Artificial Intelligence Review manuscript No.
(will be inserted by the editor)

Evaluating Logic Gate Constraints in Local Search for Structured Satisfiability Problems

M A H Newton[⋆] · **M M A Polash**[⋆] ·
D N Pham[⋆][†] · **J Thornton**[⋆][‡] ·
K Su[⋆] · **A Sattar**[⋆]

Received: date / Accepted: date

Abstract Conjunctive normal forms (CNF) of structured satisfiability problems contain logic gate patterns. So Boolean circuits (BC) by and large can be obtained from them and thus structural information that are lost in the CNF can be recovered. However, it is not known which logic gates are useful for local search on BCs or which logic gates in particular help local search the most and why. In this article, we empirically show that exploitation of xor, xnor, eq, and not gates is a key factor behind the performance of local search algorithms using single variable flips when adapted to logic gate constraints. Moreover, controlled experiments and investigations into the variables selected for flipping further elucidates these findings. To achieve these conclusions, we have adapted the AdaptNovelty+ and CCANr algorithms to cope with logic gate-based constraint models. These are two prominent families of local search algorithms for satisfiability. We performed our experiments using a large set of benchmark instances from SATLib, SAT2014, and SAT2020. We have also presented techniques to eliminate cycles among logic gates that are detected from CNF and to propagate equivalence of variables statically through the logic gate dependency relationships.

Keywords Satisfiability · Constraints · Local Search · Logic Gates

1 Introduction

Propositional Satisfiability (SAT) is an NP-complete problem (Cook 1971). It is of great interest both to the scientific and industrial communities. It is central to many domains of computer science and artificial intelligence. It is widely studied for its importance both in theory and applications.

Structured SAT problem instances in the conjunctive normal form (CNF) typically contain clausal patterns that represent logic gates. Logic gates comprise output dependent variables whose values functionally depend on that of input

Corresponding Author: M A H Newton (mahakim.newton@griffith.edu.au)
[⋆] Institute for Integrated and Intelligent Systems, Griffith University, Australia
[†] Artificial Intelligence Lab, MIMOS Berhad, Malaysia
[‡] Department of Informatics, University of Sussex, United Kingdom

variables (Ostrowski et al. 2002). Variables whose values do not functionally depend on that of any other variables are called independent variables. Detection of such logic gates essentially allows to obtain Boolean Circuits (BC). Consequently, identified dependent variables are eliminated from the search space representation and implicitly satisfied clauses representing those logic gates are also removed from the search objective function. Values of dependent variables can be immediately or transitively inferred from the values of the independent variables, leaving the search exponential only in the number of independent variables. This is a great advantage. However, removing the clauses badly affects the effectiveness of the scoring functions used to differentiate the quality of the neighbouring solutions in the search space, since the possible measurement of badness has fewer degrees when such clauses are removed. There are two other drawbacks of recognising BCs. Cyclic dependencies between variables can appear as a hindrance to the computation of the dependent variables. Also, computation of dependent variables under any change in the independent variables can incur an overhead to the search. Dependency relationships identified from the CNF formulas can be nested to a large depth. Deeply nested dependencies increase change propagation cost.

One way to efficiently simplify a given CNF formula is to detect logic gate structures using a coprocessor (Manthey 2012). The simplified and reduced CNF formula can then be used during search (Balint and Fröhlich 2010; Cai et al. 2015). Such approaches have significantly boosted performance of systematic search solvers. This is achieved first by identifying the dependent variables and then by running a version of the Davis-Putnam-Logemann-Loveland (DPLL) procedure (Davis et al. 1962) on the remaining independent variables (Jeroslow and Wang 1990). In this process, if any selection decision contradicts the output of an identified gate, the selection is discarded and a backtracking move is made. Further research on conflict-driven clause learning (CDCL) and input restricted branching (IRB) has been reported (Järvisalo and Niemelä 2008; Järvisalo and Junttila 2009). Recent SAT competitions are won by solvers (Manthey et al. 2016; Ryvchin and Nadel 2018; Biere et al. 2020) that are mostly CDCL based.

Over time, algorithms have been developed to detect various logic gates using their clausal representations (Tseitin 1983; Plaisted and Greenbaum 1986; Ostrowski et al. 2002; Roy et al. 2004). Other gate detection algorithms use pattern matching techniques (Fu and Malik 2007; Seltner 2014) on CNF formulas, but they find maximal acyclic circuits selecting maximum subsets of matching gates. Yet other gate detection algorithms use connections between gate structures and blocked sets (Järvisalo et al. 2012; Balyo et al. 2014; Iser et al. 2015), but on other types of problem encoding than CNF. This work uses CNF encoding for input.

Stochastic local search (SLS) is known as the most effective approach for solving random satisfiable problem instances. However, for structured satisfiable problem instances, SLS solvers have not been as effective as systematic solvers, particularly those that incorporate CDCL techniques. A potential reason for this is that the best performing pure SLS solvers (which have not been hybridised with systematic solvers), such as CCAnr (Cai et al. 2015) that uses configuration checking techniques), do not directly exploit the structural properties that remain hidden in CNF. The hybrid solvers e.g. CryptoMiniSat+CCAnr (Soos et al. 2020) and other solvers that use CCAnr perform well in the satisfiability competitions (Heule et al. 2018; Balyo et al. 2020; Peng et al. 2020; Luo et al. 2020). Besides CCAnr, other recent pure local search algorithms include probSAT (Heule et al. 2018) and YalSAT

(Biere 2016) but they work in tandem with CDCL solvers. Further local search algorithms that work on random satisfiability instances include BRASP(Fu et al. 2020) and EPEFV(Fu et al. 2021). This work considers only pure local search solvers for structured satisfiable problem instances.

Following the work by Ostrowski et al. (2002) on extracting logic gates from CNF formulas, Pham et al. (2007) in a landmark IJCAI best paper study, developed the notion of a dependency lattice—essentially a multi-layered directed acyclic graph (DAG)—in order to dynamically calculate the cost of potential SLS moves in CNF formulas from which logic gates had been extracted. Pham et al. (2007) then adapted the AdaptNovelty+ (Hoos 2002) algorithm to cope with logic gates and performed preliminary experiments to show that the new algorithm could solve a small selection of hard structured benchmark satisfiability problem instances significantly faster than its original CNF based counterpart. Unfortunately, not much work has been done later in this direction, particularly on how logic gates can be better exploited within local search taking their pros and cons into account. Moreover, no extensive experimentation has been reported yet using a large benchmark set or any other local search algorithm. As such the key lower level technical reasons behind these performance boost are yet to be known.

In this article, we extend the work of Pham et al. (2007): (*i*) We describe the solver system formally and in great details; (*ii*) We perform very extensive experiments using a large number of benchmark instances from SATLib and SAT competition 2014 and 2020 (SAT14 and SAT20); (*iii*) We analyse the effects of various types of logic gates and their interactions, and identify useful logic gates for local search. (*iv*) We develop procedures to detect and remove dependency cycles between logic gates so that a DAG can still be obtained; (*v*) We develop procedures to statically propagate through the DAG the equivalence relations detected between variables by the gate identification procedures; (*vi*) Besides using AdaptNovelty+, we evaluate these extensions using another recent SLS solver named CCAnr (Cai et al. 2015). These two actually represent two different prominent families of local search algorithms for CNF based satisfiability problems. For this work, both algorithms are adapted to cope with logic gate constraints. All implementations have been done on top of Kangaroo (Newton et al. 2011), a generic Constraint-Based Local Search (CBLS) system that supports efficient incremental propagation of changes through the DAG during search. While we reaffirm the basic conclusion of Pham et al. (2007) but we do that using a large set of benchmark instances (SATLib, SAT14, and SAT20) and two different local search algorithms (AdaptNovelty+ and CCAnr). However, our main and new contribution in this paper is to identify the logic gates that are behind the performance boost of the local search algorithms. We empirically show that exploitation of xor, xnor, eq, and not gates is a key factor behind this performance. Presence of large numbers of these gates in the aforementioned benchmark instances supports this. Moreover, controlled experiments and investigations into the variables selected for flipping further elucidates these findings.

The rest of the article is organised as follows: Section 2 provides preliminary knowledge; Section 3 describes our search model; Section 4 describes our search algorithms; Section 5 presents our experimental results and analyses; and finally, Section 6 concludes this article.

2 Preliminaries

We provide notations and terminologies for satisfiability problems using CNF and BC. We also briefly overview local search algorithms for satisfiability, detection of gates from clausal patterns, and propagation of gate constraints. Some parts of these overviews could be skipped by an expert reader.

2.1 CNF Satisfiability

A **variable** v has its **domain** $B \equiv \{\mathsf{true}, \mathsf{false}\}$ with only Boolean values. A **literal** l is a variable v or its **negation** $\overline{v}$ with its **involving variable** $v(l) \equiv v$. A literal $l \equiv v$ is a **positive literal** with **sign** $s(l) = \mathsf{true}$. A literal $l \equiv \overline{v}$ is a **negative literal** with **sign** $s(l) = \mathsf{false}$. Literals v and $\overline{v}$ are **opposite** to each other. Moreover, literal $\overline{l}$ denotes the **opposite literal** of a given literal l.

A **clause** $c \equiv \bigvee_k l_k$ is a **disjunction** of $|c|$ literals. An **empty clause** Θ has no literal and has a value false. A **unit clause** has only one literal and has the same value as the literal has. A **satisfied clause** has a value true while a **falsified clause** has a value false. We use $l \in c$ to denote a literal l **appears** in a clause c. We also use $v \sqsubset c$ to denote a variable v **appears** in a clause c i.e. there is a literal $l \in c$ such that $v(l) = v$. We further use $V(c) \equiv \{v : v \sqsubset c\}$ to denote the set of **involving variables** of a clause c.

A **CNF formula**, or **formula**, $F \equiv \bigwedge_j c_j$ is a **conjunction** of $|F|$ clauses. An **empty formula** Ω has no clause and has a value true. A formula having one clause has the same value as the clause has. A **satisfied formula** has a value true while a **falsified formula** has a value false. We use $c \in F$ to denote a clause **appears** in a formula F and $\|c\| = \max_{c \in F} |c|$ to denote the maximum number of literals in a clause in the formula F. We also use $v \sqsubset F$ to denote a variable v **appears** in a formula F i.e. there is a clause $c \in F$ such that $v \sqsubset c$. We further use $V(F) \equiv \{v : v \sqsubset F\}$ to denote the set of **involving variables** of a formula F.

Given a set V of variables and $W \subseteq V$, a **partial assignment** $\overrightarrow{W} \equiv \{\langle v, \overrightarrow{v}\rangle : v \in W\}$ **assigns** a value $\overrightarrow{W}(v) = \overrightarrow{v}$ to each variable $v \in W$ just once, where $\overrightarrow{v} \in B$. Given a clause $c : V(c) \subseteq W$ or a formula $F : V(F) \subseteq W$, a **satisfying partial assignment** (or **falsifying partial assignment**) $\overrightarrow{W}$ for c or F **satisfies** (or **falsifies**) c or F respectively. A partial assignment $\overrightarrow{W}$ is called an **assignment** $\overrightarrow{V}$ if $W = V$; we define **satisfying** and **falsifying assignment** analogously.

A **CNF SAT problem instance** $S \equiv \langle V, F\rangle$ has a set V of variables and a CNF formula F such that $V(F) \subseteq V$. Given a CNF SAT problem instance $S \equiv \langle V, F\rangle$, the problem is to find a satisfying assignment $\overrightarrow{V}$ for the formula F. If there exists such a satisfying assignment $\overrightarrow{V}$ of S, then $\overrightarrow{V}$ is a **solution** to S and S is **satisfiable**; otherwise S is **unsatisfiable**.

Unit clause propagation (UCP) is a well-known method to simplify a SAT formula (Davis et al. 1962). Given a formula F, for each unit clause $c \in F$ with literal $l \in c$, a value is assigned to the variable $v(l)$ such that c is true. Each clause $c \in F$ having literal $l \in c$ are the removed form F. Also, each literal $\overline{l}$ is removed from each remaining clause $c \in F$ with $\overline{l} \in c$, leading to further unit clauses.

2.2 Local Search for CNF SAT

A **Local Search (LS)** algorithm for SAT starts from an **initial assignment** $\overrightarrow{V_0}$. The initial assignment is typically obtained by assigning a random value to each variable. In each **iteration** $i \geq 0$, the search then moves from the **current assignment** $\overrightarrow{V_i}$ to the **next assignment** $\overrightarrow{V_{i+1}}$ in quest of a **'better'** assignment until a satisfying assignment is found or a given **termination criterion** is met.

A **local move**, or **move**, is an operation that changes values of a number of variables i.e. represents a partial assignment. In this work, we consider a **flip move** that at a time changes the value of only one variable and changes it either from true to false or vice versa, since the domain of each variable is B. The **application** of a **selected move** in each iteration in local search is typically preceded by an **evaluation** of a subset of **potential moves** to **select** one of the **'best'** ones.

Local search algorithms for SAT typically uses three metrics to evaluate a potential flip move. Given the current assignment $\overrightarrow{V}$, $\mathsf{make}(\overrightarrow{V}, v)$ (or $\mathsf{break}(\overrightarrow{V}, v)$) is the number of falsified (or satisfied) clauses that will become satisfied (or falsified) after flipping v, and $\mathsf{score}(\overrightarrow{V}, v)$ is the difference $\mathsf{make}(\overrightarrow{V}, v) - \mathsf{break}(\overrightarrow{V}, v)$.

LS algorithms are **incomplete** but are still embraced in SAT since they often find solutions quickly. LS algorithms might suffer from **revisitation** of the same assignment, face **stagnation** at plateaus, or get stuck at **local optima**. Various approaches are taken to address these issues; a few of them are below.

- **Using Stochasticity:** Instead of adopting **greedy strategies** to generate potential moves or to accept the selected move, decisions could be taken **incorporating randomness**. SLS does this.
- **Random Restart:** A **full restart** randomly assigns values to all variables. A **partial restart** only to a subset of variables (Ryvchin and Strichman 2008).
- **Tabu Technique:** A variable flipped in the current iteration cannot be flipped for the next τ iterations, where τ is the **tabu tenure** (Mazure et al. 1997).
- **Configuration Checking (CC):** A variable once flipped cannot be flipped again until one of its neighbour is flipped, where the **neighbour** relationship $\mathsf{neighbour}(v_0, v_1) : V \times V \mapsto B$ between each two variables $v_0 \neq v_1$ are to be defined in a problem specific way such that $\mathsf{neighbour}(v_0, v_1) = \mathsf{neighbour}(v_1, v_0)$ (Cai et al. 2015). For a given SAT instance $S \equiv \langle V, F \rangle$, we define $\mathsf{neighbour}(v_0, v_1)$ to be true, if there is a clause $c \in F$ such that $v_0 \sqsubset c$ and $v_1 \sqsubset c$; otherwise $\mathsf{neighbour}(v_0, v_1)$ is false. CC and tabu both help avoid revisitation. While tabu is chronology based, CC is problem structured based.

2.3 Boolean Circuits

A **logic gate**, or **gate**, has a form $v_0 \leftarrow\!\dashv g(v_1, v_2, \ldots, v_n)$, where n is a positive integer, $g : B^n \mapsto B$ is a **function**, v_0 is g's **output variable**, and $v_1, v_2, \ldots, v_n$ with $v_k \neq v_0$ for any $1 \leq k \leq n$ are g's **input variables** or **parameters**. A gate with n input variables is an **n-ary gate**. We say v_0 has a **definition** $v_0 \equiv g(v_1, \ldots, v_n)$ using a **defining function** g on variables $v_1, \ldots, v_n$. We use $v_0 \ll v_k$ where $1 \leq k \leq n$ to denote v_0 computationally **immediately depends** on v_k. Each gate essentially imposes a constraint of satisfying the functional consistency.

The output variable of a gate can be an input variable of another gate. This can go on, assumingly finitely, leading to a **directed acyclic graph**. Given a set C of gates, we use $\Lambda(C)$ to denote its set of **in-out variables** obtained by accumulating the input and output variables of the gates.

Given a set C of gates and their in-out variables, for any two variables $v_0 \neq v_1$, we use $v_0 \lll v_1$ to denote v_0 computationally **transitively depends** on v_1 i.e. either $v_0 \ll v_1$ or there exists a variable v_2 such that $v_0 \ll v_2$ and $v_2 \lll v_1$.

Given a set C of gates and their in-out variables, a variable is called a **dependent variable** if it is the output variable of any gate in the given set; otherwise, it is called an **independent** variable. A dependent variable is called a **nonterminal variable** if it is an input variable of any gate in the given set; otherwise, it is called a **terminal variable**. Let I, D, N and T respectively be the sets of independent, dependent, nonterminal, and terminal variables. So $\Lambda(C) \equiv I \cup D$ and $D \equiv N \cup T$.

For a dependent variable v_0, the **involving variables** of v_0 or its defining function g is a set $I(v_0) \equiv I(g) \equiv \{v_1 \in I : v_0 \lll v_1\}$ of independent variables that v_0 transitively depends on. A dependent variable v_0 has a **cyclic dependency** if there is another dependent variable $v_1 \neq v_0$ such that $v_0 \lll v_1$ and $v_1 \lll v_0$. A **dependency cycle** involving a dependent variable v_0 is defined as a sequence $\langle v_0, v_1, \ldots, v_n, v_0\rangle$, where $v_k \ll v_{(k+1) \bmod (n+1)}$ for each $0 \leq k \leq n$.

A **Boolean circuit**, or **circuit**, is a set C of gates such that (i) no two gates has the same output variable (**no multiple definitions**) and (ii) no output variable has a cyclic dependency (**no cyclic dependency**). Besides the essential properties, a circuit can have a desirable property to ensure there exists no two dependent variables with the same defining function (**no repeated computation**).

A **constrained circuit** $\Sigma \equiv \langle C, W\rangle$ comprises a circuit C and a subset W of terminal variables such that each **constrained variable** $v \in W$ has value true. A constrained circuit is **satisfied** if each constrained variable has value true; otherwise, the constraint circuit is **falsified**. Given a constrained circuit $\Sigma \equiv \langle C, W\rangle$, if there is an assignment $\overrightarrow{I}$ that results in satisfaction of Σ as values of dependent variables are computed through the circuit, then $\overrightarrow{I}$ is a **solution** to Σ and Σ is **satisfiable**; otherwise, Σ is **unsatisfiable**.

Given a circuit C, an independent variable v has a **gate level** $L(v) = 0$ and a dependent variable $v_0 \equiv g(v_1, \ldots, v_n)$ has $L(v_0) = \max_{1 \leq k \leq n} L(v_k) + 1$.

2.4 Logic Gate Detection

Logic gates can be represented by formulas (Tseitin 1983; Plaisted and Greenbaum 1986; Ostrowski et al. 2002; Roy et al. 2004), which we show in (1)–(8) for eq, not, xor, xnor, and, or, nand, and nor gates. Gates eq, not, xor, and xnor have 2^n clauses each while gates and, or, nand, and nor have $n+1$ clauses each, where n is the arity of the gates. Each formula essentially produces a **clausal pattern** that can be used to identify unique subsets of clauses representing individual gates. As long as the output of a gate is consistent with the functional value computed from the input variables, the clauses in the clausal pattern representing the gate are all satisfied and thus all such clauses can be removed from the formula.

$$v_0 \hookleftarrow \mathsf{eq}(v_1) \equiv (\overline{v_0} \vee v_1) \wedge (v_0 \vee \overline{v_1}) \quad \text{odd negative literals per clause} \qquad (1)$$

$$v_0 \hookleftarrow \mathsf{not}(v_1) \equiv (v_0 \vee v_1) \wedge (\overline{v_0} \vee \overline{v_1}) \quad \text{even negative literals per clause} \qquad (2)$$

$$v_0 \hookleftarrow \mathsf{xor}(v_1,\ldots,v_n) \equiv \bigwedge_{m \text{ is odd}} \Big(\Big[\bigvee_k l_k\Big] : m \text{ of the } l_k\text{s are negative}\Big) \quad n > 1 \tag{3}$$

$$v_0 \hookleftarrow \mathsf{xnor}(v_1,\ldots,v_n) \equiv \bigwedge_{m \text{ is even}} \Big(\Big[\bigvee_k l_k\Big] : m \text{ of the } l_k\text{s are negative}\Big) \quad n > 1 \tag{4}$$

$$v_0 \hookleftarrow \mathsf{and}(v_1,\ldots,v_n) \equiv \Big(v_0 \vee \bigvee_{k>0} \overline{v_k}\Big) \wedge \Big[\bigwedge_{k>0} (\overline{v_0} \vee v_k)\Big] \quad n > 1 \tag{5}$$

$$v_0 \hookleftarrow \mathsf{or}(v_1,\ldots,v_n) \equiv \Big(\overline{v_0} \vee \bigvee_{k>0} v_k\Big) \wedge \Big[\bigwedge_{k>0} (v_0 \vee \overline{v_k})\Big] \quad n > 1 \tag{6}$$

$$v_0 \hookleftarrow \mathsf{nand}(v_1,\ldots,v_n) \equiv \Big(\overline{v_0} \vee \bigvee_{k>0} \overline{v_k}\Big) \wedge \Big[\bigwedge_{k>0} (v_0 \vee v_k)\Big] \quad n > 1 \tag{7}$$

$$v_0 \hookleftarrow \mathsf{nor}(v_1,\ldots,v_n) \equiv \Big(v_0 \vee \bigvee_{k>0} v_k\Big) \wedge \Big[\bigwedge_{k>0} (\overline{v_0} \vee \overline{v_k})\Big] \quad n > 1 \tag{8}$$

We generalise and and nor gates to a cg (conjunction generalised) gate, and or and nand to a dg (disjunction generalised) gate. While and, or, nand, nor gates use input variables directly, the cg and dg gates internally allow selective negation of input variables before using them. To achieve this functionality, we define a **juxtaposition** operator e.g. $s_k v_k \equiv (\mathsf{if}\ s_k\ \mathsf{then}\ v_k\ \mathsf{else}\ \overline{v_k})$. The cg and dg gates use the juxtaposition operator along with the static parameters s_ks, which are specified as part of the gate definition, and do not change during search. The input variables v_ks, in contrast, take various values during search.

$$v_0 \hookleftarrow \mathsf{cg}_{\langle s_1,\ldots,s_n\rangle}(v_1,\ldots,v_n) \equiv \Big(v_0 \vee \bigvee_{k>0} \overline{s_k v_k}\Big) \wedge \Big[\bigwedge_{k>0} (\overline{v_0} \vee s_k v_k)\Big] \quad n > 1 \tag{9}$$

$$v_0 \hookleftarrow \mathsf{dg}_{\langle s_1,\ldots,s_n\rangle}(v_1,\ldots,v_n) \equiv \Big(\overline{v_0} \vee \bigvee_{k>0} s_k v_k\Big) \wedge \Big[\bigwedge_{k>0} (v_0 \vee \overline{s_k v_k})\Big] \quad n > 1 \tag{10}$$

We define gate uniqueness and gate categories for the gates mentioned above.

