# March_cluster: Improving The Performance Of Lookahead Solvers On SAT Problems With A Large Diameter

**C. Bezemer** C.Bezemer@student.tudelft.nl
**L. van Driel** L.vandriel@student.tudelft.nl
**J. Fresen** J.Fresen@student.tudelft.nl
**B.J. Schaafsma** B.J.Schaafsma@student.tudelft.nl
**R.J.T. Verwoerd** R.J.T.Verwoerd@student.tudelft.nl
*Faculty of Electrical Engineering, Mathematics and Computer Science,*
*Delft University of Technology*

## Abstract

Given the observation that lookahead SAT solvers perform relatively bad on problems with large diameters, we propose two new heuristics to improve their performance on such problems, Cluster Based Preselection (CBP) and Cluster Based Conflict Analysis (CBCA). We divide the variables into relatively independent SAT problems (clusters) based on the variable interaction graph, solve the clusters one after the other (CBP) and add locally applicable conflict clauses when a conflict arises (CBCA). We integrated the heuristics in March_ks and the results show that we get a slight increase in the number of solved benchmarks and a huge decrease of solving time.

Keywords: *SAT solver, lookahead, clustering, cluster based preselection, cluster based conflict analysis*

## 1. Introduction

Scientists have been looking for ways to solve satisfiability (SAT) problems in reasonable time for many years. As there is no known polynomial algorithm for SAT, current solvers use heuristics to narrow down the search space of large SAT problems.

In 2002 the first SAT competition was organized to find out which automated SAT solver performed best on different types of SAT problems[1.]. Every year the creators of these solvers try to improve their software to outperform the other solvers in the competition.

One of these solvers is March_ks [2], created by M. Heule and H. van Maaren, which performs excellent in two out of three main categories, the crafted and random, but poor in the industrial problems category. Problems in the industrial category are often characterized by the relative independency between variables. This independency is caused by the large diameter of the problem. The diameter of a problem is the distance between the two variables of the problem that lie the furthest away from each other.

Conflict-driven solvers perform well in the industrial category. We believe that the large diameter of many of the problems enables those solvers to create short, often used conflict clauses. In this paper we will present two techniques to improve the performance

1. see http://www.satcompetition.org/

of March_ks in the industrial problem category: Cluster Based Preselection (CBP) and Cluster Based Conflict Analysis (CBCA). Our result is the new solver March_cluster.

We will first give a short introduction on the SAT problem and conventional algorithms for solving it in section 2. Section 3 presents CBP and its implementation in March_cluster. In section 4 we will discuss CBCA and its implementation in March_cluster, including conflict clause generation, minimization and directed back jumping. We will present a case study of CBP and CBCA in section 5. A complete overview of the use of CBP and CBCA in March_cluster is given in section 6. We will present the results of our research and show a comparison with the original March_ks solver in section 7. We will finish this paper with our recommendations for future research in section 8.

## 2. Preliminaries

This section gives a formal definition of, and a basic introduction to the SAT problem and the notation we will use. Next it will focus on the basic architecture of the two main solving algorithms applied today.

### 2.1 Satisfiability

SAT is the problem of determining if there exists an assignment for a given Boolean formula that makes the formula evaluate to *true*. If such an assignment does not exist, we call the problem unsatisfiable (UNSAT).

In a more formal setting we have a CNF formula $\mathcal{F} = \{cl_0 \wedge \ldots \wedge cl_m\}$ consisting of clauses $cl_i$ in conjunctive form. Clause $cl_i = (l_{i,1} \vee \ldots \vee l_{i,k})$ consists of literals in disjunctive form. Each literal $l$ equals a variable $x_i$ or the negation of a variable $\neg x_i$, where $i = 1, \ldots, n$ the number of variables in $\mathcal{F}$. Because of the disjunctive form, a clause is satisfied when at least one literal evaluates to *true*. Each clause must be satisfied in order for the formula to be satisfied and the problem to be classified as SAT.

The SAT solvers within our scope solve SAT problems by searching for an assignment of variables $x_1, \ldots, x_n \in \{\texttt{true}, \texttt{false}\}$ that makes $\mathcal{F}$ evaluate to *true*.

