. 151-158, ACM Press, 1971.
11. Dechter, R., *Constraint Processing.* (Morgan Kaufmann Publishers, 2003).

12. Dowling, W., Gallier, J., *Linear-time algorithms for testing the satisfiability of propositional Horn formulae.* Journal of Logic Programming, Volume 1 (3), pp. 267-284, (Elsevier Science Publishers, 1984).
13. Eén, N., Sörensson, N., *An Extensible SAT-solver.* Proceedings of Theory and Applications of Satisfiability Testing, 6th International Conference (SAT 2003), pp. 502-518, LNCS 2919, (Springer 2004, http://minisat.se/MiniSat.html, [July 2011]).
14. Freuder, E. C., *A Sufficient Condition for Backtrack-Free Search.* Journal of the ACM, Volume 29, pp. 24-32, (ACM Press, 1982).
15. Golumbic, M. C., *Algorithmic Graph Theory and Perfect Graphs.* (Academic Press, 1980).
16. Heule, M., Järvisalo, M., Biere, A., *Clause Elimination Procedures for CNF Formulas.* Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference (LPAR 2010), pp. 357-371, LNCS 6397, (Springer 2010).
17. Hoos, H. H., *Programming by optimization.* Communications of the ACM, Volume 55(2), pp. 70-80, (ACM Press, 2012).
18. Hoos, H. H., Stützle, T., *SATLib: An Online Resource for Research on SAT.* Proceedings of Theory and Applications of Satisfiability Testing, 4th International Conference (SAT 2000), pp.283-292, (IOS Press, 2000, http://www.satlib.org, [March 2011]).
19. Jackson, P., Sheridan, D., *Clause Form Conversions for Propositional Circuits.* Theory and Applications of Satisfiability Testing, 7th International Conference (SAT 2004), Revised Selected Papers, pp. 183–198, Lecture Notes in Computer Science 3542, Springer 2005.
20. Järvisalo, M., Biere, A., Heule, M., *Blocked Clause Elimination.* Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference (TACAS 2010), pp. 129-144, LNCS 6015, Springer, 2010.
21. Katebi, H., Sakallah, K. A., Markov, I., L., *Symmetry and Satisfiability: An Update.* Proceedings of Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference (SAT 2010), pp. 113-127, LNCS 6175, Springer 2010.
22. Papadimitriou, C., *Computational Complexity.* (Addison Wesley, 1994).
23. Petke, J., Jeavons, P., *Local Consistency and SAT-Solvers.* Proceedings of the Principles and Practice of Constraint Programming - 16th International Conference (CP 2010), pp. 398-413, Lecture Notes in Computer Science 6308, (Springer 2010).
24. Seidel, R., *On the Complexity of Achieving K-Consistency.* Technical Report, University of British Columbia Vancouver, BC, (Canada,1983).
25. Subbarayan, S., Pradhan, D., K., *NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT Instances*. Proceedings of The 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), pp. 276-291, LNCS 3542, (Springer 2005, http://www.itu.dk/people /sathi/niver.html, [July 2011]).
26. Surynek, P., *Solving Difficult SAT Instances Using Greedy Clique Decomposition.* Proceedings of the 7th Symposium on Abstraction, Reformulation, and Approximation (SARA 2007), LNAI 4612, pp. 359-374, (Springer, 2007).
27. Surynek, P., *sigmaSAT / preprocessSIGMA - a collection of experimental SAT processing tools.* Research web page, Charles University in Prague, 2011, http://ktiml.mff.cuni.cz/~surynek/index.html.php?select=software&product=sigmasat, [July 2011].
28. Torán, J., *The Hardness of Graph Isomorphism.* SIAM Journal on Computing, , pp. 1093-1108, volume 33, number 5, SIAM, 2004.
29. Walsh, T., *SAT vs. CSP.* Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, 441-456, LNCS 1894, Springer Verlag, 2000.