Definition 1 (Gate Uniqueness) We uniquely identify each logic gate with its clausal pattern and the variables in those clauses. For example, $v_0 \hookleftarrow \mathsf{eq}(v_1) \equiv (\overline{v_0} \vee v_1) \wedge (v_0 \vee \overline{v_1})$ and $v_1 \hookleftarrow \mathsf{eq}(v_2) \equiv (\overline{v_1} \vee v_2) \wedge (v_1 \vee \overline{v_2})$ are two different eq gates, both having the same clausal pattern.

Definition 2 (Gate Types) Gate types are based on various properties of the defining functions such as **conjunction** (c), **disjunction** (d), **exclusion** (x), **equivalence** (e), **parity** (p), and **aggregation** (a).

ctype = {and, nor, cg}	xtype = {xor, xnor}	atype = ctype ∪ dtype
dtype = {or, nand, dg}	etype = {eq, not}	ptype = xtype ∪ etype

We refer to Ostrowski et al. (2002) and Roy et al. (2004) for gate detection algorithms using (1)–(10). Below we summarise key aspects of gate detection.

atype: The output variable of an n-ary atype gate can be uniquely identified from the clause having $n+1$ literals. From a formula F having no repeated literals and no repeated clauses, using (5)–(10), we can detect all possible atype gates uniquely in $\mathcal{O}(|F| \times \|c\|^2)$ time taking $\mathcal{O}(|F| \times \|c\|)$ memory. Each such gate is detected once. There could be at most $\frac{1}{3}|F|$ atype gates; this happens when each gate is a binary atype gate with 3 clauses.

ptype: The output variable of a ptype gate cannot be uniquely identified from its clausal pattern; each variable appearing the clausal pattern is a candidate. Later, we discuss how in this work we determine output variables of ptype gates. From a formula F having no repeated literals and no repeated clauses, using (5)–(10), we can detect all possible ptype gates uniquely without determining their output variable in $\mathcal{O}(|F|^2 \times \|c\|)$ time taking $\mathcal{O}(|F| \times \|c\|)$ memory. Each such gate is detected once. There could be at most $\frac{1}{2}|F|$ ptype gates; this happens when each gate is an etype gate with 2 clauses.

2.5 Gate Constraint Propagation

The definition of each dependent variable imposes an obligation that the out variable of a gate must have a value equal to the functional value computed from the input variables. Discharging this obligation requires propagation of changes from the input variables of the gate to its output variables. One way to discharge this obligation is to use the **one-way constraint propagation**, where the output variable upon each query returns a value consistent with functional value computed from the values of the input variables; the consistency could be maintained on demand or upon any change in the input variables. An **invariant** in the constraint paradigm implements a one-way constraint propagation mechanism. An **invariant engine** provides the guarantee and as such updates the value of the dependent variables as the values of the independent variables change. While one-way propagation method is a **dynamic** technique that works during search, constraints can be propagate **statically** before the search. The value of a constrained variable can be replaced and the formula can be simplified. If a variable has the same or opposite value of another variable, the first can be replaced by the equivalent literal involving the other variable and the formula can be simplified.

2.6 SLS for Boolean Circuits

SLS algorithms for Boolean circuits were introduced by Järvisalo et al. (2008b,a); however, only justification of the circuit was sought rather than finding a satisfying assignment. The justification based SLS is inspired by the justification frontier heuristics by Kuehlmann et al. (2002)) for systematic search solvers. In justification based SLS, search steps aim at correcting local inconsistencies within a circuit by justifying unjustified gates that are selected from the justification frontier. The CRSat algorithm by Belov and Stachniak (2010) incorporated Boolean constraint propagation (BCP) (Belov and Stachniak 2009) technique in the SLS presented by Järvisalo et al. (2008b). Then, the D-CRSat (Belov et al. 2011) algorithm extended the CRSat algorithm by implementing structure-based heuristics and limited reasoning by forward propagation. The forward propagation essentially resembles to the dependency propagation through the DAG in this work.

This work is different from the research mentioned above since we do not have gate structures beforehand. We find gates from CNF using existing algorithms. Then, our main focus is to study which subsets (gate type wise) of those identified gates better improve local search algorithms and why.

3 Our Constrained Circuit for CNF SAT

We describe how to build constrained circuits from CNF SAT problem instances using and not using logic gates. We then describe how we adapt and implement state-of-the-art local search families to constrained circuits.

Given a formula, with our own implementation, we always perform certain preprocessing steps typically used in SAT research. We eliminate repeated literals from clauses keeping only one, eliminate clauses having opposite literals, eliminate repeated clauses from the formula keeping only one, and perform UCP to eliminate all unit clauses. These preprocessing steps need $\mathcal{O}(|F| \times \|c\|)$ time and $\mathcal{O}(|F| \times \|c\|)$ memory. Henceforth, we assume a given SAT problem instance has no unit clause, no repeated literal or opposite literals in the clauses, and no repeated clause in the formula.

We first describe how a baseline constrained circuit is obtained directly from CNF without detecting any gates. We then describe gate graphs that hold gate information as gates are detected and are then processed before building a constrained circuit. Next, we describe the steps involved in the processing of the gates. Finally, we describe building constrained circuits from processed gate graphs.

3.1 Baseline: CNF to Circuits without Gate Detection

Given a SAT instance $S \equiv \langle V, F\rangle$, for each clause $c_j \equiv l_1 \vee \ldots \vee l_n$, for convenience abusing the clause symbol c_j to treat as a variable symbol, we define a dg gate $c_j \hookleftarrow \mathsf{dg}_{\langle s(l_1),\ldots,s(l_n)\rangle}(v(l_1), \ldots, v(l_n))$. Accumulating all these gates, we get a set C of gates with $I = V$ and $D = T = \{c_j \in F\}$ and so obtain a constrained circuit $\Sigma = \langle C, T\rangle$. It is easy to prove that $\overrightarrow{I}$ is a solution to Σ iff $\overrightarrow{V}$ is a solution to S. The time and memory complexities of transforming S to Σ are both $\mathcal{O}(|V| + |F| \times \|c\|)$ while that of transforming a solution I to a solution V is $\mathcal{O}(1)$ as $I = V$.

3.2 Gate Graphs: Data Structures to Hold Gates

We define a gate node to hold each individual gate and a gate graph to hold all gate nodes. Gates are detected from the given formula of a SAT instance S. Also, gates and their output variables are added to get a constrained circuit Σ that captures S. Moreover, hypothetical null gates are added treating independent variables as their output variables. Gate nodes help simply a circuit using known values of some variables or equivalence relationships of some other variables.

Definition 3 (Gate Node) A **gate node** $N \equiv \langle o, g, \rho, \psi, \kappa, \epsilon, \nu, \phi, \sigma\rangle$ has a number of components described below. Each component can be accessed by using a dot operator. For example, $N.o$ denotes the o component of a gate node N.

- o denotes either the output variable of a gate or an independent variable. For ptype gates, until the gate output is determined, o might be temporarily null.
- g is the gate type (any type from $\mathsf{atype} \cup \mathsf{ptype}$) when o is the output of the gate and so is a dependent variable. If o is an independent variable, g is null.

$\rho = \{v : o \ll v\}$ are the input variables of the gate if the value of o is not fixed. When $|\rho| = 1$, instead of ρ, for convenience, we use i to denote the only input of the gate g. Notice that $\rho = \emptyset$, if o denotes an independent variable.

$\psi = \{v : v \ll o\}$ are the variables having o as an input in their defining functions, if the value of o is not fixed. Notice that $\psi = \emptyset$, if o denotes a terminal variable.

$\kappa \in \{\mathsf{true}, \mathsf{false}\}$ denotes whether o is a constrained (terminal) variable.

ϵ is either null or denotes the literal which o is equivalent to. If ϵ is not null, the variable denoted by o is to be **replaced** by the literal denoted by ϵ.

ν is either null or the value true/false as is fixed for the variable denoted by o.

ϕ denotes the clauses representing the gate detected/added, when g is not null.

$\sigma \in \{\mathsf{true}, \mathsf{false}\}$ denotes whether o is a SAT variable and is not an added one.

Some notes on gate nodes are below.

- $N \equiv \langle o, \mathsf{null}, \emptyset, \psi, \mathsf{false}, \mathsf{null}, \mathsf{null}, \Omega, \sigma\rangle$ represents an independent variable o.
- Each gate node has a unique o if ϕ is Ω i.e. represented by an empty set of clauses. If ϕ is not Ω, then g is not null as there is a corresponding gate.
- When a gate is detected, a gate node N is created with g, ρ, ϕ, and $\sigma = \mathsf{true}$. Moreover, $\psi = \emptyset$, $\kappa = \mathsf{false}$, $\epsilon = \mathsf{null}$, and $\nu = \mathsf{null}$ with o could be temporarily null and is set as soon as is known. A gate node later undergoes changes.

Definition 4 (Gate Graph) A **gate graph** G is simply a set of gate nodes.

1. A gate graph G is **complete** G has a gate node N with $N.o = v$ and $N'.o \in N.\psi$ for each $v \in N'.\rho$ for each gate node $N' \neq N$. Each parameter of each gate, as an independent variable or as an output of another gate, has a gate node in the gate graph. Moreover, each gate node has its ψ computed.
2. A gate graph G is **constrained** if there is a gate node N with $N.\kappa = \mathsf{true}$ and $N.o$ is a terminal variable. Only terminal variables can be constrained.
3. A gate graph G is **free from multiple definitions** if there are no two gate nodes $N \neq N'$ such that $N.o = N'.o$. Each o is unique in the gate graph.
4. A gate graph G is **free from repeated computations** if no two gate nodes $N \neq N'$ have $N.g = N'.g$ and $N.\rho = N'.\rho$ (assuming each possible g is associative and commutative, which is the case for atype and ptype gates).
5. A gate graph G is **free from dependency cycles** if it is complete and there is no gate node N such that $N.o$ has a cyclic dependency.

Using the definitions, the connection between gate graphs and circuits is clear.

Lemma 1 (Gate Graphs Representing Circuits) *A complete gate graph free from multiple definitions and cyclic dependencies represents a circuit, and if the gate graph is constrained then the circuit is also constrained.*

3.3 CNF to Circuits with Gate Detection

Algorithm 1 shows how we get a constrained circuit from a SAT problem instance. The first two mandatory steps are to detect the specified subsets of atype and/or ptype gates and to identify their output variables. The next mandatory steps include handling multiple definitions, adding variables for clauses not part of any gates, considering independent variables, detecting and removing dependency cycles, and computing list of variables dependent on other variables. The last two steps are optional: statically propagate equivalence relationships between variables and remove temporarily the variables that are not strictly needed during search. Later, we describe all these steps in details in the next sections.

Algorithm 1 Build Σ from S using logic gates

```
1. Using clausal patterns, detect specified unique gates (a subset of atype ∪ ptype gates)
2. Identify the output variables of those gates and remove clauses in clausal patterns
3. Handle multiple definitions of dependent variables, if any, adding xnor constraints
4. Get a constrained variable in $T$ for each clause left after removing gate clausal patterns
5. Get an independent variable in $I$ for each non-dependent variable left out in $V$
6. Compute the list of variables that depend on each dependent and independent variable
7. Detect and remove dependency cycles, if any, among the dependent variables identified
8. Statically propagate equivalence relationships between variables identified through gates
9. Remove temporarily from the circuit the variables that are not strictly needed in search
```

3.4 Identify Gates and Output Variables

In Step 1 of Algorithm 1, we detect the gates specified using the clausal patterns (1)–(10). In this work, we consider only subsets of atype and/or ptype gates. In Step 2 of Algorithm 1, we identify the output variables of the detected gates.

As discussed in Section 2.4, output variables of atype gates are uniquely determined from their clausal patterns. However, after this, we could get dependency cycles and multiple definitions. Example 1 shows an example of a dependency cycle. Example 2 shows an example of multiple definitions of a variable.

Example 1 (Dependency Cycles) An and gate and an or gate detected from the formula F shown below create a dependency cycle between variables v_0 and v_1.

Given Formula F			Gates Detected	Dependency Cycle
$v_0 \vee \overline{v_1} \vee \overline{v_2}$	$\overline{v_0} \vee v_1$	$\overline{v_0} \vee v_2$	$v_0 = \mathsf{and}(v_1, v_2)$	$v_0 \ll v_1$
$\overline{v_1} \vee v_0 \vee v_2$	$v_1 \vee \overline{v_0}$	$v_1 \vee \overline{v_2}$	$v_1 = \mathsf{or}(v_0, v_2)$	$v_1 \ll v_0$

Example 2 (Multiple Definitions) An and gate and an or gate detected from the formula F shown below create multiple definitions of variable v_0.

Given Formula F			Gates Detected	Multiple Definitions
$v_0 \vee \overline{v_1} \vee \overline{v_2}$	$\overline{v_0} \vee v_1$	$\overline{v_0} \vee v_2$	$v_0 = \mathsf{and}(v_1, v_2)$	v_0 is defined once
$\overline{v_0} \vee v_3 \vee v_4$	$v_0 \vee \overline{v_3}$	$v_0 \vee \overline{v_4}$	$v_0 = \mathsf{or}(v_3, v_4)$	v_0 is defined again

As discussed in Section 2.4, output variables of ptype gates cannot be uniquely determined from their clausal patterns; any variable appearing in the clausal pattern can be designated as the output. However, this approach could lead to dependency cycles or multiple definitions, if not done carefully.

Various approaches can be used to obtain an arrangement where all ptype gates have their output variables identified and there is no dependency cycle, assuming such an arrangement does actually exist. Contextual information such as which variables are already made input or output of some other gates can be used to determine or even to eliminate candidates for the output variable of a given gate. Systematic search approaches such as breadth-first search may also be used (Roy et al. 2004). Dependency cycles may exist any way as shown in Example 1 involving even only atype gates. Moreover, no arrangement of ptype gates as mentioned above might exist either. We will describe later how we ensure all dependency cycles are eliminated. For the time being, very temporarily, we heuristically decide the output variables of the detected ptype gates using Algorithm 2.

Algorithm 2 Determining the output variable of a ptype gate

```
foreach detected ptype gate with variables $v_0, v_1, \ldots, v_n$ in the clauses
    if there is a $v_k$ which is already the output of a ptype/atype gate
            having $v_{k'}(k' \neq k)$ as an input // conservatively reuse $v_k$
        Make $v_k$ with the smallest $k$ the output variable // multiple defs
    else if there is a $v_k$, not yet the output of any ptype/atype gate
        Make $v_k$ the output variable for the smallest $k$ // new definition
    else make $v_n$ (or any $v_k$) the output variable // multiple definitions
```

The heuristic in Algorithm 2 takes a conservative approach by making a first attempt to select a variable v_k that is the output of another gate with another variable $v_{k'}(k \neq k')$ as input. This creates another definition of v_k but directly prevents the way to have a dependency cycle by making $v_{k'}$ the output. This reuse of v_k perhaps also leaves other variables available to be made output of other gates. As a second attempt, Algorithm 2 selects a variable that is not the output of any other gate and so gets its new definition. As a third attempt, when multiple definitions is inevitable, arbitrarily variable v_n is made the output. Nevertheless, any decisions made at this stage are subject to change by the dependency cycle elimination procedure described later. Algorithm 2 has a time complexity of $\mathcal{O}(|F| \times \|c\|)$ and a memory complexity of $\mathcal{O}(|F| \times \|c\|)$.

Lemma 2 (Gates and Outputs) *Detection of atype and ptype gates and their output variables in Steps 1-2 of Algorithm 1 takes $\mathcal{O}(|F|^2 \times \|c\|) + |F| \times \|c\|^2)$ time and $\mathcal{O}(|F| \times \|c\|)$ memory.*

Proof Section 2.4 shows detection of atype gates and their outputs takes $\mathcal{O}(|F| \times \|c\|^2)$ time and $\mathcal{O}(|F| \times \|c\|)$ memory while just detection of ptype gates takes $\mathcal{O}(|F|^2 \times \|c\|)$ time and $\mathcal{O}(|F| \times \|c\|)$ memory. To identify outputs of ptype gates, Algorithm 2 takes $\mathcal{O}(|F| \times \|c\|)$ time $\mathcal{O}(|F| \times \|c\|)$ memory. □

3.5 Building Complete and Constrained Gate Graphs

We define terminologies for clauses and variables left out after gate detection.

Definition 5 (Core Clauses and Variables) Given a SAT problem instance $S \equiv \langle V, F \rangle$ and a gate graph G just obtained from Step 2 of Algorithm 1 (after gate detection in Step 1 and output variable identification in Step 2), a **core clause** $c \in F$ is such that $c \notin N.\phi$ for any gate node N in G and a **core variable** $v \in V$ is such that $v \neq N.o$ for any gate node $N \in G$ i.e. v is an independent variable. Assume C_c is the set of core clauses and C_v is the set of core variables.

We now describe the details of Steps 3–6 of Algorithm 1.

- **In Step 3 of Algorithm 1**, we handle **multiple definitions**. For each two gate nodes $N \neq N'$ with $N.o = N'.o = v$, make $N'.o = v'$ where v' is a new dependent variable to denote the output of one of the gates, and add a gate node $N'' = \langle v'', \mathsf{xnor}, \{v, v'\}, \emptyset, \mathsf{true}, \mathsf{null}, \mathsf{null}, \Omega, \mathsf{false}\rangle$ to the gate graph G to have a gate $v'' \leftarrow\!\dashv \mathsf{xnor}(v, v')$ with a constrained variable v''. This ensures the two definitions will produce the same value in a solution.
- **In Step 4 of Algorithm 1**, we deal with **core clauses**. For each core clause $c \in C_\mathsf{c}$ having $c \equiv l_1 \vee \ldots \vee l_n$, add to the gate graph G a gate node $N = \langle o, g, \rho, \emptyset, \mathsf{true}, \mathsf{null}, \mathsf{null}, c, \mathsf{false}\rangle$, where $N.o = v$ is a new constrained variable with $N.\kappa = \mathsf{true}$, $N.g = \mathsf{dg}\langle s(l_1), \ldots, s(l_n)\rangle$ denotes the gate type and the static sign parameters, and $N.\rho = \{v(l_1), \ldots, v(l_n)\}$ the input variables.
- **In Step 5 of Algorithm 1**, we deal with **core variables**. For each core variable $v \in C_\mathsf{v}$, add a gate node $N = \langle v, \mathsf{null}, \emptyset, \psi, \mathsf{false}, \mathsf{null}, \mathsf{null}, \Omega, \mathsf{true}\rangle$ to the gate graph G to denote an independent variable v.
- **In Step 6 of Algorithm 1**, we compute lists of dependent variables for each variable. For each gate node N, compute $N.\psi = \{v : v \ll N.o\}$.

It is straightforward to show that the gate graph obtained from Step 6 of Algorithm 1 is a complete and constrained gate graph, which is free from repeated computations, free from multiple definitions, but could have dependency cycles. Moreover, each gate node of the gate graph can be uniquely identified by the output variable, since no variables has multiple definitions.

Lemma 3 (Complete and Constrained Gate Graph) *Steps 3–6 of Algorithm 1 altogether take $\mathcal{O}(|F| \times \|c\| + |V|)$ time and memory. There are $\mathcal{O}(|F| + |V|)$ gate nodes in the resultant gate graph and $\mathcal{O}(|F| \times \|c\|)$ edges among gate nodes.*

Proof Below we analyse each of the four steps.

Step 3: There could be $\mathcal{O}(\frac{1}{3}|F|)$ atype and $\mathcal{O}(\frac{1}{2}|F|)$ ptype gates detected. So the number of variables having multiple definitions is $\mathcal{O}(\frac{1}{4}|F|)$, if two definitions per variable is assumed and each such case takes only $\mathcal{O}(1)$ time and memory.

Step 4: There could be at most $|F|$ core clauses. So the time and memory complexities to add all core clauses to the gate graph both are $\mathcal{O}(|F| \times \|c\|)$.

Step 5: There could be at most $|V|$ core variables. So the time and memory complexities of adding all core variables to the gate graph both are $\mathcal{O}(|V|)$.

Step 6: This will visit at most each literal in each clause. So the time and memory complexities for this are both $\mathcal{O}(|F| \times \|c\|)$.

We can sum the four steps to get the overall results. □

3.6 An Illustrative Example

In the following example, we show how a complete and constrained gate graph free from multiple definitions is obtained and a DAG can be built, assuming no dependency cycle exists in the gate graph. An expert reader can skip this example.

Example 3 (DAG) The formula F below has no unit clause, no repeated clause, no repeated literal in any clause, and no clause with opposite literals.

$g_0 \vee \overline{v_1} \vee \overline{v_2}$	$\overline{g_0} \vee v_2 \vee v_3$	$\overline{g_1} \vee \overline{v_3} \vee \overline{v_4}$	$g_2 \vee \overline{g_0} \vee \overline{g_1}$	$v_0 \vee g_0$
$\overline{g_0} \vee v_1$	$g_0 \vee \overline{v_2}$	$\overline{g_1} \vee v_3 \vee v_4$	$\overline{g_2} \vee g_0$	
$\overline{g_0} \vee v_2$	$g_0 \vee \overline{v_3}$	$g_1 \vee \overline{v_3} \vee v_4$	$\overline{g_2} \vee g_1$	
		$g_1 \vee v_3 \vee \overline{v_4}$		

Gates $g_0 = \mathsf{and}(v_1, v_2)$, $g_0 = \mathsf{or}(v_2, v_3)$, $g_1 = \mathsf{xor}(v_3, v_4)$, $g_2 = \mathsf{and}(g_0, g_1)$ are detected. So the gate graph G has 4 gate nodes for these four gates.

1. There are multiple definitions of g_0. So we rename one of them as g_0' and add a new gate $e \leftarrowtail \mathsf{xnor}(g_0, g_0')$ to G. We also make e a constrained variable.
2. The set of core clauses is $C_\mathsf{c} = \{v_0 \vee g_0\}$. So we add to the gate graph G a gate $c \leftarrowtail dg_{\langle\mathsf{true},\mathsf{true}\rangle}(v_0, g_0)$. We also make c a constrained variable.
3. The set of core variables is $C_\mathsf{v} = \{v_0, \ldots, v_4\}$. So we add these as independent variables to the gate graph G. All SAT variables are now in the gate graph.

After the three steps above, we obtain a complete and constrained gate graph that has $\{v_0, v_1, v_2, v_3, v_4\}$ as independent variables, $\{g_0, g_0', g_1, g_2, c, e\}$ as dependent variables, and $\{c, e\}$ as constrained variables. There is no dependency cycle among the dependent variables and there is no multiple definition. We draw a DAG in Fig. 1 where g_2 could be connected to g_0 instead of g_0'. Notice that g_2, g_1, and v_4 are not required during search as they can take any arbitrary values to satisfy the constrained circuit and hence the CNF SAT problem instance.