### 2.2 Formula Simplification and Unit Propagation

To decrease the solving time, a process of simplification can be applied to $\mathcal{F}$ after the assignment of a variable. When variable $x_j$ is set, we can remove any clause $cl_i$ from $\mathcal{F}$ that contains a literal $l_{i,j}$ or $\neg l_{i,j}$ that evaluates to true since this clause is satisfied. Any literal $l_{i,j}$ or $\neg l_{i,j}$ that evaluates to false can be removed from clause $cl_j$ since this clause can no longer be satisfied by this literal.

These simplifications can result in clauses containing only one literal, called unit clauses. Since there is only one assignment possible that satisfies this clause, this assignment has to be made. These forced assignments are called unit propagations [2]. Since propagation can result in new unit clauses this process repeats until there are no more unit clauses left or there is an empty clause. If unit propagation result in an empty clause, then under the current assignment of $\mathcal{F}$ this clause can never be satisfied, so another assignment of the variables must be tried.

The process of setting variables in unit clauses and simplifying $\mathcal{F}$ is done, in most solvers, by the function IterativeUnitPropagation (IUP). The pseudo code of this algorithm is described in algorithm 1.

**Algorithm 1** IteratitiveUnitPropagation($\mathcal{F}$)

1: **while** $\mathcal{F}$ does not contain empty clause **and** an unit clause $y \in \mathcal{F}$ does exist **do**
2:     satisfy y and simplify $\mathcal{F}$
3: **end while**
4: **return** $\mathcal{F}$

### 2.3 Lookahead architecture

There are two main architectures used by SAT solvers these days. The first one, the lookahead architecture, is shown in algorithm 2. It is an almost direct implementation of Davis-Putman-Logemann-Loveland procedure (DPLL) [1]. Its effectiveness heavily depends on the implementation of the GetBranchVariable() function. This function is implemented in such a way that it returns the (potentially) most successful branch variable, based on heuristics. The way it determines the most successful branch variable is with a function called lookahead.

To paraphrase [2]: Lookahead evaluates for every variable $x_i$ in $\mathcal{F}$ both $IUP(\mathcal{F} \cup \{x_i\})$ and $IUP(\mathcal{F} \cup \{\neg x_i\})$. If the lookahead on either the positive or the negative literal results in a conflict, this literal is called a failed literal. This literal needs to be fixed on complement of its current assignment. When the lookahead on both $x_i$ and $\neg x_i$ result in a conflict, then $\mathcal{F}$ is unsatisfiable. When both lookaheads on a variable do not result in a conflict, the lookahead will be evaluated and a value will be assigned to the literal, which indicates its potential. This value can be based on a number of different heuristics, such as the number of created binary clauses by this assignment.

Lookahead is a very expensive procedure, therefore lookahead is usually only performed on a subset of the literals in $\mathcal{F}$, $\mathcal{P}$. This subset $\mathcal{P}$ is determined by the function $Preselect()$. This preselection is done by approximating for each literal the value that literal would get during the lookahead phase. Then the best literals are put in $\mathcal{P}$.

### 2.4 Conflict-driven architecture

The second commonly used architecture is the conflict-driven architecture, shown in algorithm 3. The most significant change from the lookahead architecture is that it does not use chronological backtracking. In this architecture, when a conflict is detected during solving, the solver uses the AnalyzeConflicts function to determine the source and level -in the search tree- of the current conflict. After determining this, a conflict clause is added to $\mathcal{F}$, which represents the state of the solver when the conflict was encountered. In the end the solver backtracks to the level of the current conflict and continues solving. When a conflict is detected at level 0 this means that every possible assignment of the variables in $\mathcal{F}$ results in UNSAT and hence $\mathcal{F}$ is UNSAT.

**Algorithm 2** RecursiveDPLL($\mathcal{F}$)

1: $\mathcal{F} \leftarrow IterativeUnitPropagation(\mathcal{F})$
2: **if** $\mathcal{F} = \emptyset$ **then**
3: **return** SAT
4: **else if** $\mathcal{F}$ contains empty clause **then**
5: **return** UNSAT
6: **end if**
7: variable set $\mathcal{P} \leftarrow Preselect()$
8: $x \leftarrow GetBranchVariable(\mathcal{P})$
9: **if** $RecursiveDPLL(\mathcal{F} \cup \{x\})$ **then**
10: **return** SAT
11: **end if**
12: **return** $RecursiveDPLL(\mathcal{F} \cup \{\neg x\})$

**Algorithm 3** IterativeDPLL($\mathcal{F}$)

1: $\mathcal{F} \leftarrow IterativeUnitPropagation(\mathcal{F})$
2: **while** TRUE **do**
3: $l_i \leftarrow GetBranchLiteral()$
4: **if** a $l_i$ is selected **then**
5: $\mathcal{F} \leftarrow IterativeUnitPropagation(\mathcal{F} \cup \{l_i\})$
6: **while** $\mathcal{F}$ *contains empty clause* **do**
7: $blevel \leftarrow AnalyzeConflicts()$
8: **if** $blevel = 0$ **then**
9: **return** UNSAT
10: **else**
11: $BackTrack(blevel)$
12: $\mathcal{F} \leftarrow IterativeUnitPropagation(\mathcal{F})$
13: **end if**
14: **end while**
15: **else**
16: **return** SAT
17: **end if**
18: **end while**

## 3. Cluster Based Preselection

In this section we will present a new heuristic for the preselection procedure. We start by introducing a new abstract graph representation of a SAT problem, based on which we then propose our new heuristic. Then follows the implementation of the heuristic in March_cluster.

### 3.1 Diameter Based Cluster Representation

In the field of graph theory many visualization and decomposition techniques exist. Since SAT problems can be represented as a graph, many of these techniques can also be applied

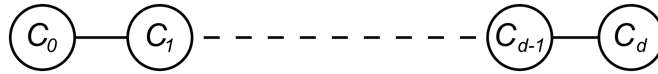


**Figure 1.** The Diameter Based Cluster Representation (DBCR)

to SAT problems. In [6], Herwig has used some of these techniques to research the relation between the diameter, the shortest longest path in the graph representation, of a SAT problem and the performance of SAT solvers. The diameter gives us insight in the fundamental structure of a problem, even though the diameter of a SAT problem is not unambiguously defined (different graph representations can lead to slightly different diameters).

The graph of a SAT problem with a large diameter has a stretched structure. This stretched structure indicates that nodes in the graph are mostly connected locally. That means that the problem is essentially divided in several 'small' SAT problems, which are only connected to their neighbouring SAT problems and therefore relatively independent of all other nodes. We propose a heuristic to solve these problems one after the other.

To be able to use this heuristic, we need to identify these 'small' SAT problems. This can be done by taking a graph representation of the SAT problem and contract groups of nodes into single new nodes, which ultimately lead to a string of such new nodes. We call these new nodes *clusters* and their representation in a string the *Diameter Based Cluster Representation* (DBCR) (see figure 1). The set of nodes is partitioned into clusters such that the number of clusters is $d + 1$, which means that the diameter of the DBCR equals the diameter of the original graph. If two nodes are connected in the original graph, then they will be either in the same cluster or in two neighbouring clusters.

When solving $\mathcal{F}$, we start by solving the first cluster, followed by the next, until the entire problem is solved. Should a conflict arise while a cluster is being solved (the *current cluster*), this is caused by the previous cluster, because that is the only cluster that can influence the current cluster. Therefore, we need to find another solution for the previous cluster in order to solve the conflict in the current cluster.

### 3.2 Preselection Based On DBCR

In order to solve the clusters one after the other, we adapt the preselection so the lookahead procedure can only select branch variables based on the lowest cluster with unassigned variables. This focuses the solver on the current cluster and has the side benefit that the preselection procedure takes less time because there are less variables to consider. We call this preselection heuristic *Cluster Based Preselecion*, or CBP.

### 3.3 Implementation of CBP in March_cluster

Because there are different graph representations of a SAT problem [2.], we need to choose between them to create the CBP. For practical purposes, we need a representation that requires little additional computation to create. The variable interaction graph (vig) is such a representation, because it is already implicitly defined in the data structures of March_ks and therefore requires no computation at all to create.

2. See [6] for an overview of the most common representations

A cluster based on the vig consists of several variables that are grouped together. There are many ways in which these groups can be created. In March_cluster, a rather simple way is used, in which each variable in cluster $C_i$ is connected in the vig to a variable in cluster $C_{i-1}$. Cluster $C_0$ is chosen to be a set of variables on the edge of the stretched structure of the vig. The variables that are on the edge can be found with the use of the diameter of the vig, as the diameter is the length of the path between two variables on the edge of the vig. When $C_0$ is known, all subsequent clusters are created by using the above inference rule.