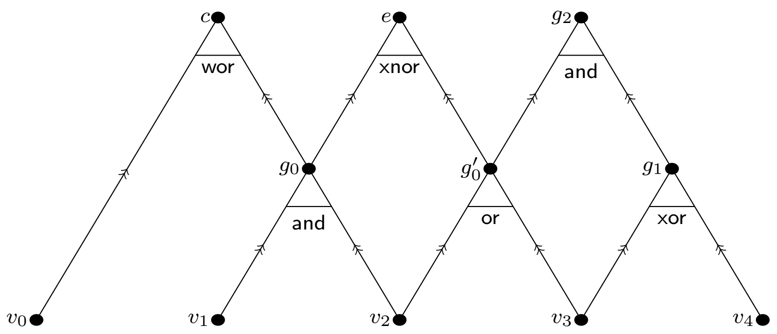


Fig. 1 A DAG for a complete and constrained gate graph where c and e are constrained variables; g_0, g_0', g_1 and g_2 are depedent variables. Variables g_2, g_1, and v_4 are not required during search as they can take any arbitrary values to satisfy the constrained circuit.

3.7 Removing Dependency Cycles

Dependency cycles must be eliminated in Step 7 of Algorithm 1 before obtaining a constrained circuit. Below we define the variables that have dependency troubles.

Definition 6 (Dependency Troubles) A dependent variable v has a **dependency trouble**, if there is a dependency cycle involving v, or if there is another dependent variable $v' \neq v$ such that $v \lll v'$ and v' has dependency troubles. Fig. 2 shows an example, where v_4 has a dependency trouble since v_2 is involved in a cycle, although v_4 has no cyclic dependency.

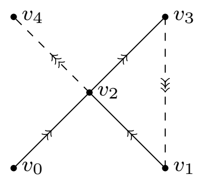


Fig. 2 Variables v_1, v_2, v_3, v_4 have dependency troubles, but v_0 has not. Solid lines are immediate dependencies and dashed lines are transitive dependencies. Arrow directions are from the input to the output.

Algorithm 3 identifies dependent variables that have dependency troubles. Procedure identifyVariablesInTrouble first assumes each dependent variable has dependency troubles. So for each dependent variable v, InputsInTrouble$[v]$ has the value equal to the number of inputs to the gate. Then, a recursive visit starts from each independent variable v to the dependent variables that immediately or transitively depend on v. In the recursive procedure visitDependentsRecursively, if no input variable has dependency troubles, then the output variable has no dependency trouble either. Also, if any input variable has dependency troubles, then the output variable also has dependency troubles. At the end of Algorithm 3, we have the dependent variables with non-zero InputsInTrouble as those having the dependency troubles. We will clear the dependency troubles.

Algorithm 3 Detecting variables having dependency troubles

```
proc identifyVariablesInTrouble()
   foreach dependent variable $v$
      InputsInTrouble$[v] = |\{v' : v \ll v'\}|$
   foreach independent variable $v$
      visitDependentsRecursively$(v)$

proc visitDependentsRecursively$(v)$
   foreach $v'$ that immediately depends on $v$
      decrease InputsInTrouble$[v']$ by 1
      if InputsInTrouble$[v'] = 0$
         visitDependentsRecursively$(v')$
```

We analyse the complexities of detecting variables having dependency troubles.

Lemma 4 (Dependency Trouble Complexity) *Algorithm 3 runs in $\mathcal{O}(|F| \times \|c\| + |V|)$ time and memory to detect variables having dependency troubles.*

Proof The complete gate graph obtained from Step 6 of Algorithm 1 has $\mathcal{O}(|F| + |V|)$ gate nodes and $\mathcal{O}(|F| \times \|c\|)$ edges (can be understood from ρ or ψ of the gate nodes). The recursive procedure visits the gate nodes via the edges. □

To break dependency cycles, we first take a preferred approach of changing the output variables of the ptype gates. If there still exist dependency cycles then we take a guaranteed approach of separating the output of a gate from its input. We describe these approaches below.

Preferred Approach: Rearranging ptype *Gate Outputs*

Figure 3 shows an example of breaking a dependency cycle where the output variable of an xor gate is swapped with an independent variable. At the left side of the figure, notice that v_2 is the output of an xor gate with inputs v_0 and v_1. Variable v_3 immediately depends on v_2 but variable v_0 transitively depends on v_3. Therefore, there is a dependency cycle $\langle v_0, v_2, v_3, \ldots, v_0 \rangle$. This cycle could be easily broken as shown in the right side of the figure if an input v_1, which is an independent variable in the left side figure, becomes the output of the xor gate and v_2 becomes an input.

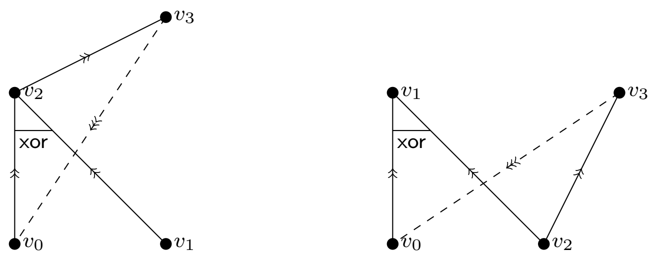


Fig. 3 Breaking a cycle (left) by swapping an input and the output of an XOR gate (right). Solid lines are immediate dependencies and dashed lines are transitive dependencies. Arrow directions are from the input to the output.

In Fig. 3, we make an independent variable v_1 the new output of the gate so that multiple definitions of v_1 cannot happen, and v_2 becomes the new independent variable. This keeps the numbers of dependent and independent variables unchanged. Unfortunately, the output of a gate can be involved in more than one cycle. So it might be swapped back to break another cycle. In order to avoid back and forth swapping, we allow only one swapping for each variable. Algorithm 4 describes the procedures required to break cycles in this way. Notice that before changing the output of the gate, we need to call unvisitDependentsRecursively, which essentially reverts the effects of procedure visitDependentsRecursively described in Algorithm 3. Moreover, we need to call visitDependentsRecursively again after changing the output of the gate. Procedure changeGateOutput in Algorithm 4 changes the gate graph by removing the old input-output relation between the two variables swapped, making changes to the other inputs of the gate, and adding the new input-output relation of the two variables swapped.

We analyse the complexities of breaking the dependency cycles by the preferred method of rearranging outputs of the ptype gates.

Lemma 5 (Preferred Cycle Breaking) *Algorithm 4 in its main loop conditionally break cycles taking $\mathcal{O}((|F| + |V| + \|c\|) \times |V|)$ time and $\mathcal{O}(|V|)$ memory.*

Algorithm 4 Break dependency cycles by changing outputs of ptype gates

```
proc changeOutputsToBreakCycles()
    foreach variable $v$ in the gate graph,
        markedAsSwapped[$v$] = false
    foreach $v$ having InputsInTrouble[$v$] > 0
            and $v$ is the output of a ptype gate
        if $v'$ : $v \ll v'$ is an independent variable
                and markedAsSwapped[$v'$] = false
            unvisitDependentsRecursively($v'$)
            changeGateOutput($v$, $v'$)
            markedAsSwapped[$v'$] = true
            visitDependentsRecursively($v$)

proc unvisitDependentsRecursively($v$)
    foreach $v' \ll v$ // $v'$ is a dependent of $v$
        increase InputsInTrouble[$v'$] by 1
        if InputsInTrouble[$v'$] == 1
            unvisitDependentsRecursively($v'$)

proc changeGateOutput($v$,$v'$) // $v$ to $v'$
    remove $v \ll v'$ from the gate graph
    swap inputs of $v$ and $v'$ in gate graph, so
        $o$, $\rho$, $\psi$ of related gate nodes change
    add $v' \ll v$ to the gate graph
```

Proof The cycle is broken since an independent variable, if not tried before, is made the output of the gate. Each of the $\mathcal{O}(|V|)$ variables can be swapped once in Algorithm 4, provided they are inputs or outputs of the ptype gates. For each swapping, the recursive procedures visit variables that (immediately or transitively) depend on the variables swapped. In general, it is difficult to ascertain the size of the transitive closure of each variable, but at the worst case it is $\mathcal{O}(|F|+|V|)$. Changing the output and the dependency relations take $\mathcal{O}(||c||)$ time. The memory complexity comes from the flags to denote whether a variable is swapped. □

Guaranteed Dependency Cycle Removal

An extreme but guaranteed way to break all cycles is to bring the clausal representations back and then to treat them as core clauses. We do not take this approach. In this article, we show a new technique in Fig. 4 to separate the output variable of a gate from the input variables and thus break cycles. Assume v_2 is a dependent variable that is part of two dependency cycles $\langle v_2, v_3, \ldots, v_0, v_2\rangle$ and $\langle v_2, v_4, \ldots, v_1, v_2\rangle$. We replace v_2 with two variables v_2 and v_2'. As we see before separation, we have $v_3 \ll v_2$, $v_4 \ll v_2$, $v_2 \ll v_0$, and $v_2 \ll v_1$, but after separation we get $v_3 \ll v_2'$, $v_4 \ll v_2'$, $v_2 \ll v_0$ and $v_2 \ll v_1$. We need a constrained variable e with a gate $e \hookleftarrow \mathsf{xnor}(v_2, v_2')$ to ensure v_2 and v_2' have the same value in case of a satisfying assignment.

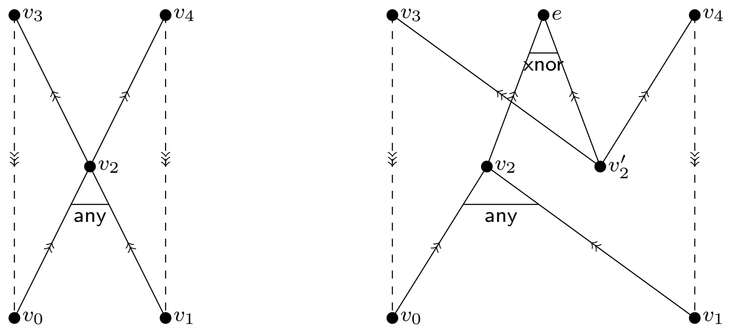


Fig. 4 Breaking cycles (left) by separating output v_2 from inputs v_0 and v_1 (right). Solid lines are immediate dependencies and dashed lines are transitive dependencies. Arrow directions are from the input to the output.

Algorithm 5 Break cycles by separating the output from the inputs

```
proc separateInputOutputsToBreakCycles()
  while there exists a dependent variable $v'$ having InputsInTrouble$[v'] > 0$
    Select the variable $v$ involved in the maximum number of cycles
    Create an independent variable $v'$ with
      a gate node $\langle v', \mathsf{null}, \emptyset, \emptyset, \mathsf{false}, \mathsf{null}, \mathsf{null}, \Omega, \mathsf{false}\rangle$
    foreach $v'' \ll v$, remove $v'' \ll v$ and make $v'' \ll v'$
    Add a gate $e \leftrightarrow \mathsf{xnor}(v, v')$ with a new output variable $e$ and
      a gate node $\langle e, \{v, v'\}, \emptyset, \mathsf{true}, \mathsf{null}, \mathsf{null}, \Omega, \mathsf{false}\rangle$
```

Algorithm 5 describes the required procedure that breaks all the remaining cycles after running Algorithm 4. Separation of the output of a gate from the inputs cause addition of one independent and one dependent variable. One way to minimise the number of variables to be added is to select for separation an output variable that is involved in the maximum number of cycles. This is what is done in the algorithm. We use Algorithm 6 to find the variables involved in the maximum number of cycles. A recursive depth-first visit starts from a variable having dependency troubles. If a cycle is detected, it is counted for all variables in the cycle. To ensure uniqueness of the cycles, every recursive visit starts from a variable that is not part of any previously detected cycle.

Algorithm 6 Finding variables involved in the maximum number of cycles.

```
proc findVarInMostCycles()
  foreach dependent variable $v$ having InputsInTrouble$[v] > 0$
    VarInCycleCount$[v]$ = 0 // initially not in any cycle
  foreach dependent variable $v$ having InputsInTrouble$[v] = 0$
    findCyclicPath($v$, $\mathcal{P}$) // $\mathcal{P}$ is an empty list of variables on the path
  return a variable $v$ having the maximum value of VarInCycleCount$[v]$

proc findCyclicPath($v$, $\mathcal{P}$)
  if $v$ is already in the list $\mathcal{P}$ // cycle detected
    Increase VarInCycleCount$[v]$ by one // v is in the cycle
    Increase VarInCycleCount$[v']$ by one foreach $v'$ that is after $v$ in $\mathcal{P}$
  else // no cycle detected yet
    Append $v$ to path $\mathcal{P}$ traversed so far
    foreach $v'$ that $v$ depends on and InputsInTrouble$[v'] > 0$
      findCyclicPath($v'$, $\mathcal{P}$)
```

Lemma 6 (Breaking Dependency Cycles) *Algorithms 5 and 6 do remove all dependency cycles. Assume there are $\|F\|$ variables involved in the dependency cycles in the complete gate graph obtained from Step 6 of Algorithm 1. Algorithm 6 takes $\mathcal{O}(\|F\|^2)$ time and $\mathcal{O}(\|F\|)$ memory. So Algorithm 5 takes $\mathcal{O}(\|F\|^3 + \|c\| \times |F|^2)$ time and $\mathcal{O}(\|F\| + \|c\|)$ memory.*

Proof The removal of each dependency cycle is guaranteed since a new independent variable is created and the variables that immediately depended on the output of the gate now immediately depends on the new variable. Algorithm 6 starts finding a cycle from each such variable v and recursively visits the other variables but once for each v. So it takes $\mathcal{O}(\|F\|^2)$ time and $\mathcal{O}(\|F\|)$ memory. Algorithm 5 in each iteration calls Algorithm 6 once and use $\mathcal{O}(\|c\|)$ time and $\mathcal{O}(\|c\|)$ memory to break the cycle. Assuming at least two variables are needed to create a cycle, there could be at most $\frac{1}{2}\|F\|$ cycles. So the time complexity of breaking all cycles by Algorithm 5 is $\mathcal{O}(\|F\|^3 + \|c\| \times \|F\|^2)$. Additional memory needed is $\mathcal{O}(\|F\| + \|c\|)$ to detect cycles and to break them. $\square$

3.8 Constrained Circuits for CNF SAT

The gate graph G obtained from Step 7 of Algorithm 1 represents a constrained circuit. It essentially captures all necessary properties of a SAT problem instance. Below we describe various types of gates in such a gate graph.

Definition 7 (Gate Node Types) Assume G be a gate graph obtained from Step 7 of Algorithm 1. Since there is no multiple definitions, each gate node N in G is uniquely identified by the output variable $N.o$. We define the following types of gate nodes and as such corresponding types of variables.

1. **Independent Variables**: $V_{\mathrm{I}} = \{N.o : N.g = \mathsf{null}\}$ that do not depend on any other variables. In Fig. 1, the independent variables are $V_{\mathrm{I}} = \{v_0, v_1, v_2, v_3, v_4\}$.
2. **Constrained Variables**: $V_{\mathrm{C}} = \{N.o : N.g \neq \mathsf{null} \wedge N.\kappa = \mathsf{true} \wedge N.\psi = \emptyset\}$ that are terminal variables and are flagged as constrained variables. In Fig. 1, such constrained variables are $V_{\mathrm{C}} = \{c, e\}$. Their values must be true.
3. **Unconstrained Variables**: $V_{\mathrm{U}} = \{N.o : N.g \neq \mathsf{null} \wedge N.\kappa = \mathsf{false}\}$ that are either terminal variables without being flagged as constrained or are intermediate variables. In Fig. 1, such unconstrained variables are $V_{\mathrm{U}} = \{g_0, g_0', g_1, g_2\}$.
4. **Imported SAT variables**: $V_{\mathrm{S}} = \{N.o : N.\sigma = \mathsf{true}\}$ that are the SAT variables coming as the core variables or as the output of the gates detected. In Fig. 1, the imported SAT variables are $V_{\mathrm{S}} = \{v_0, v_1, v_2, v_3, v_4, g_0, g_1, g_2\}$.
5. **Required Variables**: $V_{\mathrm{R}} = \{N.o \notin V_{\mathrm{C}} : \exists_{N'.o \in V_{\mathrm{C}}}[N'.o \lll N.o]\}$ that are such that some constrained variables depend on them. In Fig. 1, $V_{\mathrm{R}} = \{g_0, g_0'\}$.
6. **Deferrable Variables**: $V_{\mathrm{D}} = \{N.o \notin V_{\mathrm{C}} : \neg\exists_{N'.o \in V_{\mathrm{C}}}[N'.o \lll N.o]\}$ that are such that no constrained dependent variable depends on them. In Fig. 1 of Example 3, such variables are $V_{\mathrm{D}} = \{g_1, g_2, v_4\}$.

The following summarised change log of various types of gate nodes is useful.

1. Core variables are first made the independent variables in Step 5 of Algorithm 1 with $\sigma = \mathsf{true}$. During changing the outputs of the ptype gates to eliminate dependency cycles, Algorithm 4 might change some independent variables with $\sigma = \mathsf{true}$ to unconstrained variables with $\sigma = \mathsf{true}$ and vice versa. Then, to break remaining dependency cycles in a guaranteed way, Algorithm 5 might add some independent variables with $\sigma = \mathsf{false}$.
2. Constrained variables are initially obtained from the core clauses. Then, to handle multiple definitions, some new constrained variables might be created. Also, during breaking the dependency cycles in a guaranteed way, further constrained variables are created by Algorithm 5.
3. Unconstrained variables are obtained with $\sigma = \mathsf{true}$ from the output variables of the gates detected. Some unconstrained variables are obtained with $\sigma = \mathsf{false}$ by

renaming the output variable of a gate while handling the multiple definitions. During changing the outputs of the ptype gates to eliminate dependency cycles, Algorithm 4 might change some unconstrained variables with $\sigma = \mathsf{true}$ to independent variables with $\sigma = \mathsf{true}$ and vice versa.

We prove the correctness of the constrained circuits for CNF SAT instances.

Theorem 1 (Correctness of Constrained Circuit) *Assume $\Sigma \equiv \langle C, W\rangle$ is a constrained gate graph obtained by Steps 1–7 of Algorithm 1 for a given SAT instance $S \equiv \langle V, F\rangle$. There is a solution $\overrightarrow{I}$ to Σ iff there is a solution $\overrightarrow{V}$ to S.*

Proof We address all related issues of the proof point by point below:

1. Each SAT variable $v \in V$ is either an independent variable in I or a dependent variable in D. Initially, SAT variables that become the output of the detected gates are in V_{U} and that are not output of any gates are in V_{I} as core variables. Only Algorithm 4 then moves some variables from V_{I} to V_{U} or vice versa, but these changes happen only among the imported SAT variables that have $\sigma = \mathsf{true}$. This proves the inclusiveness of all SAT variables.
2. For each SAT clause $c \in F$, either there is a logic gate having c in the gate's clausal pattern or there is a constrained variable in V_{C} as, not being an output of any gate, c is a core clause. For each constrained variable, the search algorithm ensures it's value is true in the solution (if any) i.e. the clause is satisfied. For a clause in the clausal representation of a detected gate, the invariant engine ensures the functional value of the gate is equal to the value of the output variable upon all queries and as such all clauses in the clausal pattern are satisfied. This proves the inclusiveness of all SAT clauses.
3. While clauses are part of the clausal patterns representing detected gates, outputs of two gates can be the same SAT variable and so the value of the two outputs must be the same in a solution. As we handle the multiple definitions, we make sure it does happen in the solution as we define an xnor gate with the two outputs, renaming one of them. The renamed variable is in V_{U} and a newly created constrained variable for the output of the xnor is in V_{C}.
4. After gates are detected, cyclic dependencies might arise as shown in Example 1. The dependency cycles hinder computation of the gate outputs by the invariant engine and so must be eliminated. Algorithm 5 as per Lemma 6 does eliminate all cycles. An xnor gate and a constrained variable in V_{C} ensure the new variable created to break the cycle and the output variable of the gate involved in the cycle have the same value in a solution.

These prove each solution of the SAT problem instance is captured by the constrained circuit, or vice versa. If a solution is found for one, we can get a solution for the other. The back and forth translations from $\overrightarrow{I}$ to $\overrightarrow{V}$ and vice versa are trivial because dependent variables have their definitions. □

We now analyse the time and memory complexities of Steps 1–7 of Algorithm 1.

Theorem 2 (Complexities of Model Building) *Steps 1–7 of Algorithm 1 altogether take $\mathcal{O}(|F|^2 + |V|^2)$ time and $\mathcal{O}((|F| + \|c\| + |V|)$ memory, where $\|F\|$ is the number of dependency cycles in the gate graph obtained from Step 6 and we assume $\|F\|$ and $\|c\|$ are very small compared to $|F|$.*

Proof Below we show the time and memory complexities of key steps.

Steps	Lemma	Time	Memory
1–2	2	$\mathcal{O}(\lvert F\rvert^2 \times \lVert c\rVert) + \lvert F\rvert \times \lVert c\rVert^2)$	$\mathcal{O}(\lvert F\rvert \times \lVert c\rVert)$
3–6	3	$\mathcal{O}(\lvert F\rvert \times \lVert c\rVert + \lvert V\rvert)$	$\mathcal{O}(\lvert F\rvert \times \lVert c\rVert + \lvert V\rvert)$
7	4	$\mathcal{O}(\lvert F\rvert \times \lVert c\rVert + \lvert V\rvert)$	$\mathcal{O}(\lvert F\rvert \times \lVert c\rVert + \lvert V\rvert)$
7	5	$\mathcal{O}((\lvert F\rvert + \lvert V\rvert + \lVert c\rVert) \times \lvert V\rvert)$	$\mathcal{O}(\lvert V\rvert)$
7	6	$\mathcal{O}(\lVert F\rVert^3 + \lVert c\rVert \times \lVert F\rVert^2)$	$\mathcal{O}(\lVert F\rVert + \lVert c\rVert)$

The total complexities are obtained by adding the complexities of the steps. □

3.9 Propagate Equivalence Relationships

This is optional Step 8 of Algorithm 1. In the gate graph obtained after running Steps 1–7 of Algorithm 1, if there is a gate $v_0 \hookleftarrow \mathsf{eq}(v_1)$, every occurrence of v_0 as input of any other gates can then be replaced by v_1. Also, such replacement of v_0 with v_1 is possible when there exist two gates such that $v_0 \hookleftarrow \mathsf{not}(v_2)$ and $v_2 \hookleftarrow \mathsf{not}(v_1)$. Simplification is also possible just for $v_0 \hookleftarrow \mathsf{not}(v_1)$. These simplifications are done in Procedures propagateEqRels and propagateNotRels in Algorithm 7. Procedures replaceEq and replaceNot perform the replacement in each gate. However, as these replacements take place, some simplification could be obtained using the semantic of the gate. For example, if v_0 and v_1 are both inputs of an and or an or gate, then one of v_0 and v_1 could be safely removed, while they can both be removed from an xor or xnor gate with more than 4 inputs.

When one variable is replaced by another variable in the inputs of a gate, the functional value of the gate might be fixed. For example, replacement of v_0 by v_1 in the gate $v_2 \hookleftarrow \mathsf{xor}(v_1, v_0)$ will result into $v_2 = \mathsf{false}$. As v_2's value is now known, this could be propagated through the gate graph and Procedure propagateValue in Algorithm 8 does this. In Procedure propagateValue, we actually show how this value propagation can lead to fixation of values of some other variables and also how a particular type of gate can change to another type of gate.

Below we show the soundness and complexities of the above simplification.

Lemma 7 (Equivalence Simplification) *Algorithms 7 and 8 are satisfiability preserving. For each pair of variables that are equivalent or opposite, the time complexity is $\mathcal{O}(\lvert F\rvert \times \lVert c\rVert)$ with no additional memory.*

Proof The simplifications are based on the specific semantics of the logic gates. These constraint based simplifications are very similar to the UCP and equivalence literal elimination in CNF SAT. For each pair of equivalent variables v_0 and v_1, v_0 is replaced by v_1 in the definition of each variable v_2 that depends on v_0. This could trigger a chain of replacements and the chain effect propagates through the DAG. So the number of gate nodes that could be visited is $\mathcal{O}(\lvert F\rvert)$, which is the number of dependent variables. Now to perform the reasoning based on the specific gate semantic, we may need to visit each input in the gate. So the time complexity becomes $\mathcal{O}(\lvert F\rvert \times \lVert c\rVert)$ for each pair of equivalent variables. The same analysis holds when v_0 and v_1 are opposite of each other. □

Algorithm 7 Propagate equivalence of variables on a gate graph.

```
proc propagateEqRels()
    while true
        if exists $v_0 \leftarrowtail eq(v_1)$
            foreach $v_2 \ll v_0$
                replaceEq$(v_0, v_1, v_2)$
        else if exists $v_0 \leftarrowtail not(v_2)$
            if exists $v_2 \leftarrowtail not(v_1)$
                foreach $v_3 \ll v_0$
                    replaceEq$(v_0, v_1, v_3)$
        else break

// Replace $v_0$ with $v_1$ in $v_2$
proc replaceEq$(v_0, v_1, v_2)$ // in $v_2$
    if $v_1$ not an input of $v_2$ already
        Replace $v_0$ with $v_1$
    else if $v_2 \leftarrowtail$ cg/dg$_{\langle s_1,\ldots,s_n\rangle}(\ldots)$
            and $s$ for $v_0$ and $v_1$ are opposite
        propagateValue($v_2$, false/true)
    else if $v_2 \leftarrowtail$ xor/xnor$(v_0, v_1)$
        propagateValue($v_2$, false/true)
    else if $v_2 \leftarrowtail$ xor/xnor$(\ldots)$
        Remove $v_0$ and $v_1$ from $v_2$'s input
        if $v_2$ has only one input $v_3$
            Make $v_2 \leftarrowtail$ eq/not$(v_3)$
    else // and/nand/or/nor/cg/dg
        Remove $v_0$ from $v_2$'s input
        if $v_2$ has only one input $v_3$
            if and/or/cg/dg
                Make $v_2 \leftarrowtail$ eq$(v_3)$
            else if nand/nor
                Make $v_2 \leftarrowtail$ not$(v_3)$

proc propagateNotRels()
    while exists $v_0 \leftarrowtail not(v_2)$
        foreach $v_3 \ll v_0$
            replaceNot$(v_0, v_1, v_3)$

// Replace $v_0$ with $\overline{v_1}$ in $v_2$
proc replaceNot$(v_0, v_1, v_2)$
    if $v_1$ not an input of $v_2$ already
        Assume $v_2 \leftarrowtail g(\ldots, v_0, \ldots)$
        if $g$ = xor/xnor
            Replace $v_0$ with $v_1$
            $g \leftarrow$ xnor/xor
        else if $g$ = cg/dg
            Replace $v_0$ with $v_1$, flip $s$
        else if $g$ = and/nor/or/nand
            $g \leftarrow$ cg if $g$ = and/nor
            $g \leftarrow$ dg if $g$ = or/nand
            Replace $v_0$ with $v_1$, flip $s$
    else if $g$ = and/or/nand/nor
        $b \leftarrow$ false/true/true/false
        propagateValue($v_2$, $b$)
    else if $g$ = xor/xnor with 2 inputs
        propagateValue($v_2$, true/false)
    else if $g$ = xor/xnor
        Remove $v_0$ and $v_1$ from $g$
        if $g$ has one input, $g \leftarrow$ not/eq
    else if $g$ = cg/dg
        if $v_0$ and $v_1$ has the same sign
            propagateValue($v_2$, false/true)
        else // has opposite sign
            remove $v_0$ from $g$
            if $v_2$ has one input, $g \leftarrow$ eq
```

The variables that are replaced by their equivalent literals are marked as **replaced** in the gate graph using ϵ of a gate node. Moreover, the variables whose values are **fixed** are marked so using ν of a gate node. These replaced and fixed variables are not needed during search. The values of the replaced variables can be computed from the equivalent literals once a solution is found.

3.10 Removing Variables from DAG

This is optional Step 9 of Algorithm 1. Look at the variables g_1, g_2, and v_4 in Fig. 1. These variables are not strictly required during search. Explicit constraints are defined only on c and e. We need their values during search and for that we also need to compute the other variables such as v_0, v_1, v_2, v_3, g_0, and g_0', since c and e depend on them. Once we satisfy the constraints on c and e, we can have any value for v_4 and then g_1 and g_2 can just be computed using the gate functions. Below we formally define the variables that are not strictly needed during search.

Lemma 8 (Remove Deferable Variables) *Given a gate graph obtained from Step 7 or 8 of Algorithm 1, we can temporarily remove any gate node N such that $N.o$ is a deferrable variable. We do not need these variables during search.*

Algorithm 8 Propagate fixed value of a variable on a gate graph.

```
proc propagateValue($v, b$) when $b$ = true
  Mark $v$ to be fixed with true
  foreach $v_0 \hookleftarrow g(\ldots, v, \ldots)$
    if $g$ = and/nand
      Remove $v$ from $v_0$'s input
      if $v_0$ has only one input
        $g \leftarrow$ eq/not if $g$ = and/nand
    else if $g$ = or/nor
      $b' \leftarrow$ true/false if $g$ = or/nor
      propagateValue($v_0, b'$)
    else if $g$ = xor/xnor
      Remove $v$ from $v_0$'s input
      if $v_0$ has only one input
        $g \leftarrow$ not/eq if $g$ = xor/xnor
      else // more than one input
        $g \leftarrow$ xnor/xor if $g$ = xor/xnor
    else if $g$ = eq/not
      $b' \leftarrow$ true/false if $g$ = eq/not
      propagateValue($v_0, b'$)
    else if $g$ = cg/dg$_{\langle s_1,\ldots,s_n\rangle}$
      if $s$ = true/false for $v$ in cg/dg
        Remove $v$ from $v_0$'s input
        if $v_0$ has only one input $v_1$
          if $s$ is true/false for $v_1$
            $g \leftarrow$ eq/not
      else// $s$ = false/true for $v$ in cg/dg
        $b' \leftarrow$ false/true if $g$ = cg/dg
        propagateValue($v_0, b'$)
```

```
proc propagateValue($v, b$) when $b$ = false
  Mark $v$ to be fixed with false
  foreach $v_0 \hookleftarrow g(\ldots, v, \ldots)$
    if $g$ = or/nor
      Remove $v$ from $v_0$'s input
      if $v_0$ has only one input
        $g \leftarrow$ eq/not if $g$ = or/nor
    else if $g$ = and/nand
      $b' \leftarrow$ false/true if $g$ = and/nand
      propagateValue($v_0, b'$)
    else if $g$ = xor/xnor
      Remove $v$ from $v_0$'s input
      if $v_0$ has only one input
        $g \leftarrow$ eq/not if $g$ = xor/xnor
      else // more than one input
        $g \leftarrow$ xor/xnor if $g$ = xor/xnor
    else if $g$ = eq/not
      $b' \leftarrow$ false/true if $g$ = eq/not
      propagateValue($v_0, b'$)
    else if $g$ = cg/dg$_{\langle s_1,\ldots,s_n\rangle}$
      if $s$ = false/true for $v$ in cg/dg
        Remove $v$ from $v_0$'s input
        if $v_0$ has only one input $v_1$
          if $s$ is true/false for $v_1$
            $g \leftarrow$ eq/not
      else// $s$ = true/false for $v$ in cg/dg
        $b' \leftarrow$ false/true if $g$ = cg/dg
        propagateValue($v_0, b'$)
```

Proof These variables only have the constraints that their values must be equal to the functional values of the respective gates. There is no constraint that their values must be true to obtain a solution. Once the constrained variables all have their values true, the values of these variables could be computed from other variables that they depend on or from random selection if these are independent variables. Removal of these variables thus improve the model efficiency. □

3.11 More Simplification Before Gate Graphs

Even before Step 1 of Algorithm 1, we detect cg or a dg gates only from the clauses that opposite literals. This is because $v \wedge \text{false} = \text{false}$ and $v \vee \text{true} = \text{true}$. Once we can fix the value of a dependent variable in this way, we perform UCP with that on the formula. This is not discussed before for clarity.

4 Our Adaptation of CNF Local Search to Circuits

4.1 Local Search Algorithms

To test the effectiveness of the gate based constraint circuit models, we use the local search framework described in Algorithm 9. Starting from a random assignment, in each iteration of the search, it flips a variable selected by a given variable selection heuristic. To avoid recently made flips, it uses a tabu metaheuristic. It keeps track of the length of the run of non-improving moves (plateau scenario) compared to the global best assignment, and performs restart from scratch if the run length exceeds a given threshold. So The main issue in the algorithm is the variable selection heuristic to be used to select a variable for flipping.

Algorithm 9 Local Search Framework

```
Initialise all independent variables in $I = V_I$ randomly
Compute dependent variables $D = V_C \cup V_U$ using the DAG
currViolation = Number of variables in $V_C$ with value false
bestViolation = currViolation, plateauLength = 0
while not timeout
    if currViolation =0 then return solution found
    if currViolation ≥ bestViolation then ++plateauLength
    else bestViolation = currViolation, plateauLength = 0
    if plateauLength > maxPlateauLength //restart from scratch
        Assign random values to all variables, plateauLength = 0
    else // using a heuristic, select a variable to flip
        Select and flip a non-tabu variable according to
            AdaptNovelty+ or CCAnr variable selection heuristic
    Propagate changes using the DAG and compute currViolation
```

Exploring the literature of local search for CNF SAT, we found representatives of two prominent families of local search algorithms namely AdaptNovelty+ (Hoos and Tompkins 2007) and CCAnr (Cai et al. 2015). We adapt the AdaptNovelty+ and the CCAnr variable selection heuristics to cope with our constrained circuit models. There exists other solvers that use hybridisation of local search and systematic search algorithms. In this work, our scope is to evaluate local search algorithms with constrained circuits. So we choose these two solvers since they are solely based on local search algorithms. Below we briefly discuss these two solvers.

AdaptNovelty+ can be considered as a representative of the WalkSAT family of SLS algorithms for SAT. The walk probability used in a WalkSAT algorithm has an important impact on the performance (Hoos and Tompkins 2007; Hoos et al. 2002; Hoos and Stützle 2000) and the optimal value of walk probability is different for each problem (Hoos and Stützle 2000). Moreover, the performance of a WalkSAT algorithm can fluctuate significantly even with a minor change in the walk probability and this sensitivity increases proportionally with the size and the hardness of the problem (Hoos et al. 2002). For these reasons, different approaches have been proposed to automatically adjust the value of the walk probability prior to perform or during a WalkSAT search (Hoos et al. 2002; McAllester et al. 1997; Patterson and Kautz 2001). The most successful attempt is an adaptive mechanism introduced by Hoos et al. (2002). The idea of the adaptive mechanism is closely related to the work by Battiti and Tecchiolli (1994) and is to use a high walk probability value in situations where the search cannot escape local optima.

CCAnr is a recent SLS solver by Cai et al. (2015) for non-random SAT. It uses the CC technique to avoid revisitation scenarios. In CCAnr, each variable has a **configuration** that comprises its neighbouring variables and the time when they have been flipped most recently. In CCAnr, a variable is not flipped again until at least one of its neighbouring variables has been flipped. CCAnr has two local search mode: an overall one mostly based on greediness and a focussed one mostly based on controlled diversification. CCAnr performs better than the other state-of-the-art SLS solvers on the combinatorial and application benchmarks instances.

While adapting the CNF-based local search algorithms to gate-based constrained circuit models, we deal with a number of issues. Below we discuss these issues and describe how we address these.

Clauses Replaced by Constrained Variables. To replace the concept of selection of a falsified clause in SAT, in constrained circuits, we use the constrained variables that are false. Also, we use the constrained variables to compute the make, break, and score for each potential assignment.

Redefining Neighbour Variables. Variables appearing in the same clause are neighbours in the CNF-based CCAnr (Cai et al. 2015). In constrained circuits, for CCAnr, we define two independent variables become neighbours if there exists a constrained dependent variable that transitively depends on both of the independent variables i.e. $\mathsf{neighbour}(v, v') = v \in V_{\mathrm{I}} \wedge v' \wedge V_{\mathrm{I}} \wedge \exists_{v'' \in V_{\mathrm{C}}}[v'' \lll v \wedge v'' \lll v'']$. For an independent variable v, we use $\mathsf{neighbour}(v) = \{v' : \mathsf{neighbour}(v, v')\}$ to denote the set of neighbouring variables of v. Under this definition, the number of neighbours of a variable could be much larger in constrained circuits than in CNF SAT. Because of this, the CC heuristic may loose its effectiveness in tackling the revisitation issue of a local search as a variable can soon become available for selection.

Using Tabu Technique. The tabu technique (Mazure et al. 1997) is the older alternative of the CC technique. Algorithm 9 uses the tabu technique with tabu tenure 5 in our experiments. This common tabu tenure was selected after some preliminary runs although tabu tenure is often decided in an instance specific way. In a gate based constrained circuit model, the number of independent variables is smaller than that in a CNF based model. So the revisitation problem is more likely to happen. The tabu technique might be effective when the CC technique in CCAnr is less effective because of the large numbers of neighbours of each variable. The tabu technique might help AdaptNovelty+ as well.

Performing Search Restarts. In the gate-based constrained circuit model, the number of variables in V_{C} is in general much smaller than the number of clauses in the CNF-based model. Thus, the granularity level (the number of distinct values the function could take for all possible input combinations) of the score function is significantly smaller in the constrained circuit. This essentially affects the accessibility of the search space to the search algorithm as plateaus are more likely in the constrained circuit model than in the CNF-based model. To deal with that, we use restarts (Ryvchin and Strichman 2008) from scratch in Algorithm 9. If the search reaches the local minima $10,000$ times, the algorithm resorts to the restart procedure. Since our focus is more to study the constraint models, in this work, we do not try any other restart strategies or other criteria to trigger the restarts.

Defining Impact Variables. In CNF-based models, flipping a variable v from a falsified clause c makes c satisfiable. In gate-based constrained circuit models, because of the layered DAG, flipping an independent variable $v \in I$ might not change the value of $v' \in V_C$ even if $v \in I(v')$. We, therefore, have to find a mechanism to maintain a set of independent variables that can consequently change the value of v', given the current assignment $\overrightarrow{I}$. We will call such independent variables the *impact variables* for v' w.r.t. the current assignment $\overrightarrow{I}$ and will denote the set of those variables by $\mathcal{I}(v', \overrightarrow{I})$. A variable selection heuristic will only consider the impact variables. Fig. 5 shows the set of impact variables for the dependent variables in the DAG, given the current assignment $v_0 =$ true, $v_1 =$ false, $v_2 =$ true, $v_3 = false$, $v_4 =$ false. We see $e =$ false, which could be changed to true if either g_0 or g_0' changes. However, g_0 will change if v_1 changes and g_0' will change if v_2 does. So the set of impact variables for g_0, g_0', and e are respectively the sets $\{v_1\}$, $\{v_2\}$, and $\{v_1, v_2\}$. Notice that given the current assignment, changing v_3 will not change e although $e \lll v_3$.

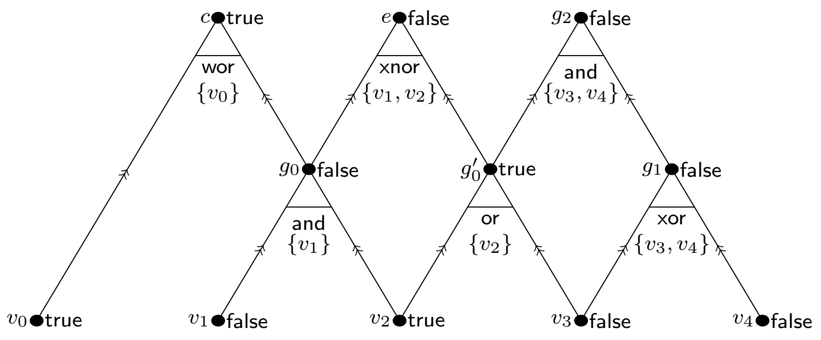


Fig. 5 List of variables that could change gate outputs, given a current assignment. For each gate, the list of such variables is shown as a set under the gate type name.

Computing Impact Variables. We provide the recursive definitions to compute the sets of impact variables for the output variables of various types of gates. These definitions are used in Kangaroo to compute the impact variables incrementally as variables are flipped during local search. Note that we use bitmaps to represent sets of inputs that are in the impact variables. In the definitions below, v is the output of a gate while v' is an input.

1. **Independent Variables**: For recursion basis, $\mathcal{I}(v, \overrightarrow{I}) = \{v\}$ for any $v \in I$ and any $\overrightarrow{I}$.
2. **Outputs of and gates**: When the output is true, any change in any input will change the output. When the output is false, it will be true if all inputs that are false change but no input that is true does. So we have $\mathcal{I}(v, \overrightarrow{I}) = \bigcup_{v'=\text{true}} \mathcal{I}(v', \overrightarrow{I})$ if $v =$ true and $\mathcal{I}(v, \overrightarrow{I}) = \bigcap_{v'=\text{false}} \mathcal{I}(v', \overrightarrow{I}) \setminus \bigcup_{v'=\text{true}} \mathcal{I}(v', \overrightarrow{I})$ if $v =$ false.
3. **Outputs of or gates:** When the output is false, any change in any input will change the output. When the output is true, it will be false if all inputs

that are true change but no input that is false does. So we have $\mathcal{I}(v, \overrightarrow{I}) = \bigcup_{v'=\mathsf{false}} \mathcal{I}(v', \overrightarrow{I})$ if $v = \mathsf{false}$ and $\mathcal{I}(v, \overrightarrow{I}) = \bigcap_{v'=\mathsf{true}} \mathcal{I}(v', \overrightarrow{I}) \backslash \bigcup_{v'=\mathsf{false}} \mathcal{I}(v', \overrightarrow{I})$ if $v = \mathsf{true}$.

4. **Outputs of atype gates**: The impact variables for nand, nor, cg, and dg gates can be easily computed from that for and and or gates considering negation of the inputs and outputs.
5. **Outputs of etype gates**: The output changes as the input changes. So $\mathcal{I}(v, \overrightarrow{I}) = \mathcal{I}(v', \overrightarrow{I})$
6. **Outputs of xtype gates**: The output of a binary gate will change when one input changes but not both. So we have $\mathcal{I}(v, \overrightarrow{I}) = \bigcup \mathcal{I}(v', \overrightarrow{I}) \setminus \bigcap \mathcal{I}(v', \overrightarrow{I})$. For an n-ary gate with $n > 2$, each independent variable that appears in the set of impact variables of an odd number of inputs can change the output. So $\mathcal{I}(v, \overrightarrow{I}) = \{v'' : \mathsf{odd}(|\{v' : v'' \in \mathcal{I}(v', \overrightarrow{I})\}|)\}$.

4.2 Implementation on Kangaroo

As discussed before, an invariant engine is required to discharge the obligation of maintaining the consistency between the input variables and the output variable of each logic gate. For this, we use Kangaroo (Newton et al. 2011) a generic CBLS system that supports invariants implementing algebraic and combinatorial expressions. It has an invariant engine that incrementally and so efficiently propagates changes form independent variables to the dependent variables; however, it could be slower than an invariant engine developed for a specific problem. Kangaroo allows to take a declarative approach to problem modelling and quick implementation of CBLS algorithms using off-the-shelf methods from its library. Also, new invariants and methods can be added to Kangaroo for future users.

We implement our algorithms on top of Kangaroo. Given the gate graph obtained after Algorithm 1, we calculate the gate level of each variable in the gate graph. Then, the independent variables are created in Kangaroo, which is a `C++` library. Next, invariants are created for the dependent variables and their defining functions in the increasing order of their gate levels. The invariants not only do compute the functional values but also compute the impact variables for each gate. The score for each potential flip move is calculated incrementally by invariants as well. All our algorithms are implemented in `C++` on top of the invariants and variables created in Kangaroo.

5 Experimental Results

We run all experiments on the High Performance Computing Cluster Gowonda at Griffith University, Australia. Each cluster node is equipped with Intel Xeon CPU E5-2650 processors @2.60 GHz. We evaluate our gate-based constraint circuit models and algorithms using the following two sets of experiments.

Detailed Experiments: We run a number of versions of our solvers on satlib (`http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`) and sat14 (SAT Competition 2014 `http://www.satcompetition.org/`) problem instances. We use 4GB memory for these experiments. We perform a wide range of analyses of the results. These are presented in Sections 5.2–5.10.

Additional Experiments: We run a number of selected versions of our solvers on sat20 (SAT Competition 2020 `http://www.satcompetition.org/`) problem instances. To obtain an understanding about where our solvers stand with respect to the current state of the art, we also compare our algorithms with kissat (Biere et al. 2020), which is a CDCL based solver and the winner of the SAT Competition 2020. We use 32GB memory for these experiments and the results are presented in Section 5.11.

5.1 Performing Statistical Tests

To check whether there exist significance differences in the performances of various solver versions, wherever possible, with 95% level of significance, we perform pairwise t test on the solution times. For multi-comparison among more than two solver versions, instead of performing t tests, we perform one-way analysis of variance (ANOVA) plus Tukey's honestly significant difference (HSD) test on the solution times with 95% level of significance. Later, in our analysis wherever appropriate, we just mention the statistical test names assuming the level significance to be 95%. Nevertheless, to check whether performances of two solvers correlate with each other, we also compute Pearson's correlation coefficient on the solution times and use that in our analysis. Since not all solver versions could solve all instances, with the missing data, sometimes these statistical tests might not lead to meaningful conclusions in which case we consider the number of problems solved or some other criteria, mentioned where appropriate, to make a selection.

5.2 Detailed Experiment Benchmarks

From satlib, we use 4 ssa (single-stuck-at fault), 10 qg (quasigroup), 5 p32 (parity 32), 5 p16 (parity 16), 4 log (logistics), 7 bw (blockworld), and 13 bmc (bounded model checking) satisfiable instances. From sat14, we use 9 app (application) and 12 hc (hard combinatorial) instances. We actually tried other app and hc instances but none of the solver versions that we have used could solve them within the resource (time and memory) limits; so we do not show them in our results. As shown in Table 1, for convenience, we rename the instances by using single letter names. In total 69 instances used in our detailed experiments.

5.3 Before Gate Detection

As mentioned earlier, before detecting gates, with our own implementation, we clean up the CNF formula using mostly SAT based simplification techniques (mainly removing repeated clauses, unit clause propagation). Tables 2 and 3 show the numbers of variables that have been fixed and the numbers of clauses that have been deleted respectively against the numbers of variables and the numbers of clauses in the problem instances. No cleaning up was possible in the log and bw problem instances. In the hc, app, and ssa problem instances, the percentages of variables fixed and the percentages of clauses deleted are small while in the p16, p32, bmc, and qg problem instances, the percentages are noticeable.