### 3.4 Clustering Algorithm In March_cluster

To create the clustering, we start by calculating the diameter of the vig. This is done by performing Algorithm 4 for each of the variables in the graph. This gives us both the longest shortest path, and the variables where this path starts $x_s$. Next we have three ways of creating $C_0$. Each of the following methods will result in a $C_0$ that contains only variables that are the starting point of a longest shortest path.

- $C_0$ consists of variable $x_s$ (not used in March_cluster)
- $C_0$ consists of the variable(s) having $depth(x) = maxDiameter$
- $C_0$ consists of the variable(s) having $depth(x) = maxDiameter$ when running $getMaxDistance(\mathcal{F}, C)$,
  with $\mathcal{C} = \{x \in \mathcal{F} : depth(x) = maxDiameter\}$

In the first method, we cluster $\mathcal{F}$ starting from $x_e$ by running algorithm 4 with $\mathcal{C} = \{x_e\}$. Now $distance(\cdot)$ contains the cluster indices. Although this clustering is suitable for our SAT solver, we will not use it directly but feed it to a second way of clustering. Here we use the variable set $\mathcal{C} = \{x \in \mathcal{F} : distance(x) = maxDiameter\}$ to run $getMaxDistance(\mathcal{F}, \mathcal{C})$ again, in the opposite direction. We call this reverse clustering. The third way to cluster is to repeat the previous procedure which results in forward clustering. Repeating this procedure again might result in yet another clustering, but our large diameter problems do not show this behaviour.

**Algorithm 4** $getMaxDistance(\mathcal{F}, C)$

1: **for each** variable $x \in \mathcal{F}$ **do**
2: $distance(x) \leftarrow 0$
3: **end for**
4: $variableStack.push(\mathcal{C})$
5: **while** $variableStack \neq \emptyset$ **do**
6: variable $x \leftarrow variableStack.pop()$
7: $maxDistance \leftarrow distance(x)$
8: **for each** clause $cl \in \mathcal{F}$ containing $x$ **do**
9: **for each** variable $y \in cl$ with $distance(y) = 0$ and $y \notin variableStack$ **do**
10: $distance(y) \leftarrow maxDistance + 1$
11: $variableStack.push(y)$
12: **end for**
13: **end for**
14: **end while**
15: **return** $maxDistance$

## 4. Cluster Based Conflict Analysis

The concept of conflict clause analysis, storage and reasoning has proven to reduce the DPLL search space significantly [7]. However this reduction comes at a cost, the storage of and reasoning with conflict clauses increases the computational load of the solver. Even with this increase in computation the conflict-driven solvers prove to outperform lookahead solvers on problems with larger diameters [3.].

In this section, we will introduce our approach to conflict analysis and reasoning based on clustering called Cluster Based Conflict Analysis (CBCA). Next we discuss our approach for fast (un)satisfiability checking of conflict clauses, conflict clause minimization and back-jumping.

### 4.1 General Approach

The principle of conflict clauses is to remember some parts of the solver state after unit propagation results in an empty clause and the solver classifies the current state as UNSAT. By storing this state in a clause representing the conflict, we can recognize assignments that will lead to a future conflict and avoid running into the same conflict again. The construction of such a conflict clause is usually based on a complex analysis of the foregoing steps taken in the search tree. However due to our use of clusters, we can apply an easier and more straightforward approach.

To explain our approach, suppose when solving $\mathcal{F}$, all assignments of variables in cluster $C_i$ result in a conflict. This means that an assignment of a variable in $C_{i-1}$ resulted in the shortening of a clause with variables in $C_i$ and $C_{i-1}$ and now there are no more assignments of $C_i$ that do not result in an empty It is clear that should $\mathcal{F}$ be satisfiable, then the cause of the current conflict is the assignment of $C_{i-1}$.

3. For more details we refer the reader to the website of the SAT07 Competition: http://www.cril.univ-artois.fr/SAT07/

When we backtrack to cluster $C_{i-1}$, we can store the current assignment of cluster $C_{i-1}$ as a conflict clause for that cluster. We derive the conflict clause from the assignment $x_p \wedge \ldots \wedge x_q$ in cluster $C_{i-1}$ by taking the negation $\neg x_p \vee \ldots \vee \neg x_q$ - see algorithm 5. Should none of the negated literals of this newly created conflict clause be satisfied, there will be a conflict causing the solver to backtrack.

We believe that our approach of creating conflict clauses in this way, is an improvement over more complex implementations of AnalyzeConflict() in conflict-driven implementations because:

1. We only add conflict clauses when backtracking from $C_i$ to $C_{i-1}$, thus potentially resulting in significantly less added conflict clauses.

2. Conflict clauses created in this way can be stored per cluster, making it more easy to reason only with conflict-clauses applicable to the current cluster.

Furthermore we would like to point out that conflict clauses generated with our method are short locally applicable conflict clauses, which are considered in [6] to be the most powerful.

**Algorithm 5** $addConflictClause(cluster)$

1: $j \leftarrow 0$
2: create conflict clause $cc$
3: **for** $the\ number\ of\ variables\ in\ cluster$ **do**
4: $\quad cc_j = \neg C_{cluster,\, j}$
5: $\quad j{+}{+}$
6: **end for**
7: $MinimizeCC(cc)$

### 4.2 Watch Literals

In lookahead SAT solvers, including March_ks, the data structures that contain formula $\mathcal{F}$ are often fixed or have a tight dynamic interplay with other data structures. Therefore, the proposed addition of conflict clauses in March_cluster should be handled in a separate structure, with those clause reasoning algorithms that conflict clauses require. As mentioned in 4.1 the fact that a conflict clause, created by using CBCA, contains only -a subset of- literals of the cluster it originates from, allows us to store conflict clauses per cluster and validate them only when variables in this cluster are being assigned or backtracked.

For a conflict clause to be satisfied, we need to check whether at least one of the literals in the conflict clause is unassigned, or assigned and satisfied. Because this check should be performed after each variable assignment a fast algorithm is required to update conflict clauses. To avoid the iteration over all literals of each conflict clause for each update, we implemented the two point watch literal structure as described in [4].

A watch literal $wl$ is a pointer to a literal in a (conflict) clause, given that this literal is currently not falsified. In case the assignment of the variable of $wl$ changes into a falsified literal, we need to move it to the next literal which is unassigned or assigned and satisfied. If, during an update, there is no suitable literal available in this conflict clause, we know

this conflict cause is falsified and therefore the current assignment of $\mathcal{F}$ is falsified. Because the moving of a watch literal occurs only rarely compared to the frequency of updates, a watch literal delivers a speedup in the order of the cluster size.

An important part of solving algorithms is iterative unit propagation. This concept can also be applied to conflict clauses by propagating the last unassigned variable in a conflict clause when all other literals are unsatisfied. To detect these unit propagations effectively, we use watch literals $wl_l$ and $wl_r$ for each conflict clause, each watching a separate literal. These watch literals have the same behaviour as a single watch literal, with the exception that we allow $wl_l$ to be on an unsatisfied literal if there are no other literals available, except for the literal that is watched by $wl_r$. To see whether unit propagation is required, we check if $wl_l$ is unsatisfied and $wl_r$ is unassigned. Furthermore when a conflict clause is created the watch literals are placed on the last ($wl_r$) and fore last ($wl_l$) set literal in the clause. This placement is done to support backjumping. Backjumping will be discussed in more detail in section 4.4. A pseudo code implementation of the update algorithm can be seen in algorithm 6.

**Algorithm 6** $updateWatchliterals(x_i)$

```
1: j ← cluster of x_i
2: for every cc in C_j that has wl_l or wl_r watching ¬x_i or x_i do
3:    if wl_l is updated and wl_r = free then
4:       swap wl_l and wl_r
5:    end if
6:    if wl_r is satisfied then
7:       continue to next conflict clause
8:    end if
9:    if there is a satisfied or free variable that is not watched by wl_l then
10:      let wl_r watch the found free or satisfied variable
11:   else
12:      if IterativeUnitPropagation(F ∪ wl_l) = UNSAT then
13:         return UNSAT
14:      end if
15:   end if
16: end for
```

### 4.3 Minimization

One conflict clause is added each time we backtrack to a previous cluster. This can potentially result in a large ($> 500.000$) amount of clauses per cluster, where each clause contains all variables of its cluster. To lower the computational burden of introducing conflict clauses and increase their effectiveness, we can both reduce the number of clauses added and the number of literals in each clause. Given that we do not find it desirable to lose any information contained in these removed clauses and literals, we first apply a sound inference technique to create smaller clauses, then we remove clauses that are implicated by smaller ones.