Table 1 Satisfiable benchmark problem instances from satlib and sat14

p16	satlib
A	par16-1.cnf
B	par16-2.cnf
C	par16-3.cnf
D	par16-4.cnf
E	par16-5.cnf

p32	satlib
A	par32-1.cnf
B	par32-2.cnf
C	par32-3.cnf
D	par32-4.cnf
E	par32-5.cnf

bmc	satlib
A	bmc-ibm-1.cnf
B	bmc-ibm-2.cnf
C	bmc-ibm-3.cnf
D	bmc-ibm-4.cnf
E	bmc-ibm-5.cnf
F	bmc-ibm-6.cnf
G	bmc-ibm-7.cnf
H	bmc-galileo-8.cnf
I	bmc-galileo-9.cnf
J	bmc-ibm-10.cnf
K	bmc-ibm-11.cnf
L	bmc-ibm-12.cnf
M	bmc-ibm-13.cnf

ssa	satlib
A	ssa7552-038.cnf
B	ssa7552-158.cnf
C	ssa7552-159.cnf
D	ssa7552-160.cnf

qg	satlib
A	qg1-07.cnf
B	qg1-08.cnf
C	qg2-07.cnf
D	qg2-08.cnf
E	qg3-08.cnf
F	qg4-09.cnf
G	qg5-11.cnf
H	qg6-09.cnf
I	qg7-09.cnf
J	qg7-13.cnf

app	sat14
A	atco_enc1_opt1_03_56.cnf
B	atco_enc1_opt1_04_32.cnf
C	atco_enc1_opt1_05_21.cnf
D	atco_enc1_opt1_10_15.cnf
E	atco_enc1_opt1_10_21.cnf
F	atco_enc1_opt1_15_240.cnf
G	atco_enc2_opt1_05_21.cnf
H	atco_enc2_opt1_10_21.cnf
I	atco_enc2_opt2_05_9.cnf

bw	satlib
A	bw_large.a.cnf
B	bw_large.b.cnf
C	bw_large.c.cnf
D	bw_large.d.cnf
E	huge.cnf
F	medium.cnf
G	anomaly.cnf

log	satlib
A	logistics.a.cnf
B	logistics.b.cnf
C	logistics.c.cnf
D	logistics.d.cnf

hc	sat14
A	Chvatal-k5.cnf
B	rnd-v25-e13-k3.cnf
C	Composite-024BitPrimes-1.used-as.sat04-861.cnf
D	instance_n8_i8_pp_ci_ce.cnf
E	instance_n8_i9_pp.cnf
F	jkkk-one-one-10-34-sat.cnf
G	prime2209-98.cnf
H	SAT_instance_N=31.cnf
I	SAT_instance_N=33.cnf
J	toughsat_factoring_155s.cnf
K	toughsat_factoring_958s.cnf
L	toughsat_factoring_inf.cnf

Table 2 Average effect of cleaning up before gate detection over satlib and sc14 problem instance types

Problem Instances	p16	p32	bmc	ssa	qg	log	bw	app	hc
Variables Fixed (%)	38.8	24.8	36.3	7.3	44.8	0	0	1.3	5
Clauses Deleted (%)	42.8	27.8	45.2	9.5	72.8	0	0	4.8	6

5.4 Detected Gate Statistics

Table 4 shows the numbers of gates detected from the satlib and sat14 benchmark problem instances. The cg and dg gates are when both positive and negative literals are present as inputs, not just variables as in the and, or, nand, and nor gates. The bmc problem instances have all types of gates, although 6 problem instances have no xor gate. The ssa problem instances have all but and, dg, and xor gates. The qg problem instances all have large numbers of nor gates and small numbers of not gates, with some of the problem instances having significant numbers of eq gates, while 2 problem instances having significant numbers of and gates. The log problem instances have mostly nor and not gates except one problem instance having or, cg and eq gates as well. The p16 and p32 problem instances have and, cg, xor, eq, and not gates. The bw problem instances have mostly and, nor, cg, and eq gates with 3-4 not gates as well. The app problem instances mostly have and, or, nand, cg, eq and not gates except one problem instance that has nor gates as well. The hc problem instances have mostly and, xor, and xnor gates with different problem instances having different types of gates as well, but no problem instance has nand gates. Overall the problem instances include various types of gate distributions.

Table 3 Effect of cleaning up before gate detection on satlib and sc14 problem instances

id	vars	fixed	%	clauses	deleted	%
			p16			
A	1015	408	40	3310	1466	44
B	1015	383	38	3374	1416	42
C	1015	395	39	3344	1440	43
D	1015	396	39	3324	1442	43
E	1015	388	38	3358	1426	42
			ssa			
A	1501	40	3	3575	220	6
B	1363	186	14	3034	511	17
C	1363	132	10	3032	376	12
D	1391	25	2	3126	97	3
			qg			
A	343	175	51	68083	61324	90
B	512	231	45	148957	128373	86
C	343	164	48	68083	60518	89
D	512	216	42	148957	126431	85
E	512	239	47	10469	7093	68
F	729	294	40	15580	9327	60
G	1331	507	38	64054	36639	57
H	729	340	47	21844	14507	66
I	729	395	54	22060	16332	74
J	2197	785	36	97072	51710	53
			app			
A	43998	552	1	276198	14195	5
B	57220	720	1	582308	31762	5
C	59517	324	1	561784	24818	4
D	46759	552	1	261059	11232	4
E	46993	552	1	270831	11232	4
F	61642	1004	2	644099	32318	5
G	56533	324	1	526872	24818	5
H	44854	535	1	239768	11474	5
I	14912	472	3	390488	24966	6

id	vars	fixed	%	clauses	deleted	%
			p32			
A	3176	758	24	10277	2817	27
B	3176	784	25	10253	2869	28
C	3176	781	25	10297	2863	28
D	3176	791	25	10313	2883	28
E	3176	791	25	10325	2883	28
			bmc			
A	9685	2600	27	55870	20571	37
B	2810	1666	59	11683	8413	72
C	14930	2990	20	72106	16235	23
D	28161	14793	53	139716	83332	60
E	9396	4533	48	41207	24641	60
F	51639	26426	51	368352	203504	55
G	8710	4844	56	39774	25493	64
H	58074	14111	24	294821	112560	38
I	63624	14223	22	326999	118689	36
J	59056	26679	45	323700	169675	52
K	32109	9397	29	150027	58909	39
L	39598	6881	17	194778	45977	24
M	13215	2784	21	65728	18142	28
			hc			
A	4104	364	9	31868	4810	15
B	26575	1544	6	442285	34780	8
C	1199	96	8	11158	1124	10
D	21640	0	0	257551	6272	2
E	24336	0	0	287938	7056	2
F	11607	995	9	73804	8056	10
G	118482	6174	5	349076	17934	5
H	7808	608	8	25725	1437	6
I	8912	644	7	29422	1520	5
J	1824	55	3	9776	328	3
K	1824	55	3	9776	328	3
L	2878	67	2	15516	426	3

5.5 Solvers, Options, Settings, and Runs

Henceforth, we use the following terminologies for solvers, options, and settings:

Solvers: A solver is a broad class of runnable complete systems that are used to solve satisfiability problems given as CNF formulas. We use two solvers anp and cca, where they respectively use the AdaptNovelty+ and the CCAnr variable selection heuristics. A solver can have many characteristics, such as a restart method, a tabu technique, and detection of atype gates. Every solver characteristic is by default deactivated and needs to be explicitly activated.

Options: An option is a single parameter that can be used to activate a certain characteristic of a given solver. For example, we respectively use t and r as two options to enable using the tabu technique and the restart method within a given solver. We use a concatenation of multiple options (e.g. tr or rt) ignoring their order to mean they are used simultaneously.

Settings: Given a number of options available for a solver, each unique combination or subset of the available options is called a setting of the solver. Certain combinations of options may not be supported by the solver and hence such a setting is invalid. Given two options {r, t} of a solver anp, we have {rt, r, t, -} as four settings of anp, where - denotes an empty subset. We may respectively use anp-rt, anp-r, anp-t, and anp– to represent these settings. To discuss a number of settings conveniently, we might mention the **shared subsetting**

Table 4 Numbers of gates detected from the satlib and sat14 benchmark problem instances

id	and	or	nand	nor	cg	dg	xor	xnor	eq	not
p32										
A	125	0	0	0	61	0	1158	0	1073	30
B	125	0	0	0	61	0	1146	0	1052	37
C	125	0	0	0	61	0	1168	0	1036	34
D	125	0	0	0	61	0	1176	0	1010	42
E	125	0	0	0	61	0	1182	0	1010	36
p16										
A	31	0	0	0	30	0	270	0	273	17
B	31	0	0	0	30	0	302	0	265	18
C	31	0	0	0	30	0	287	0	272	14
D	31	0	0	0	30	0	277	0	278	17
E	31	0	0	0	30	0	294	0	268	18
log										
A	0	0	0	89	0	0	0	0	0	46
B	0	0	0	76	0	0	0	0	0	62
C	0	0	0	98	0	0	0	0	0	64
D	0	256	0	201	122	0	0	0	211	22
bw										
A	72	0	0	18	32	0	0	0	12	3
B	110	0	0	27	70	0	0	0	16	3
C	210	0	0	42	161	0	0	0	24	3
D	342	0	0	54	280	0	0	0	31	4
E	72	0	0	18	32	0	0	0	12	3
F	20	0	0	11	9	0	0	0	5	3
G	6	0	0	7	3	0	0	0	3	4
bmc										
A	1057	888	224	122	1027	1768	0	0	722	37
B	192	167	0	35	57	155	0	0	348	84
C	3227	2637	198	67	1483	1692	72	984	1303	196
D	1788	1891	221	72	2725	2254	61	356	3088	227
E	944	627	18	15	416	441	0	26	1298	202
F	3046	2650	127	240	5777	4285	164	650	5915	1030
G	801	757	13	161	255	511	0	24	911	124
H	18754	8566	507	61	1022	1922	0	670	10417	966
I	21061	9598	570	67	1212	2211	0	766	11560	1071
J	4803	6451	231	461	8887	4489	112	29	5800	705
K	4533	4206	489	304	2721	2897	1202	1807	3735	266
L	6194	5862	688	505	5348	5127	1370	2319	4452	265
M	2253	2173	212	27	1510	928	300	950	1540	352
ssa										
A	0	23	41	42	40	0	0	15	921	95
B	0	7	23	23	34	0	0	3	804	87
C	0	11	25	26	34	0	0	8	838	86
D	0	19	41	35	38	0	0	15	909	92
qg										
A	0	0	0	110	0	0	0	0	0	4
B	0	0	0	152	0	0	0	0	0	4
C	0	0	0	116	0	0	0	0	0	4
D	0	0	0	158	0	0	0	0	0	4
E	20	0	0	152	0	0	0	0	0	4
F	24	0	0	200	0	0	0	0	0	4
G	0	0	0	306	0	0	0	0	76	6
H	0	0	0	193	0	0	0	0	18	5
I	0	0	0	180	0	0	0	0	28	6
J	0	0	0	433	0	0	0	0	48	5
hc										
A	649	0	0	100	0	0	0	0	77	10
B	7040	0	0	535	0	0	0	0	160	25
C	0	0	0	0	0	9	0	0	14	9
D	64	448	0	0	448	0	0	448	3652	28
E	72	504	0	0	504	0	0	504	3160	28
F	2105	954	0	1	2775	5	80	4550	0	0
G	29008	1	0	55762	23029	0	0	0	2254	588
H	3591	858	0	0	0	47	2642	0	51	49
I	4155	980	0	0	0	37	3030	0	87	37
J	596	0	0	0	0	6	75	24	52	6
K	597	0	0	0	0	5	77	22	53	5
L	959	0	0	0	0	2	148	24	61	2
app										
A	108	382	720	0	0	0	0	0	6779	3108
B	108	396	720	0	36	0	0	0	8642	3300
C	120	430	720	0	36	0	0	0	9176	3468
D	126	416	744	0	26	0	0	0	7509	3398
E	126	416	744	0	26	0	0	0	7509	3398
F	186	464	744	0	0	0	0	0	8891	3398
G	120	430	0	0	36	0	0	0	6192	3468
H	126	416	0	0	26	0	0	0	5386	3398
I	120	146	36	14	36	0	0	0	288	344

separately and then the **disjoint subsetting** of each setting separately. For example, to discuss a set of settings {rt, r}, we separate the shared subsetting r from the disjoint subsettings t and -. In some charts, to show the performance of a particular subsetting r regardless of its accompanying subsettings t or -, we put t and - in the x-axis and r in the legend, and plot the performance of rt and r on the chart as points and draw a line between the two points.

We perform two types of running of the solver settings on the satlib and sat14 benchmark problem instances.

Pilot Runs: A solver setting is run 5 times on a problem instance, each with 1-hour timeout.

Final Runs: A solver setting is run 25 times on a problem instance, each with 25-hour timeout.

The pilot runs are to perform various analyses and the final runs to show final results. We consider a problem instance is solved by a solver setting if (#successful-run/#total-run) i.e. the success rate is at least 50%, and we take the median solution times of the successful runs as the representative solution time and use it in further analyses. The reason to consider at least 50% success rate is to perform a robust analysis. Because of stochasticity, local search algorithms all on a sudden can perform too good or too bad in a single run. So with low success rates (e.g. solved just in a few runs), the likelihood of having outliers is high and the statistical central measures might not be very reliable. Yet for the sake of analysing the best performance of the solver settings over the number of runs attempted, later we show a chart (e.g Fig. 11) only for the best performing settings.

5.6 Pilot Runs: Option Effects

The following aspects can be studied for the anp and cca solvers that we have developed for the constrained circuit models for SAT. These relate to the gate types and search algorithms.

1. Use various combinations of the 10 types of gates to see their interaction during search.
2. Simplify the equivalent variables from the constrained circuit model, if eq and not gates are detected.
3. Use the tabu technique to address the revisitation problem of the local search algorithm.
4. Use the restart method to address the local minima and plateaus encountered during search.

Besides the above aspects, we can also remove deferrable variables from each constrained circuit model. However, instead of considering this as a solver option, we perform this as a mandatory step, since this does not affect the search behaviour. So we could have in total 13 options, each of which can be on or off. However, trying $2^{13} = 8192$ settings for each solver is impractical. Therefore, we consider grouping them and thus get 5 options with $2^5 = 32$ settings for each solver.

1. c to use the ctype gates i.e. and, nor, and cg gates
2. d to use the dtype gates i.e. or, nand, and dg gates
3. p to use the ptype gates i.e. xor, xnor, eq, and not, plus equivalence simplication
4. t to use the tabu technique during search with tabu tenure 5
5. r to use the restart method with maximum plateau length $10,000$

In the analysis of these runs, we mainly consider the numbers of problem instances solved by each solver setting although we have tacitly considered t or ANOVA plus HSD test results on the solution times. Fig. 6 in the Top-Left chart shows that regardless of combinations of r, t, c, and d, both solvers solve almost twice the numbers of problem instances with subsetting p (i.e. anp-p and cca-p) than with subsetting - (i.e. anp– and cca–). Henceforth, we decide to use ptype gates in further analysis. The Top-Right chart of Fig. 6 shows that both solvers solve more or equal numbers of problem instances when using the tabu technique (i.e. anp-tp and cca-tp) than when not using (i.e. anp-p and cca-p), regardless of the combinations of r, c, and d. Henceforth, we decide to use the tabu technique along with ptype gates in further analysis. The Bottom-Left chart of Fig. 6 shows both solvers solve almost the same numbers of problem instances when using the restart method (i.e. anp-rtp and cca-rtp) and when not using (i.e. anp-tp and cca-tp), regardless of the combinations of c and d. Henceforth, we decide to use the restart method, because it helps solve slightly more problems with the best settings tp and tpd of anp and cca solvers respectively. Fig. 6 Bottom-Right chart shows the numbers of problem instances solved by the two solvers when various combinations of c and d are used along with rtp. Both solvers perform worse with cd than with c or d. Even not using c and d both can achieve almost the best performance.

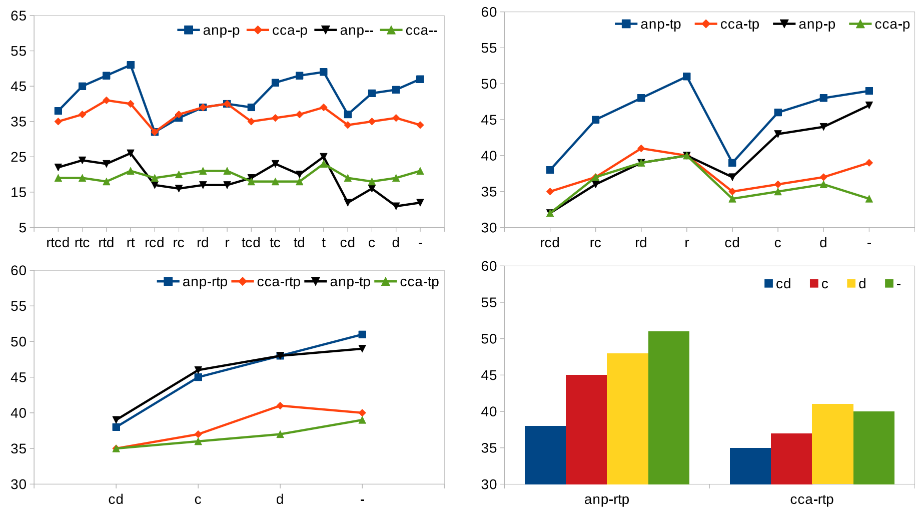


Fig. 6 Various solver settings (x-axis) vs numbers of satlib and sc14 problem instances solved (y-axis) when various other settings (legend) are also used along with. Top-Left: when ptype gates are used and not used (respectively presence and absence of p in the legend). Top-Right: when with ptype gates (presence of p in the legend), tabu technique is used and not used (respectively presence and absence of t in the legend). Bottom-Left: when with ptype gates and tabu technique (presence of tp in the legend), restart method is used and not used (respectively presence and absence of r in the legend). Bottom-Right: when with ptype gates, tabu technique, and restart method (rtp in the x-axis), atype gates are used and not used (all possible subsettings of {c, d} in the legend).

To study the mutual interaction between options c and d, we compute Pearson correlation coefficients among settings cd, c, d, and - of both anp and cca solvers, where all settings have rtp in common. These coefficients are in Table 5. Because of the common rtp, some positive correlations are expected among the four settings and we have to consider the differential effects. As we observe, comparatively high correlations between cd and c or d show that both options c and d need not be used together. Also, comparatively low or moderate correlations between - and c or d show the importance of using options c or d separately.

5.7 Pilot Runs: Generalised vs Specialised Gates

The cg and dg gates as defined in (9)–(10) can have both positive or negative literals as inputs. As such, these gates generalise over the specialised gates and, nor,

Table 5 Pearson's correlation coefficients among settings cd, c, d, - of both anp and cca solvers, where all settings have rtp in common. Only coefficients larger than 0.66 are shown.

anp-rtp				cca-rtp			
setting	cd	c	d	setting	cd	c	d
c	0.92			c	0.98		
d	0.67			d	0.94	0.79	
-		0.68	0.89	-			0.69

or, and nand that can have only variables as inputs. We however detect cg and dg gates only when both positive and negative literals are inputs. When only variables are involved, the gate is detected as a specialised gate rather than a generalised gate. The specialised gets are more efficient than the generalised gates in that the specialised gates do not check the signs of the inputs while the generalised gates do. We investigate whether this difference in efficiency is significant to ponder further. For this, we design the two following paired runs for each solver setting for each problem instance.

1. g to detect cg gates instead of and and nor gates, and dg gates instead of or and nand gates
2. s to detect and, nor, or, and nand gates as they are usually, and not as the generalised gates

In terms of Table 4, for s runs, the numbers of and, nor, nand, or, cg, and dg gates are the same as shown in the table. In g runs, the numbers of cg gates will be the sum of the columns and, nor, and cg of the table. Similarly, the numbers of dg gates in g runs will be the sum of the columns or, nand, and dg of the table. We change the gate detection algorithm accordingly to detect the gates as per the design of g runs. We use the same random number seed for each pair of corresponding g and s runs, and we run 5 such paired runs with 5 distinct random number seeds. We consider using rtp setting along with as we decided in Section 5.6, but we vary settings on combinations of c and d. Figure 7 shows that there is no significant difference between g and s runs. The reason could be that in each iteration of the search, only a few inputs in each gate actually change because of each single variable flip move. With incremental propagation of changes from the independent to the dependent variables, the efficiency in not checking the signs of the inputs in the specialised gates are not much (statistically not significant as per t test) compared to the sign checking in the generalised gates. Henceforth, we will only consider g runs. So all atype gates will be either cg or dg gates.

5.8 Pilot Runs: Using ptype Gates

Fig. 6 has already showed that the ptype gates make big differences in the performances of both the solvers. To further analyse, we divide the ptype gates into xtype and etype gates as shown in Definition 2. For the etype gates, we have a choice to simplify equivalent variables as per Step 8 of Algorithm 1 and Algorithm 7 or not to simplify. Below we list the options and in the experiments, we consider each possible combination of these options.

1. x to detect the xtype gates

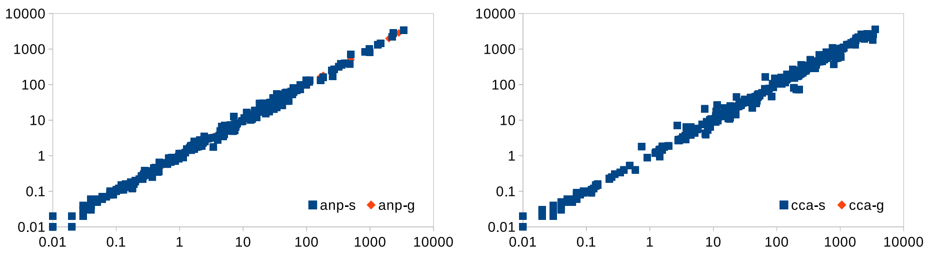


Fig. 7 Paired comparison of time performance of `g` runs (x-axis) vs `s` runs (y-axis) for `anp` (left) and `cca` (right) solvers with the fixed setting of `rtp`, but varied on `c` and `d` combinations. Differences are not significant.

2. `n` to detect the `etype` gates, but not to perform equivalence simplification
3. `e` to detect the `etype` gates and then perform equivalence simplification

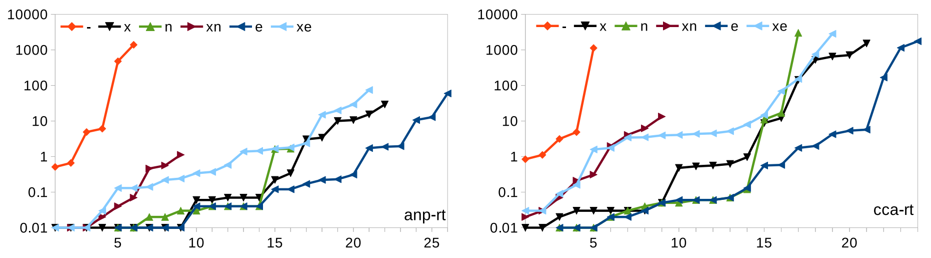


Fig. 8 Numbers of problem instances solved (x-axis) vs time in seconds (y-axis) for `anp` (left) and `cca` (right) solvers, when various settings are used for `ptype` gates but along with the same setting of `rt`.

We run these solver settings along with the same `rt` setting. Like Fig. 6 Top-Left, Fig. 8 also shows `ptype` gates lead to significantly better performances than using no `ptype` gates (as per ANOVA plus HSD tests). Nevertheless, we notice from Fig. 8 that setting `e` of both `anp` and `cca` solvers obtains the best performance, where the equivalence simplification makes significant difference from setting `n`. Setting `x` performs the second followed by setting `xe` that combines `x` and `e`. In terms of Fig. 8, Fig. 6 actually shows performance of the `xe` setting. Interestingly, setting `xn` performs the worst and its performance difference from `xe` setting shows the usefulness of equivalence simplification again. To analyse the correlation between solver options statistically, we compute Pearson correlation coefficients among the top three settings `e`, `x`, and `xe`. The coefficient between `e` and `xe` is above 0.9 while that between `x` and `xe` and also between `x` and `e` are below 0.2. These coefficients confirm option `e` is dominant over options `x` while both options are useful.

5.9 Final Runs: Final Experiments

From Fig. 8, we see the three settings $\{\mathsf{x}, \mathsf{e}, \mathsf{xe}\}$ are the best performers when used along with the same setting `rt`. From Fig. 6 bottom-right, we see the three settings

$\{$c, d, -$\}$ are the best performers along with the same setting rtp. So we run our final experiments using the same setting rt i.e. with the restart method and the tabu technique, but varying on the combinations of gate types $\{$c, d, -$\} \times \{$x, e, xe$\}$. We further include a setting rt where no gate is detected, but just the restart method and the tabu technique are used. We also include an additional setting o to run the original AdaptNovelty+ and CCAnr solvers implemented by the respective authors. We include another setting denoted by l, where equivalence literals are eliminated from the CNF formula using equivalence literal elimination technique Manthey (2012) and then the original AdaptNovelty+ and CCAnr solvers are run on the obtained CNF formula. Setting l is included because option e is just equivalence simplification statically performed on gate graphs before search and we want to see how these two differ in performance. The solution times reported for setting l include the time needed for equivalence literal elimination.

So we have the following 12 settings for each of the two solvers anp and cca.

$$(\{rt\}\times\{\text{c, d, -}\}\times\{\text{x, e, xe}\})\cup\{\text{rt, o, l}\} = \{\text{rtcx, rtce, rtcxe, rtdx, rtde, rtdxe, rtx, rte, rtxe, rt, o, l}\}$$

We use gtype setting to mean any of the settings $\{$rtcx,rtce,rtcxe,rtdx,rtde,rtdxe,rtx,rte,rtxe$\}$ that have c, d, x or e in it. Setting rt is considered as a baseline setting, and o and l as original settings.

Table 6 Average of various statistical data over problem instance of the same types

(a) Average numbers of gates detected

	-	e	xe	de	dxe	ce	cxe	x	dx	cx
ssa	0	958	968	1006	1016	1026	1036	10	58	78
qg	0	22	22	22	22	226	226	0	0	204
p32	0	1079	2241	1079	2241	1265	2427	1162	1162	1348
p16	0	288	574	288	574	349	635	286	286	347
log	0	101	101	168	168	247	247	0	66	146
hc	0	867	1836	1197	2165	11867	12835	969	1298	11968
bw	0	18	18	18	18	246	246	0	0	228
bmc	0	4355	5267	10438	11351	12234	13147	912	6996	8792
app	0	9739	9739	10620	10620	9892	9892	0	880	153

(b) Average of maximum $L(v) : v \in \Lambda(C)$

	-	e	xe	de	dxe	ce	cxe	x	dx	cx
ssa	3	5	5	9	9	8	8	4	5	5
qg	3	6	6	6	6	23	23	3	3	22
p32	3	3	103	3	103	66	165	66	66	68
p16	3	3	51	3	51	34	81	34	34	35
log	3	5	5	5	5	5	5	3	4	5
hc	3	4	7	5	8	14	17	6	7	18
bw	3	4	4	4	4	8	8	3	3	6
bmc	3	4	22	4	22	22	40	15	15	23
app	3	4	4	5	5	4	4	3	4	4

(c) Average numbers of independent variables $|I|$

	-	e	xe	de	dxe	ce	cxe	x	dx	cx
ssa	1309	351	341	304	294	283	273	1299	1251	1231
qg	459	438	438	438	438	281	281	459	459	290
p32	2395	1323	157	1323	157	1138	33	1229	1229	1104
p16	621	333	47	333	47	273	17	335	335	304
log	1881	1782	1782	1718	1718	1642	1642	1881	1816	1737
hc	9023	8430	7461	8118	7156	6229	5431	8055	7736	6414
bw	1644	1626	1626	1626	1626	1405	1405	1644	1644	1423
bmc	19919	15561	14649	9604	8808	7820	6979	19009	12945	11136
app	47488	37749	37749	36869	36869	37596	37596	47488	46608	47335

(d) Average size of neighbour$(v) : v \in I$

	-	e	xe	de	dxe	ce	cxe	x	dx	cx
ssa	4	6	6	9	9	8	9	2	2	2
qg	111	112	112	112	112	279	279	111	111	289
p32	4	2	130	2	130	31	32	16	16	4
p16	4	2	46	2	46	14	16	9	9	3
log	22	22	22	22	22	34	34	21	22	34
hc	21	15	22	19	33	82	110	31	43	40
bw	26	26	26	26	26	47	47	26	26	45
bmc	12	10	13	26	33	37	49	13	22	21
app	19	2	2	2	2	2	2	18	20	19

(e) Average numbers of constrained dependent variables $|V_{\mathrm{C}}|$

	-	e	xe	de	dxe	ce	cxe	x	dx	cx
ssa	2891	934	934	772	772	706	706	934	772	706
qg	15291	15248	15248	15248	15248	13914	13914	15247	15247	13902
p32	7430	622	622	622	622	126	187	622	622	186
p16	1904	184	184	184	184	32	62	184	184	61
log	11682	11482	11482	11264	11264	10522	10522	11480	11261	10515
hc	94057	55615	55615	54713	55528	47542	47234	88921	87571	79578
bw	29495	29459	29459	29459	29459	28554	28554	29459	29459	28554
bmc	90611	51569	51570	31022	31139	21565	21636	78248	43055	36743
app	396288	15173	15173	14928	14928	15173	15173	376810	370879	375999

(f) Average of average $|V(v)| : v \in V_{\mathrm{C}}$

	-	e	xe	de	dxe	ce	cxe	x	dx	cx
ssa	2	3	3	3	3	3	3	3	3	3
qg	3	3	3	3	3	37	37	3	3	38
p32	3	2	30	2	30	5	32	27	27	34
p16	3	2	17	2	17	5	16	15	15	18
log	2	2	2	2	2	2	2	2	2	2
hc	3	3	4	3	5	8	7	6	4	8
bw	2	2	2	2	2	4	4	2	2	4
bmc	3	3	4	6	7	17	17	3	5	5
app	4	2	2	2	2	2	2	4	4	4

Table 6 shows various statistical data over problem instances of the same type, e.g. (a) the average numbers of gates detected, (b) the average of maximum gate levels, (c) the average numbers of independent variables, (d) the average size of the neighbourhood of the independent variables, (e) the average numbers of constrained dependent variables, and (f) the average numbers of involving variables of the constrained dependent variables. Detecting more gates could result into an increase in the number of gate levels in the DAG, a decrease in the number of independent variables, a decrease in the number of dependent variables, and an increase in the number of dependency cycles. The more the number of gates and the bigger the number of gate levels, the bigger the cost of the change propagation. The smaller the number of independent variables, the smaller the search space in terms of the search combinatorics, but for a local search the revisitation problem might be more likely. The smaller the number of constrained dependent variables, the less informative the comparative evaluation of the current and the next potential assignments will be. The bigger the number of dependency cycles, the bigger the number of variables added to break the cycles. Overall detecting more gates has both advantages and disadvantages.

Tables 7 and 8 show the detailed results for each solver setting for each problem instance. Fig. 9 shows the numbers of problem instances solved by various solver settings vs time in seconds. Table 10 shows the total numbers of problem instances solved by each solver setting. Table 10 also shows the total numbers of problem instances with the best time performance by each solver setting among other settings of the same solver and among all settings of both solvers. Fig. 10 shows the performances of the best settings of both solvers along with their baseline and original settings. Below we discuss the results presented in Tables 7 and 8.

1. ssa: These problem instances are all solved by each solver setting in fractions of a second. For anp solver, all gtype settings are faster than the orignal settings o and l, which are faster than the baseline setting b. For cca solver, setting o is faster than all gtype settings, which are faster than the baseline setting rt. Setting l with its cost of equivalence literal elimination is slightly slower than setting o. The best performing settings for these problem instances are anp-rtcx, anp-rtdx, cca-o, anp-rtxe and anp-rte.
2. qg: Problem instances B, D, G, and J are comparatively harder for both anp and cca settings, and are not solved by rtce, rtcxe, rtx, rtdx, and rtcx settings of cca. In 8 out of 10 problem instances, cca-o is the fastest and solves instances within fractions of seconds, but anp settings taking up to several seconds perform better than other cca settings that take tens of seconds. Setting l is somewhat better than setting o for anp but not for cca, where o is faster than l.
3. p32: Settings o, l and rt of anp and settings o, l, rt, rtxe, rtdxe, rtcxe, rtcx of cca could not solve any p32 problem instances. Settings rtxe, rtdxe, and rtcxe of anp struggle to solve these problem instances. Settings rte, rtde, and rtce of both solver solve these problem instances in fractions of seconds and are comparatively faster than settings rtx and rtdx.
4. p16: Settings o, l, and rt of anp could solve p16 problem instances each in thousands of seconds. Original settings o and l of cca could solve those problem instances each in tens of seconds while the baseline setting rt cannot solve any of them. Settings rte, rtde, rtce, rtx, rtdx, rtcx of both solvers solve these problem

Table 7 anp success rates (%s) and solution times (sec). Blank %s means 100. Blank sec means %s < 50

		o		l		rt		rte		rtxe		rtde		rtdxe		rtce		rtcxe		rtx		rtdx		rtcx	
		%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec
	A		0.24		0.13		2.39		0.03		0.02		0.19		0.06		0.03		0.03		0.01		0.01		0.01
	B		0.32		0.08		4.94		0.01		0.01		0.01		0.01		0.01		0.01		0.01		0		0
ssa	C		0.07		0.06		1.09		0.01		0.01		0.01		0.01		0.01		0.01		0.01		0.01		0
	D		0.06		0.08		1.51		0.01		0.01		0.07		0.06		0.1		0.08		0.01		0.01		0.01
	A		8.6		7.2		1.52		1.21		0.8		1.04		0.94		5.87		8.67		1.1		2.17		28.04
	B		629		173		1186		828		435		396		628	92	13488	96	17828		1119		830	88	34164
	C		0.86		1.2		0.97		0.63		0.81		0.85		0.82		5.52		5.44		2.2		1.79		26.8
	D		673		981		2890		2042		921		1458		1413	56	34305	60	53317		15740		11925	20	
qg	E		5.27		4.1		0.17		0.27		0.32		0.32		0.28		0.82		0.46		0.28		0.36		1.8
	F		19.6		28.2		0.73		1.24		0.94		1.11		1.72		14.07		12.65		2.43		2.23		32
	G		9966		7498		28.5		13.3		20.1		16.6		18.6		136		168		482		564		12393
	H		67		58.3		1.22		3.06		2.25		1.76		2.22		1.87		0.94		2.11		2.27		7.89
	I		8.63		6.6		0.09		0.1		0.11		0.1		0.13		0.22		0.27		0.12		0.14		0.93
	J	28		55	30813		444		135		117		78		86	56	22811	68	44214		2678		3569	0	
	A	0		0		0			0.04	8			0.04	0			0.04	76	25515		0.06		0.06		0.07
	B	0		0		0			0.04	24			0.04	16			0.03	68	37362		0.07		0.07		0.06
p32	C	0		0		0			0.04	4			0.04	4			0.03	4			0.06		0.06		0.06
	D	0		0		0			0.04	4			0.04	12			0.04	20			0.06		0.06		0.06
	E	0		0		0			0.04	12			0.04	12			0.04	44			0.06		0.06		0.07
	A		488		679		2646		0.01		0.58		0.01		1.12		0.01		0.16		0.01		0.01		0.01
	B		1307		1067		8189		0.01		0.62		0.01		0.4		0.01		0.16		0.01		0.01		0.01
p16	C		234		886		5851		0.01		0.67		0		0.61		0		0.25		0.01		0.01		0
	D		1038		1082	96	4705		0.01		0.9		0		1.01		0.01		0.51		0.01		0.01		0.01
	E		136		1027		3569		0.01		1		0.01		1.02		0		0.37		0.01		0.01		0.01
	A		0.1		0.12		106		122		121		138		165		89		152		0.13		0.16		0.06
	B		0.18		0.09		643		651		284		648		731		498		428		0.27		0.24		0.08
log	C		0.34		0.41		19330		15476	96	15833		11332	92	10418		1369		1611		0.52		0.62		0.21
	D		2.34		4.05		263		187		138		18		14		110		80		0.42		0.39		0.52
	A		137		156	4		12		4		4		12		96	15711	96	8926		10		7		0.59
	B	0		0		0		0		0		0		0		0		4			99		100.4		57.3
	C	0		0		0			0.17		0.17		0.17		0.17		0.17		0.17	0		0		0	
	D	0		0		0			13		12		12		13		13.11		12	0		0		0	
	E	0		0		0			20		16		20		20		18.78		15	0		0		0	
hc	F	0		0		0			1.23	0			2.61	0			1.06	0		0		0		0	
	G	0		0		0		0		0		0		0		0		0		0		0			8.02
	H	0		0		0			0.23		107		0.23	0			0.18		12362	0		0			7.63
	I	0		0		0			0.26		0.32		0.25		0.29	0		0		0		0			261
	J	0		0		0			0.12		0.14		0.13		0.14	4		0		0		0		0	
	K	0		0		0			0.12		0.14		0.13		0.15		0.13	0		0		0		0	
	L	0		0		0			0.23		0.28		0.22		0.25		0.17		0.17	0		0		0	
	A		0.01		0.03		0.22		0.23		0.15		0.23		0.17		0.39		0.47		0.3		0.18		0.23
	B		0.3		0.4		19		29		29		33.74		26.47		301		356		38		26		152
	C		98.7		73	96	23424	72	22992	72	25935	76	30807	68	40335	16		4			18526	84	22328	12	
bw	D	44			23567	0		0		0		0		0		0		0		0		0		0	
	E		0.02		0.07		0.15		0.22		0.2		0.25		0.27		0.68		0.84		0.2		0.19		0.77
	F		0		0.01		0		0		0		0		0		0		0		0		0		0.01
	G		0		0		0		0		0		0		0		0		0		0		0		0
	A	0		0		0			3.29		3.73	0		0			7.74		10.36		0.54		0.3		0.37
	B	80	16468	0			3386		0.03		0.03	0		0			0.03		0.03		0.03		0.02		0.02
	C	0		0		0		0		0		0		0			2506	0		0			31		247
	D	0		0		0			27		26		1.6		2		216		48		1.7		0.6		0.7
	E	0		0		0		0		0		36		0			0.29		0.36		0.27		0.16		0.16
	F	0		0		0			2.11		2.1		1.57		1.6	0		0			3.88		1.84		2.68
bmc	G	0		0			688		0.16		0.16		0.14		0.14		0.14		0.14		0.24		0.16		0.16
	H	0		0		0		0		0		0		0		0		0			20		2.57		1.75
	I	0		0		0		0		0		0		0		0		0			46		2.92		2.17
	J	0		0		0			1.85		1.84		1.62		1.63		1.69		1.57		5.09		1.95		1.86
	K	0		0		0		0		0		0		0		0		0			37		1.33		1.56
	L	0		0		0			1.68		2.46		1.46		1.98		2.11		3.67	0			2.5		3.09
	M	0		0		0		0		0		0		0		0		0		28			1.14		1.79
	A	0		0		0			5.58		5.81		5.45		5.68		5.68		5.9	0		0		0	
	B	0		0		0			33.25		31.31		29.56		31.72		28.2		38.05	0		0		0	
	C	0		0		0			38.64		38.56		39.32		39.85		41.27		34.14	0		0		0	
app	D	0		0		0			4.98		5.02		4.98		4.98		4.98		4.99	0		0		0	
	E	0		0		0			5.31		5.38		5.73		5.37		5.48		5.31	0		0		0	
	F	0		0		0			54.56		55.15		57.14		53.99		54.1		52.67	0		0		0	
	G	0		0		0			25.66		33.44		32.17		24.02		33.7		26.81	0		0		0	
	H	0		0		0			6.43		5.41		5.09		5.04		5.33		5.4	0		0		0	
	I	0		0		0			22.85		28.21		22.65		22.38		22.86		22.2	0		0		0	

Table 8 cca success rates (%s) and solution times (sec). Blank %s means 100. Blank sec means %s < 50

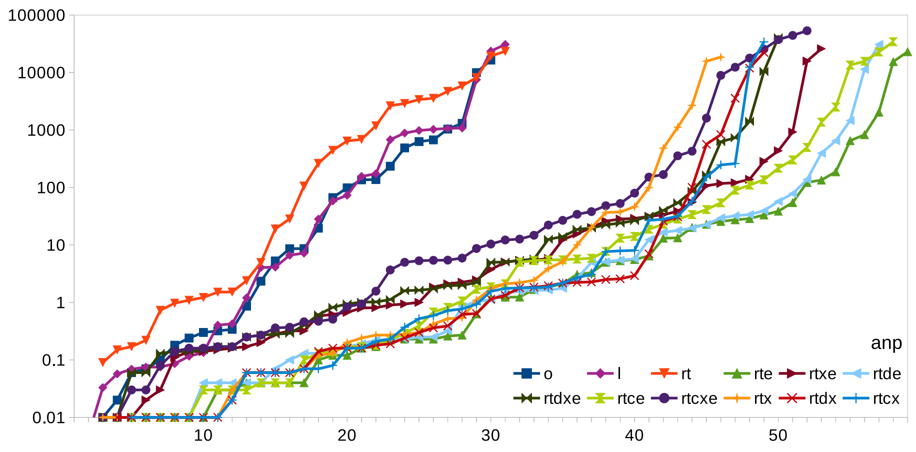
		o		l		rt		rte		rtxe		rtde		rtdxe		rtce		rtcxe		rtx		rtdx		rtcx	
		%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec	%s	sec
	A		0.01		0.1		4.39		0.11		0.17		2.66		0.27		0.67		0.51		0.06		0.05		0.05
	B		0.01		0.06		1.04		0.03		0.03		0.02		0.03		0.01		0.02		0.03		0.02		0.02
ssa	C		0.01		0.06		1.31		0.03		0.03		0.02		0.03		0.01		0.01		0.03		0.02		0.02
	D		0.01		0.07		2.11		0.05		0.09		1.41		0.08		0.05		2.83		0.04		0.03		0.03
	A		0.06		0.29		21.96		14.79		10		16.14		16.01		138		196.74		33.85		36.69		284
	B		1.09		3.2	92	25896	80	26561	76	20623	88	21336	84	42365	8		12		76	16219	92	19299	4	
	C		0.06		0.28		12.09		14.06		15.22		20.68		14.43		50.67		37.57		52.31		40.69		198
	D		12.55		15.6	56	14321	72	38249	60	57246	64	41662	68	50185	4		4		0		0		0	
qg	E		0.02		0.12		8.86		17.56		18.67		19		16.6		28.83		30.64		15.56		21.14		73.6
	F		0.04		0.2		164.1		89.31		104.74		147		148		617		880		199.8		210		4517
	G		0.11		0.9		6618		5577		4327		3867		4556	0		0		36		36		0	
	H		0.03		0.19		26.86		29.17		19.42		22.23		21.17		106		70.91		136.4		141		1559
	I		0.02		0.19		4.69		5.02		4.9		3.64		3.33		9.9		11.22		11.11		13		39.39
	J		8413		8025	16		52	35936	44		36		56	36958	0		0		0		0		0	
	A	0		0		0			0.06	8			0.06	0			0.05	48			0.67		0.71	4	
	B	0		0		0			0.07	4			0.07	4			0.04	44			0.78		0.84	12	
p32	C	0		0		0			0.06	0			0.06	0			0.05	24			0.74		0.69	16	
	D	0		0		0			0.06	4			0.06	0			0.05	12			0.67		0.74	8	
	E	0		0		0			0.06	0			0.06	0			0.05	28			0.73		0.66	8	
	A		15.87		4.9	4			0.01		1.39		0.01		0.88		0.01		0.18		0.02		0.02		0.01
	B		16.17		17.5	0			0.01		1.08		0.01		1.75		0.01		0.29		0.02		0.02		0.01
p16	C		18.28		28.7	0			0.01		2		0.01		1.91		0.01		0.46		0.02		0.02		0.01
	D		17.76		10.1	0			0.01		4.86		0.01		5.95		0.01		0.84		0.02		0.02		0.01
	E		72.02		11.8	0			0.01		2.15		0.01		2.05		0.01		0.45		0.02		0.02		0.01
	A		0.01		0.05		54.85		53.98		50.7		48.46		31.82		68.54		56.19		51.26		52.4		1.48
	B		0.01		0.05		47.73		32.03		37.15		23.24		33.16		58.46		59.1		42.66		41.54		3.21
log	C		0.02		0.09		129.8		126		94.22		82		89.68		203		259		126.01		127.82		6.51
	D		0.07		0.71	52	9262	96	4485	96	3864	96	3830	92	2194	48		44		64	9283	56	143		188.7
	A		4.22		0.64	8		8		8		0		8		0		0			587		831		505
	B		1.67		14.7	0		0		0		0		0		0		0		0		0		8	
	C	0		0		0			0.18		0.18		0.18		0.19		0.18		0.18	0		0		0	
	D	0		0		0		0		4			1300	8		4		0		0		0		0	
	E	0		0		0			1812	84	8343		219		676	36		4		0		0		0	
hc	F	0		0		0			160	0			179	0		60	49057	0		0		0		0	
	G	0		0		0		0		0		0		0			137.6		133	0		0		0	
	H	0		0		0			0.6		46.23		0.48	0			0.25		6414	0		0			14.5
	I	0		0		0			0.63		3.06		0.52		0.88	0		0		0		0			29
	J	0		0		0			1.91		4.84		1.96		4.9		15.35		22.63	0		0			5523
	K	0		0		0			1.95		5.15		1.86		5.26		0.97	0		0		0			6897
	L	0		0		0			5.14		13.48		5.44		14.26		0.19		0.19	0		0		0	
	A		0.01		0.04		18.34		11.72		13.51		14.87		6.62		5.04		10.11		21.4		24.44		8.33
	B		0.03		0.2		2872		1026		2515		2747		3015		3619		3254		5735		5499	96	17633
	C		0.5		1.6	16		16		16		16		12		0		0		8		0		0	
bw	D		5.42		21.3	0		0		0		0		0		0		0		0		0		0	
	E		0		0.07		13.38		12.21		19.34		16.07		19.7		5.77		14.76		21.21		25.04		11.51
	F		0		0.01		0.05		0.05		0.05		0.04		0.04		0.03		0.03		0.06		0.07		0.04
	G		0		0.01		0		0		0		0		0		0		0		0		0		0
	A		20611		431	0		4		0			1412		753	64	3924	64	4447		46.94		70.7		17.27
	B		0.03		1.8		116.4		25.51		27.68		184		188		36.55		51.65		0.15		0.06		0.07
	C	0		0		0		0		0		0		0		0		0			612.8		402		179.8
	D	0		0		0			461		440		1508		625		685		711		139.4		220		42.48
	E		7.52		14.4	0		4		0		0		0			2.27		4.97		11.52		5.4		2.93
	F	0		0		0			5.14		5.15		2.93		2.77	0		0			617.8	24			1611
bmc	G		0.13		1.9		3773		1663		1562		493		128		15.8		26.62		13.41		338		4.6
	H	0		0		0		0		0		0		0		0		0		0		0			1426
	I	0		0		0		0		0		0		0		0		0		0		0			662.7
	J	0		0		0			8.24		9.33		4.28		4.67		3.36		646	0			391		241.7
	K	0		0		0		0		0		0		0		0		0			674		269		227
	L	0		0		0		0		0		0		0		0		0		0			635		508
	M	0		0		0		16		0		0		0		0		0			456		301		1324
	A	0		0		0		16		12		4		8		16		8		0		0		0	
	B	0		0		0		0		0		0		0		0		0		0		0		0	
	C	0		0		0		0		0		0		0		0		0		0		0		0	
app	D	0		0		0		8		4		4		8		8		8		0		0		0	
	E	0		0		0		20		12		8		8		4		4		0		0		0	
	F	0		0		0		0		0		0		0		0		0		0		0		0	
	G	0		0		0		0		0		0		0		0		0		0		0		0	
	H	0		0		0		0		0		0		8		0		0		0		0		0	
	I	0		0		0			48.51		47.58		51.11		47.62		47.64		49.93	0		0		0	

instances in fractions of seconds, while settings rtxe, rtdxe, and rtcxe can take several seconds.

5. log: Settings o, l, rtx, rtdx, and rtcx of anp solve these problem instances within a second, while settings rt, rte, rtxe, rtde, rtdxe, and rtcxe take tens to thousands of seconds. For cca, the original settings o and l solve most of these instances within a second, settings rtx, rtdx, and rtcx solve within about 10 seconds while settings rt, rte, rtxe, rtde, rtdxe, rtce, rtcxe solve in about 100 seconds.
6. hc: Settings o, l, rt, rtx, and rtdx of both anp or cca could not solve $10-11$ out of 12 problem instances while setting rtcx cannot solve 5 instances. Other settings of anp can solve some problem instances within a second while other settings of cca solve some problem instances in several seconds. Problem instances B and G are not solved by most solver settings.
7. bw: Problem instance D is not solved by any settings of anp and cca, except setting l. Problem instance C is challenging for anp gtype settings and could not be solved by any cca gtype settings. Other problem instances are solved within one second by anp and cca settings. Setting o of both solvers are better than the other settings including setting l.
8. bmc: Settings o, l, and rt of solver anp could not solve up to 12 out of 13 bmc problem instances. Setting o, l, and rt of cca could solve 2–4 problem instances. Settings rtcx and rtdx of anp solve all problem instances within several seconds, while the same settings of cca solve most problem instances in several hundreds of seconds.
9. app: Settings o, l, and rt of both anp and cca could not solve any app problem instances. Settings rte, rtxe, rtde, rtdxe, rtce, rtcxe of anp could solve all 9 app problem instances within about 60 seconds. The same settings of cca could solve only 1 problem instance within similar times and have very low success rates in several other problem instances.

Fig. 9 (top chart) shows gtype settings of anp perform much better than o, l, and rt settings. Setting o and l performs very close to each other while setting rt is worse than them. In the short run, settings rtxe, rtdxe and rtcxe show very similar performance, while in the long run rte, rtde, rtce, rtx, rtdx, rtcx show similar performance. The best three anp settings are rte rtde, and rtce, followed by rtxe, rtdxe, rtcx, rtdx, rtcxe, and rtx. Fig. 9 (bottom chart) shows cca-o performs better in the problem instances that take up to 10 seconds time. The difference in the performance of o and rt settings of cca is very much. In the problem instances taking small solution times, setting l is worse than setting o but in the long run their performances are similar. Performances of gtype settings of cca are similar to the same settings of anp. Setting rtcx of cca shows some noticeable behaviour: in terms of the numbers of problem instances solved, rtcx comes the third but over the time duration, it shows significantly worse performance than setting rtce (as per t test). Overall, we consider setting rtce is the third for the cca solver.

Table 9 presents the Pearson's correlation coefficients of 9 gtype settings of both anp and cca solvers. Since all these settings have rt in common, some positive correlations are expected among the four settings and we have to consider the differential effects. We see a cluster of very strong positive correlations (coefficients larger than 0.9 in top left corners of both tables) between all possible pairs of e, xe, de, dxe of both anp and cca. These results show the dominance of option e over



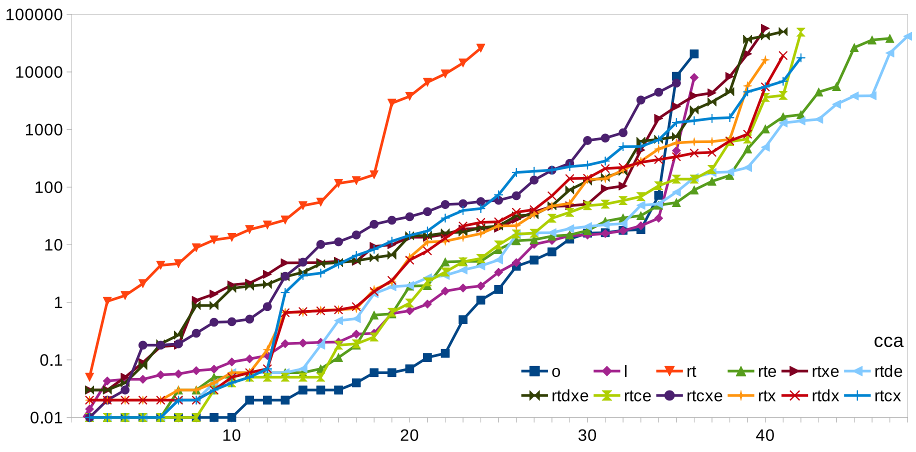


Fig. 9 Numbers of problem instances solved (x-axis) vs time in seconds (y-axis) by anp (top) and cca (bottom) settings

other options x and d. There is another cluster of very strong positive correlations (coefficients larger than 0.9 bottom right corners of both tables) between all possible pairs of x, dx, and cx of both solvers. These results also show the dominance of option x over other options d and c. Moreover, we see last two rows x and dx in both tables have moderate to strong correlations with the other settings of both solvers. However, most these are because of common options x or e or also their moderate to strong correlatios.

There are in total 69 problem instances that we used in the experiments. Table 10 shows the numbers of problem instances solved by each solver setting. Among the anp settings, rte, rtce, rtde are the three best performers. The best four performers (with a tie at the third position) among the cca settings are rtde, rte, rtce, and rtcx. The gtype settings solve significantly more problem instances than the o and rt settings of each solver, except setting rtcxe of cca. While the performance of setting rt is similar to that of settings o and l for anp, there is a significant difference in these settings of cca. Although our emphasis is on solv-

Table 9 Pearson's correlation coefficients among 9 gtype settings of both anp and cca solvers, where all settings have rt in common. Only the coefficients larger than 0.66 are shown.

anp-rt								
setting	e	xe	de	dxe	ce	cxe	cx	dx
xe	**1.00**							
de	**0.97**	**0.98**						
dxe	**0.94**	**0.96**	**0.99**					
ce								
cxe					0.80			
cx								
dx	0.74	0.74	0.83	0.85	0.87	0.75	**0.96**	
x	0.68	0.67	0.72	0.74	0.83	0.70	**0.99**	**0.98**

cca-rt								
setting	e	xe	de	dxe	ce	cxe	cx	dx
xe	**0.96**							
de	**0.99**	**0.98**						
dxe	**0.98**	**0.93**	**0.97**					
ce		0.84		0.82				
cxe				0.74				
cx								
dx	**0.96**	**0.97**	**0.97**	**0.98**	0.68	0.67	**0.96**	
x	**0.94**	**0.96**	**0.96**	**0.96**	0.68	0.67	**0.96**	**1.00**

ing more problem instances, in Table 10, we also show the numbers of problem instances where each setting of each solver performs the best (in terms of solution times) over other settings of the same solver or over all settings of both solvers. Settings rtde, rtce, and rtcx of anp, among all anp settings, perform the best in 17-18 problem instances each. For cca, setting o, among all cca settings, is the best in 27 problem instances, while setting rtce is the second best. Among all settings of both solvers, cca-o is the best in 24 problem instances, and anp-rtce and anp-rtde are the second and third best settings.

Table 10 Summarised performance of various anp and cca settings on 69 problem instances

total numbers of problem instances solved by each setting; top ones emboldened												
	o	l	rt	rte	rtxe	rtde	rtdxe	rtce	rtcxe	rtx	rtdx	rtcx
anp	30	31	31	**59**	53	**57**	50	**58**	52	46	49	49
cca	36	36	24	**47**	40	**48**	41	**42**	35	40	41	**42**

total numbers of problem instances with the best time performance of each setting among all the settings of the same solver; top ones emboldened												
	o	l	rt	rte	rtxe	rtde	rtdxe	rtce	rtcxe	rtx	rtdx	rtcx
anp	6	3	5	15	5	**17**	7	**18**	13	6	15	**18**
cca	**27**	2	1	9	3	10	2	19	5	1	2	13

total numbers of problem instances with the best time performance of each setting among all the settings of both the solvers; top ones emboldened												
	o	l	rt	rte	rtxe	rtde	rtdxe	rtce	rtcxe	rtx	rtdx	rtcx
anp	3	0	2	13	4	16	6	17	11	6	14	14
cca	**24**	0	1	3	1	3	1	3	1	1	1	3

Fig. 10 shows the selected settings having e or x of both anp and cca along with the same setting rt. These settings are selected because the ptype gates are the key behind the performances. Settings rt, l, and o are also included. Both rte and rtx settings of anp clearly outperform the same settings of cca. Setting cca-o and cca-l clearly separate themselves from other o, and l and rt settings. Setting cca-o is the best in problems taking very short solution times. Both baseline settings perform worse than their respective original settings. Setting l is rather worse than setting o. The gtype settings of both solvers perform outstandingly better than the o, l, and rt settings.

Fig. 11 shows similar results as Fig. 10 shows, but not using medians of at least 50% successful runs, rather using the best performance over all runs, successful or unsuccessful. This means Fig. 11 even includes solution times when just one

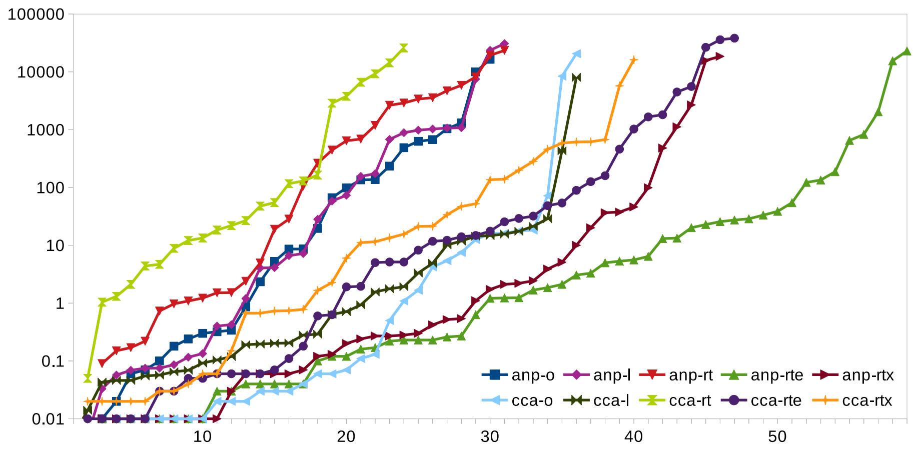


Fig. 10 Numbers of problem instances solved (x-axis) vs time in seconds (y-axis) for selected settings that have `e` or `x`, but not `c` or `d`, and of course along with the same setting `rt`. Setting `o` is also shown for both solvers.

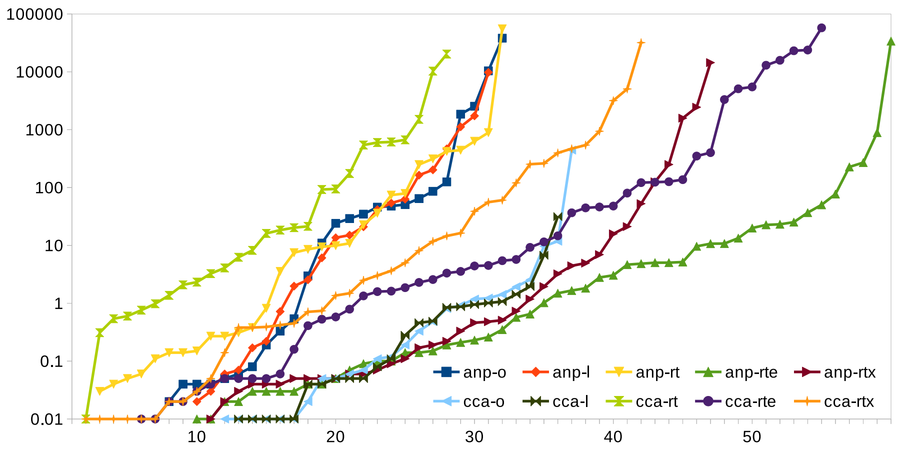


Fig. 11 Numbers of problem instances solved (x-axis) vs time in seconds (y-axis) for selected settings that have `e` or `x`, but not `c` or `d`, and of course along with the same setting `rt`. Setting `o` is also shown for both solvers. These results are based on the best performance in 25 runs.

run was successful. As it appears, the performance trends in Fig. 11 are not very different from what we see from Fig. 10.

5.10 Performance Analysis

We investigate the reasons behind certain very noticeable behaviours of the solver settings.

Baseline Solver Settings

In the final experiments, the baseline settings anp-rt and cca-rt of both solvers appeared to be significantly (as per ANOVA plus HSD tests) worse than the original solvers. The difference is huge in case of the cca solver. The reason is that the specialised implementations of the native SAT solvers have clear advantages in time efficiency over our implementation on top of a generic CBLS system Kangaroo. This is in general the case between a specific system and a generic platform. For CCAnr, we have tried to implement the broader notions of the solver as are described by Cai et al. (2015), but there could be some differences in the implementation details. Considering the scope of this work, we are interested in how the logic gate constraints can be exploited to make significant performance differences in solving more and harder problem instances than when not exploiting those gates. At the end the gtype settings perform better (as per ANOVA plus HSD) than the original solvers, showing their advantages.

Running Original CCAnr

In the final experiments, performances of cca-o and cca-l are significantly better (as per ANOVA plus HSD tests) in 21 problem instances (10 qg, 4 log, and 7 bw) problem instances than that of the gtype settings of cca. We found two reasons behind these. First, the original cca solver can already solve most of these problem instances very quickly. Second, Table 6 shows these problem instances have small numbers of logic gates compared to the other types of problem instances. So the gate detection procedures essentially spend the time but could not find a large number of gates, incurring an overhead to the small search times needed for these problem instances. These are the same reasons behind cca-l being slower than cca-o in total solutions times; the numbers of etype gates are small in these problem instances.

Performance of Setting rtcx

In the final experiments, setting rtcx of both anp and cca obtains very good performance in bmc problem instances. In fact, it is the effect of using xtype gates in these problem instances. The p16, p32 and bmc are the three types of problem instances that have significant numbers of xtype gates, and rtx and rtcx or rtdx settings perform very good in these problem instances.

Numbers of Iterations

In the final experiments, we see the average ratios of the numbers of iterations needed by various solver settings to solve problem instances. Table 11 shows the ratios for the selected solver settings. While computing the ratios, setting rte has been considered as the reference. Clearly, settings rte and rtx of anp take much fewer iterations in most cases than the other three settings. For cca, ratios are lower for o, l, and rt in qg, log, and bw problem instances. We have already discussed that cca is faster in these problem instances that have small numbers of gates.

Table 11 Average ratios of numbers of iterations needed by various solver settings to solve problem instances. Blank entries mean problem instances not solved. Some problem instance types are omitted because the ratios cannot be computed or not meaningful. The number of iterations required by the setting rte is the reference.

anp	o	l	b	rte	rtx
ssa	202.8	25.9	416.2	1.0	0.1
qg	3817.4	2641.4	1.2	1.0	7.2
p32				1.0	2.0
p16	23167464.0	39111319.0	26475128.0	1.0	1.4
log	0.0	0.1	1.3	1.0	0.0
bw	1.0	0.8	1.3	1.0	1.3

cca	o	l	b	rte	rtx
ssa	7.5	3.1	17.6	1.0	0.04
qg	0.4	0.5	1.2	1.0	2.0
p32				1.0	13.8
p16	4342798.0	1968600.1		1.0	4.2
log	0.1	0.1	1.4	1.0	0.0
bw	0.3	0.3	2.0	1.0	2.7

Model Building Times

Fig. 12 shows the model building times required by Algorithm 1 against the total solution times for the problem instances solved by the respective solver settings anp-rte, anp-rtx, cca-rte, and cca-rtx. In most cases, the model building times are very small compared to the total solution times that include the model building times. However, in many cases, particularly for anp-rte, the model building times are almost equal to the solution times.

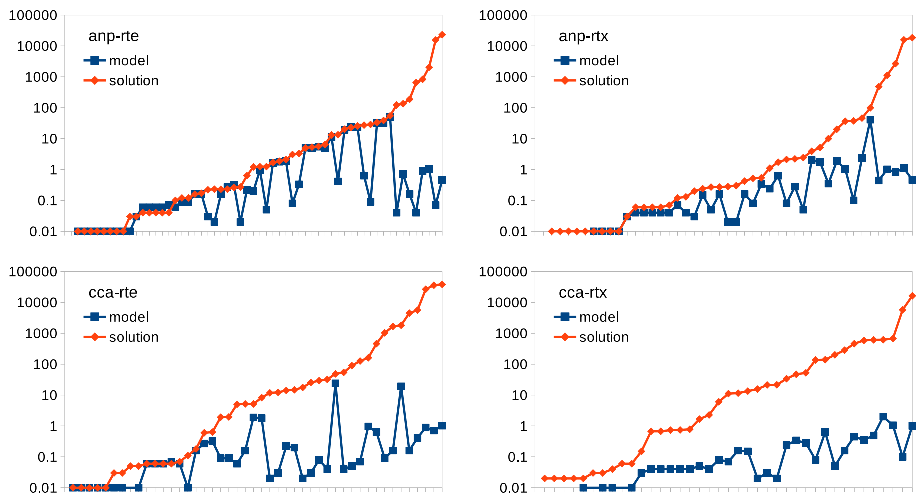


Fig. 12 Model building time required by Algorithm 1 for problem instances and total solution times that include the model building time. Problem instances (x-axis) sorted on solution times vs time in seconds (y-axis) for Top-Left anp-rte, Top-Right anp-rtx, Bottom-Left cca-rte, and Bottom-Right cca-rtx.

Interaction Between Gates

Fig. 6 Bottom-Right shows in the number of problems solved, ctype and dtype gates do not mutually help when both of them are used together. Also, Section 5.6 and Table 5 somewhat explain this with Pearson's correlation coefficients. Hence,

we have not used setting cd in our final experiments. Section 5.8 with Pearson's correlation coefficients and Fig. 8 show etype and xtype gates do not mutually help when both of them are used together. Although statistically insignificant (as per ANOVA), Figure 13 shows settings rtcx and rtdx of anp perform somewhat better than setting rtx of the same solver. For cca, setting rtcx is somewhat worse than setting rtx but setting rtdx is close to setting rtx.

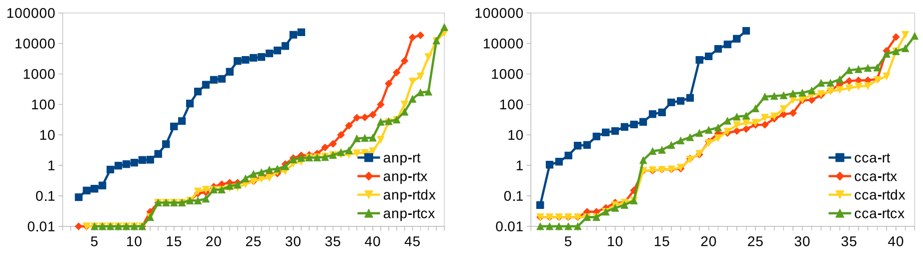


Fig. 13 Numbers of problem instances solved (x-axis) vs time in seconds (y-axis) for (left) anp and (right) cca solvers with various combinations of c, d and x settings assuming setting rt is also used.

Restart and Tabu

Using the pilot runs, Fig. 6 Top-Right shows that the tabu technique has a very clear impact on anp performance, but a marginal impact on cca performance. For both anp and cca, the restart method has a very small effect. We used both r and t in the final experiments.

The restart method may have considerably less effect within the 1-hour timeout of the pilot experiments. This is because the restart method is invoked when a plateau of a given length (e.g. 10,000) is encountered. So to check the long term effect of the restart method, we perform further final runs with and without the restart method and use ANOVA plus HSD tests. Fig. 14 left shows that anp-rte and anp-te have similar performances. Settings anp-rt and anp-t have similar performances as well. In the same figure at the right, we observe the similar performances for cca settings, e.g. cca-rt vs cca-t and cca-rte vs cca-te. We also see the effect of the tabu technique in the long run of the cca settings. Performances of cca-rt and cca-r are close and that of cca-rte and cca-re are also close.

Performance of ptype Gates

Fig. 8, 9, and 10 show the performance of ptype gates. Fig. 13 and 14 even clearly show the difference xtype and etype gates can make. Table 4 shows that p32, p16, log, bmc, ssa, hc, and app problem instances have large numbers of etype gates. The same table also shows that ssa, p32, p16, bmc, and hc problem instances have (often large numbers of) xtype gates.

1. For etype gates, we have found that when a number of variables are equivalent to each other, the baseline setting of both solvers spend significant numbers of iterations to get the same value for each of the variables in the equivalence

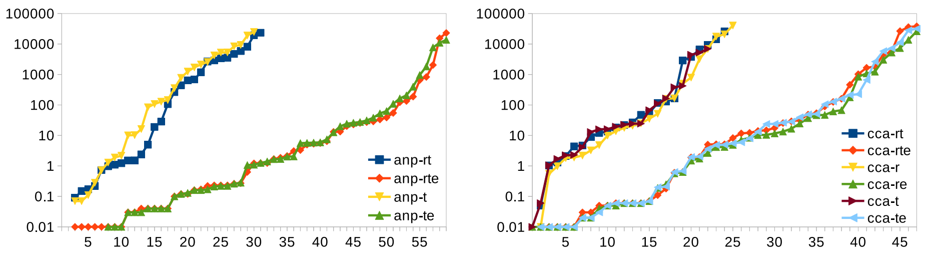


Fig. 14 Numbers of problem instances solved (x-axis) vs time in seconds (y-axis) for (left) anp and (right) cca solvers with various combinations of r and t and e settings

group. Given such an equivalence group, the number of variables having the same value true or false goes up and down over iterations, while it should be either 0 or the number of variables in the group. We then have created a simple problem instance having a chain of equivalent variables and found a similar behaviour. Detection of etype gates and then performing simplification alleviates this issue.

2. For xtype gates, we have found that when a variable is in the clausal pattern of a number of xtype gates, the baseline setting of both the solvers spend significant numbers of iterations to get a value that could be consistent with all other variables. For example, in p16 problem instances, we have found that such variables are flipped significantly more times than other variables. Detection of xtype gates and functionally computing the output variables and then propagating these values through the chains of such gates alleviates this issue.

The above findings help us explain why ptype gates led to the enormous performance difference.

Performance of atype Gates

Section 5.6 and Table 5 with Pearson's correlation coefficients and Fig. 6 Bottom-Right in the number of problem instances solved show that ctype and dtype gates do not mutually help when both of them are used together. Moreover, not using atype gates even performs better in the number of problem instances solved than using them. Fig. 9 and Table 10 show similar results. With option e, adding options c or d worsens the performance while with option x, adding options c or d somewhat improves. However, option c or d along with option e produces better results than along with option x.

To understand why atype gates do not help much compared to xtype gates, we have found that local symmetries at the gate level plays a significant role. In an atype gate with n inputs, only 1 out of 2^n input combinations produces the desired output (e.g. true for and and false for or) at any time. In an xtype gate with n inputs, 2^{n-1} out of 2^n input combinations produce the desired output. So there is no symmetry in an atype gate for the desired output while an xtype gate has an abundance of symmetry. Absence of symmetries or breaking symmetries have negative effects on local search (Prestwich and Roli 2005). Local search wanders around the search space and the more the solutions, the more the likelihood of finding a solution. When symmetries are absent or broken, the effects could be extremely strong, slowing down local search performance by several orders of magnitude. One great example is Golomb Rulers (Prestwich 2002; Polash et al. 2017) that have only a handful number of solutions in the enormous search space and local search struggles hard. Because of lack of symmetries, in atype gates, changing the output from the undesired one to the desired one could often need changing more than one input and essentially more than one involving variable. However, in our approach, we consider the impact variables such that changing just one of them could change the output of a gate. Clearly, we do not have any impact variables for an atype gate when its output will change to the desired one if more than one variable needs flipping. Designing moves involving multi-variable flips might address this issue, but the computation of potential multi-variable moves will incur further overhead while even the single-variable moves already pose a trade-off. Further investigation in this aspect is out of scope of this paper.

Overall Performance

The gtype settings solve hc, app, bmc, and p32 problem instances which are not solved by o, l, or rt settings of the solvers. There are 39 problem instances (12 hc, 9 app, 13 bmc, and 5 p32) of these types and these are very hard instances for the AdaptNovelty+ and CCAnr solvers. The gtype settings of both solvers could solve most of them.

5.11 Additional Experiments

We compare our purely local search solvers with the winner of the SAT Competition 2020 kissat (Biere et al. 2020), which is a CDCL based solver. We run two sets of experiments for this comparison: one set on the satlib and sc14 problem instances and the other set on the sc20 problem instances.

Comparison on satlib and sc14 Problem Instances

Fig. 15 shows the performance of kissat (Biere et al. 2020), on our 69 satlib and sc14 problem instances. Fig. 15 also shows the selected best settings of anp and cca solvers. These experiments are run with 5000 seconds timeout.

Note that out of 69 problem instances, kissat solves 63 problem instances while anp-rte solves 59 problem instances. In particular, kissat cannot solve 4 out of 5 p32 instances of satlib, while some anp and cca settings solve them within a fraction of a second. Moreover, kissat solves 6 out of 21 sc14 instances taking more than 1 minute and 2 remains unsolved. Overall, kissat takes more than 1 second to solve 19 out of 69 of these problem instances, more than 1 minute to solve 7 problem instances, and 6 problem instances remain unsolved with 5000 seconds timout. These show that our satlib and sc14 problem instances still pose significant challenges to the state-of-the-art SAT solvers such as kissat. Nevertheless, this comparison is mainly to show how our gate constraint based local search solvers compare with the state-of-the-art SAT solvers, and not to claim better performances.

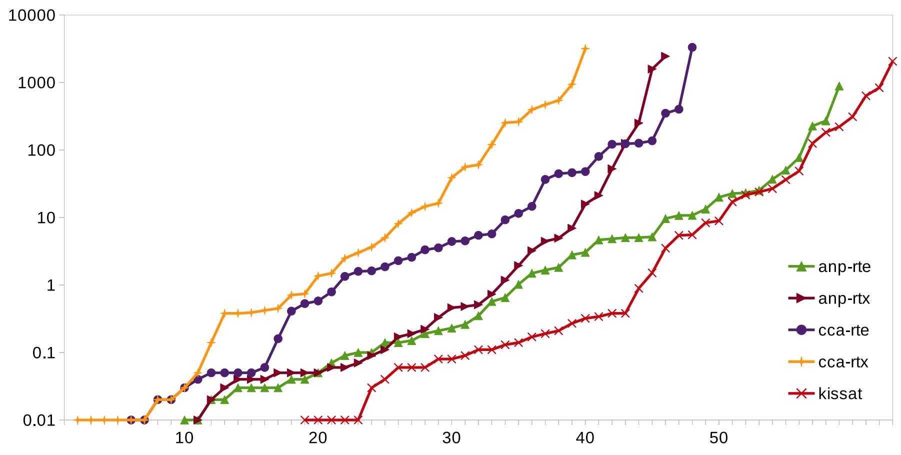


Fig. 15 Numbers of problem instances solved (x-axis) vs time in seconds (y-axis) for kissat solver and for selected best settings of anp and cca solvers.

Comparison on sc20 Problem Instances

We run kissat, anp, and cca solvers on the problem instances from the main track of SAT Competition 2020 (https://satcompetition.github.io/2020/downloads/sc2020-main.uri). These experiments are based on running each solver setting on each problem instance once as is done in SAT competitions (Cai et al. 2015). We have used 32GB memory limit and 5000 seconds time limit for each run.

SAT Competition 2020 has 400 problem instances but for various reasons we show the performances of the solvers on 107 problem instances. Below we describe the filtering process used in selecting these 107 problem instances.

1. **Unsatisfiable Problem Instances:** Purely local search solvers are incomplete and hence cannot prove that a problem instance is unsatisfiable. So for meaningful experiments with our anp and cca solvers, we have to use only the satisfiable problem instances. The 400 problem instances mentioned above include both satisfiable and unsatifiable problem instnaces. To identify the unsatisfiable ones, we run kissat on all 400 problem instances since as a CDCL based solver kissat can prove unsatisfiability. We have found that within the resource (memory and time) limits, kissat has proved 126 problem instances to be unsatisfiable. The remaining 374 problem instances could still have unsatisfiable ones. So we consider only the satisfiable problem instances solved by kissat and those that could not be solved by kissat within the resource limits but are solved by at least one of anp and cca solver settings.
2. **Gate Count and Detection Time:** We perform get detection on the remaining 374 instances with a timeout of 1 minute. From the problem instances with the gate detection process completed within the timeout, we ignore the problem instances having no gates detected. Since we are interested in using gate constraints with local search solvers, problem instances with no gate are not relevant. Nevertheless, this filtering gives us about 139 problem instances that have some gates. The numbers of gates in these problem instances range from 40 to 313686 and with an average of 21734.

After the filtering process, from the 139 problem instances obtained, we have found that 32 problem instances could not be solved by kissat or any of the anp and cca solver settings. We divide the remaining 107 problem instances into two groups sc20m and sc20f based on whether the problem instances have at least 100 (in other words, 100 or more) etype gates or fewer. This grouping allows us to see the effect of etype gates on the solver performances as we have seen them to be the key factor in our detailed experiments.

sc20m: There are 60 problem instances each with 100 or more etype gates.
sc20f: There are 47 problem instances each with fewer than 100 etype gates.

Table 12 Summarised performance of various anp and cca settings on sc20 problem instances where kissat has solved in total 74 problem instances; the best settings are emboldened.

on 107 sc20 problem instances												
solver	o	l	rt	rte	rtxe	rtde	rtdxe	rtce	rtcxe	rtx	rtdx	rtcx
anp	2	3	2	**46**	32	**47**	27	**48**	24	7	14	29
cca	12	9	0	**40**	25	**40**	15	**48**	25	8	21	23

on 60 sc20m problem instances having at least 100 etype gates											
kissat	solver	rt	rte	rtxe	rtde	rtdxe	rtce	rtcxe	rtx	rtdx	rtcx
37	anp	0	**44**	30	**44**	26	**40**	24	6	12	16
	cca	0	**40**	25	**40**	15	**40**	25	4	11	19

on 47 sc20f problem instances having fewer than 100 etype gates											
kissat	solver	rt	rte	rtxe	rtde	rtdxe	rtce	rtcxe	rtx	rtdx	rtcx
37	anp	2	2	2	3	1	**8**	0	1	2	**13**
	cca	0	0	0	0	0	**8**	0	4	**10**	4

Table 12 (top part) shows the summarised performance of various anp and cca solver settings on 107 sc20 problem instances. Note that kissat has solved in total 74 problem instances. Among anp settings, rte, rtde, and rtce are the best performing ones solving 46, 47 and 48 problem instances respectively. Among cca settings, rte, rtde and rtce are the best performing solving 40, 40, and 48 settings respectively. Table 12 (middle part) shows the summarised performance of the solvers on 60 sc20m problem instances while Table 12 (bottom part) shows on 47 sc20f problem instances. For sc20m, rte, rtde, and rtce are the best settings for both anp and cca solvers. For sc20f, rtce and rtcx are the best settings for anp while rtce and rtdx are for cca. For both sc20m and sc20f, kissat has solved 37 instances. In the numbers of problem instances solved, while anp and cca solvers have performed better than kissat on sc20m problem instances, they are far worse than kissat on sc20f problem instances. From Table 12 (top part), we see that rte performs better than rtxe, which performs better than rtdxe and rtcxe. In the absence of option e, we see rtdx and rtcx perform better than rtx.

Fig. 16 (top chart) shows kissat and the seletected best settings rte, rtce, and rtde of both anp and cca solvers on sc20m problem instances. Clearly, anp and cca solver settings are very significantly better than kissat (as per ANOVA plus HSD tests). Since rte, rtce, rtde settings of anp and cca solvers outstanding performance in the solutions times on sc20m problem instances, we compute Pearson's correlations coefficients among these settings and show the coefficients in Table 13. We see very strong correlation between rtde and rtce with the coefficient 1.00 while high correlation between rte and the other settings with the coefficients between

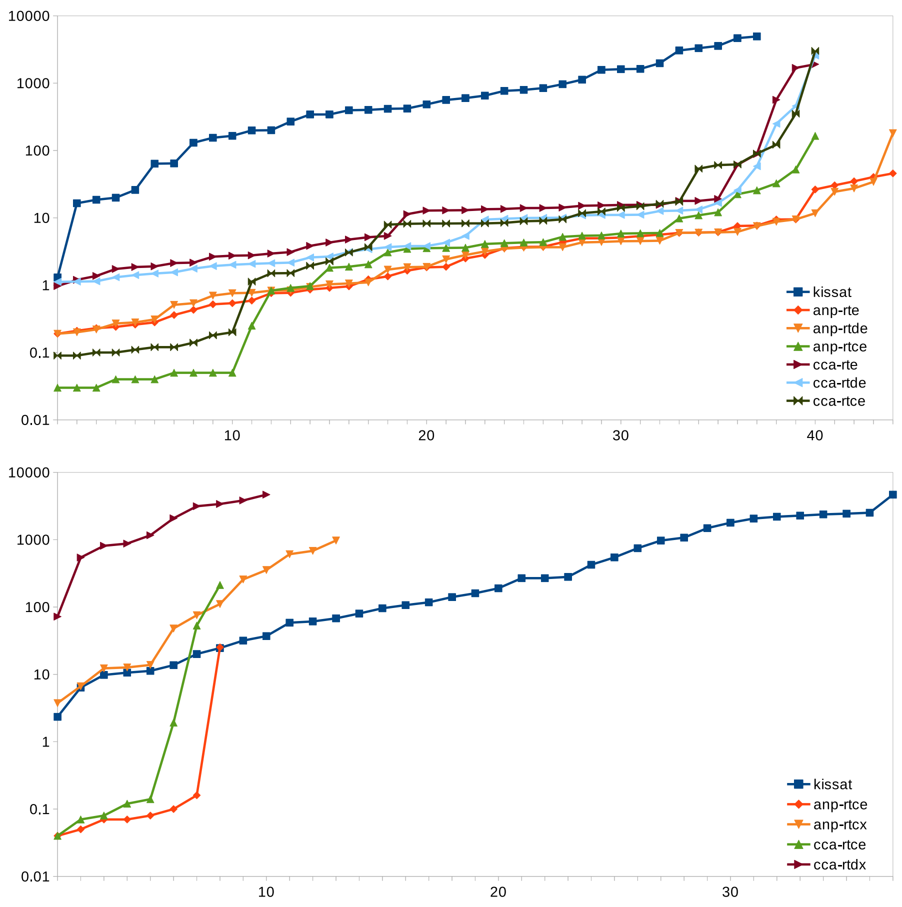


Fig. 16 Numbers of problem instances solved (x-axis) vs time in seconds (y-axis) by kissat and selected anp and cca solver settings on problem instances having at least 100 etype gates (top chart) and fewer than 100 etype gates (bottom chart)

0.67 and 0.73. Also, there is no significant difference among these settings (as per ANOVA test). Fig. 16 (bottom chart) shows kissat and the selected best settings anp-rtce, anp-rtcx, cca-rtce, and rtdx of our solvers on sc20f problem instances. Clearly, kissat shows the dominance by solving much more problems although for the first 10-15 problem instances solved by any solver or setting, anp-rtce and cca-rtce show dominance.

Table 13 Pearson's correlation coefficients among rte, rtde, rtce settings of anp and cca solvers on sc20m problem instances

anp			cca		
	rte	rtde		rte	rtde
rtde	0.70		rtde	0.73	
rtce	0.67	1.00	rtce	0.68	1.00

Overall, the conclusions from our detailed experiments qualitatively remain the same in the additional experiments.

6 Conclusions

Structured satisfiability problems, in their conjunctive normal (CNF) forms, contain logic gate patterns. Boolean circuits (BC) then can be obtained using the detected logic gates and local search algorithms can be adapted to BCs. However, it is not known which logic gates are useful to local search the most and why. In this work, we empirically evaluate the effect of using various types of logic gate constraints in local search algorithms. We also study the interactions among such logic gates. To show all these, we adapt two state-of-the-art local search families AdaptNovelty+ and CCANr to logic gate constraints. We describe how variable dependencies that comprise logic gate constraints can be heuristically made cycle free and then be represented by a directed acyclic graph. We then develop an algorithm to statically propagate equivalence of variables through the directed acyclic graph. We implement our satisfiability local search algorithms on top of a constraint-based local search system that allows dynamic propagation of changes from the input variables of the logic gates to their output variables. For experiments, we use benchmark instances from SATLib, SAT2014, and SAT2020. We empirically show that exploitation of xor, xnor, eq, and not gates is a key factor behind the performance of local search algorithms using single variable flips when adapted to logic gate constraints. Controlled experiments and investigations into the variables selected for flipping further elucidates these findings.

Acknowledgement

This research is partially supported by the Australian Research Council under the grant number DP150101618.

References

Balint A, Fröhlich A (2010) Improving stochastic local search for sat with a new probability distribution. In: International Conference on Theory and Applications of Satisfiability Testing, Springer, pp 10–15

Balyo T, Fröhlich A, Heule MJ, Biere A (2014) Everything you always wanted to know about blocked sets (but were afraid to ask). In: International Conference on Theory and Applications of Satisfiability Testing, Springer, pp 317–332

Balyo T, Froleyks N, Heule MJ, Iser M, Järvisalo M, Suda M (eds) (2020) Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions, University of Helsinki, Department of Computer Science

Battiti R, Tecchiolli G (1994) The reactive tabu search. ORSA journal on computing 6(2):126–140

Belov A, Stachniak Z (2009) Improving variable selection process in stochastic local search for propositional satisfiability. SAT 9:258–264

Belov A, Stachniak Z (2010) Improved local search for circuit satisfiability. SAT 6175:293–299

Belov A, Järvisalo M, Stachniak Z (2011) Depth-driven circuit-level stochastic local search for SAT. In: IJCAI, pp 504–509

Biere A (2016) Splatz, lingeling, plingeling, treengeling, yalsat entering the sat competition 2016. Proc of SAT Competition pp 44–45

Biere A, Fazekas K, Fleury M, Heisinger M (2020) CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo T, Froleyks N, Heule M, Iser M, Järvisalo M, Suda M (eds) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions, University of Helsinki, Department of Computer Science Report Series B, vol B-2020-1, pp 51–53

Cai S, Luo C, Su K (2015) CCAnr: A configuration checking based local search solver for non-random satisfiability. In: Proceedings of SAT, M. Heule and S. Weaver (Eds), LNCS, vol 9340, pp 1–8

Cook SA (1971) The complexity of theorem-proving procedures. In: Proceedings of the third annual ACM symposium on Theory of computing, ACM, pp 151–158

Davis M, Logemann G, Loveland D (1962) A machine program for theorem-proving. Commun ACM 5(7):394–397, DOI 10.1145/368273.368557

Fu H, Wu G, Liu J, Xu Y (2020) More efficient stochastic local search for satisfiability. Applied Intelligence pp 1–20

Fu H, Xu Y, Wu G, Liu J, Chen S, He X (2021) Emphasis on the flipping variable: Towards effective local search for hard random satisfiability. Information Sciences 566:118–139

Fu Z, Malik S (2007) Extracting logic circuit structure from conjunctive normal form descriptions. In: VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on, IEEE, pp 37–42

Heule MJ, Järvisalo MJ, Suda M, et al. (eds) (2018) Proceedings of sat competition 2018: Solver and benchmark descriptions, Department of Computer Science, University of Helsinki

Hoos HH (2002) An adaptive noise mechanism for walksat. In: AAAI/IAAI, pp 655–660

Hoos HH, Stützle T (2000) Local search algorithms for SAT: An empirical evaluation. Journal of Automated Reasoning 24(4):421–481

Hoos HH, Tompkins DA (2007) Adaptive novelty+. SAT Competition

Hoos HH, et al. (2002) An adaptive noise mechanism for WalkSAT. In: AAAI/IAAI, pp 655–660

Iser M, Manthey N, Sinz C (2015) Recognition of nested gates in CNF formulas. In: International Conference on Theory and Applications of Satisfiability Testing, Springer, pp 255–271

Järvisalo M, Junttila T (2009) Limitations of restricted branching in clause learning. Constraints 14(3):325–356

Järvisalo M, Niemelä I (2008) The effect of structural branching on the efficiency of clause learning sat solving: An experimental study. Journal of Algorithms 63(1-3):90–113

Järvisalo M, Junttila TA, Niemelä I (2008a) Justification-based local search with adaptive noise strategies. In: LPAR, Springer, pp 31–46

Järvisalo M, Junttila TA, Niemelä I (2008b) Justification-based non-clausal local search for SAT. In: ECAI, pp 535–539

Järvisalo M, Biere A, Heule MJ (2012) Simulating circuit-level simplifications on CNF. Journal of Automated Reasoning 49(4):583–619

Jeroslow RG, Wang J (1990) Solving propositional satisfiability problems. Annals of Mathematics and Artificial Intelligence 1(1):167–187, DOI 10.1007/BF01531077

Kuehlmann A, Paruthi V, Krohm F, Ganai MK (2002) Robust boolean reasoning for equivalence checking and functional property verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21(12):1377–1394

Luo C, Hoos H, Cai S (2020) Pbo-ccsat: Boosting local search for satisfiability using programming by optimisation. In: International Conference on Parallel Problem Solving from Nature, Springer, pp 373–389

Manthey N (2012) Coprocessor 2.0-a flexible CNF simplifier-(tool presentation). SAT 7317:436–441

Manthey N, Stephan A, Werner E (2016) Riss 6 solver and derivatives. Proceedings of SAT Competition pp 56–57

Mazure B, Sais L, Grégoire É (1997) Tabu search for SAT. In: AAAI/IAAI, pp 281–285

McAllester D, Selman B, Kautz H (1997) Evidence for invariants in local search. In: AAAI/IAAI, Rhode Island, USA, pp 321–326

Newton M, Pham D, Sattar A, Maher M (2011) Kangaroo: An efficient constraint-based local search system using lazy propagation. CP LNCS, Springer, Heidelberg 6876:645–659

Ostrowski R, Grégoire É, Mazure B, Sais L (2002) Recovering and exploiting structural knowledge from CNF formulas. In: International Conference on Principles and Practice of Constraint Programming, Springer, pp 185–199

Patterson DJ, Kautz H (2001) Auto-WalkSAT: a self-tuning implementation of WalkSAT. Electronic Notes in Discrete Mathematics 9:360–368

Peng C, Xu Z, Mei M (2020) Applying aspiration in local search for satisfiability. PloS one 15(4):e0231702

Pham DN, Thornton J, Sattar A (2007) Building structure into local search for SAT. In: IJCAI, vol 7, pp 2359–2364

Plaisted DA, Greenbaum S (1986) A structure-preserving clause form translation. Journal of Symbolic Computation 2(3):293–304

Polash MA, Newton MH, Sattar A (2017) Constraint-based search for optimal golomb rulers. Journal of Heuristics 23(6):501–532

Prestwich S (2002) Supersymmetric modeling for local search. In: Second International Workshop on Symmetry in Constraint Satisfaction Problems, Citeseer

Prestwich S, Roli A (2005) Symmetry breaking and local search spaces. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, Springer, pp 273–287

Roy JA, Markov IL, Bertacco V (2004) Restoring circuit structure from SAT instances. Ann Arbor 1001:48109–2122

Ryvchin V, Nadel A (2018) Maple lcm dist chronobt: Featuring chronological backtracking. Proceedings of SAT competition 2018

Ryvchin V, Strichman O (2008) Local restarts in sat. Constraint Programming Letters (CPL) 4:3–13

Seltner H (2014) Extracting hardware circuits from CNF formulas. Master's thesis, Institute for Formal Models and Verification

Soos M, Devriendt J, Gocht S, Shaw A, Meel KS (2020) Cryptominisat with ccanr at the sat competition 2020. SAT COMPETITION 2020 p 27

Tseitin GS (1983) On the complexity of derivation in propositional calculus. In: Automation of reasoning, Springer, pp 466–483