Many inference techniques have been proposed in conflict-driven research. The high computational costs of complete inference techniques have led to a number of refined methods that deliver reasonable inference at a low cost. We applied the Minisat conflict clause minimization technique as presented in [5]. This technique minimizes new conflict clauses based on the previous unit propagation of existing ones. To quote [5]: "'For every literal p of the newly generated conflict-clause C, the algorithm tries to self subsume C by the reason clause for p. If successful, p can be removed."'

To increase the number of times the Minisat algorithm could be applied to our conflict clauses, we added a number of initial conflict clauses. These conflict clauses are equal to the clauses in $\mathcal{F}$ that have literals whose corresponding variables that are all in the same cluster.

Because there is no interaction between existing conflict clauses, Minisat minimization requires only one loop over all conflict clauses. The precise Minisat algorithm is shown in algorithm 7.

To keep the number of conflict clauses down, we follow the inference loop by a super set loop over all conflict clauses. This super set loop removes all existing conflict clauses that are a super set of the minimized clause and are therefore a direct implication of the newly created conflict clause.

**Algorithm 7** $strengthenCC(C)$

1: **for all** $p \in C$ **do**
2:    **if** $reason(\neg p) \backslash \{p\}$ **then**
3:       mark p
4:    **end if**
5:    remove all marked literals in $C$
6: **end for**

By reason(p) we denote the clause that became unit and propagated p = True

### 4.4 Backjumping

As mentioned in [7] if all the literals in a newly created conflict clause correspond to variables that were assigned at levels higher up in the DPLL tree than the current level, no satisfying assignment can be found until the search process backtracks to the decision level in which variables were set that are part of the newly created conflict clause.

This concept is incorporated in March_cluster in the function checkConsistency(). This function is used during backtracking, when new conflict clauses have been generated in one or more clusters. It guides backtracking to the proper point in the DPLL search tree.

Literals in a newly created conflict clause correspond to variables in the current level of the DPLL tree, when at least one literal's corresponding variable is free. This implies that for our implementation in March_cluster it is sufficient to check the literals that are watched by the watch literals of that clause. This holds since watch literals of newly created conflict clauses stand on the last and fore last set literals in that clause, which are the first literals to be freed during backtracking.

Should none of these literals correspond to free variables, than this conflict clause can never be satisfied and we need to backtrack further up the DPLL search tree.

Should one of these variables be free -this is the last set variable in the clause- than this variable must be set so the watched literal evaluates to true. Should this assignment result in a conflict then we need to backtrack further up the DPLL search tree.

The pseudo code implementation of our backjumping algorithm can be seen in Algorithm 8.

**Algorithm 8** checkConsistency()

1: **for** *each newly created cc* **do**
2:   **if** $wl_l = UNSAT\ AND\ wl_r = UNSAT$ **then**
3:     **return** UNSAT
4:   **else if** $wl_l = UNSAT\ AND\ wl_r = FREE$ **then**
5:     **if** $IterativeUnitPropagation(wl_r) = unsatisfiable$ **then**
6:       **return** UNSAT
7:     **end if**
8:   **end if**
9: **end for**

## 5. A case study: dubois6

To clarify the heuristics CBP and CBCA we will now apply the technique on the dubois6 benchmark by giving a demonstration of the steps taken in the process of solving. The problem consists of several XOR clauses shown in Table 1. The vig of this problem is shown in Figure 5.

**Table 1.** Dubois6 SAT problem

| Clauses | |
|---|---|
| $(a)\ x_1 \oplus x_2 \oplus x_{13}$ | |
| $(b)\ x_2 \oplus x_3 \oplus x_{14}$ | |
| $(c)\ x_3 \oplus x_4 \oplus x_{15}$ | |
| $(d)\ x_4 \oplus x_5 \oplus x_{16}$ | Note that: |
| $(e)\ x_5 \oplus x_{17} \oplus x_{18}$ | |
| $(f)\ x_6 \oplus x_{17} \oplus x_{18}$ | $x_1 \oplus x_2 \oplus x_3 = \begin{cases} x_1 \vee x_2 \vee x_3 \\ \neg x_1 \vee \neg x_2 \vee x_3 \\ \neg x_1 \vee x_2 \vee \neg x_3 \\ \neg x_1 \vee \neg x_2 \vee \neg x_3 \end{cases}$ |
| $(g)\ x_6 \oplus x_7 \oplus x_{16}$ | |
| $(h)\ x_7 \oplus x_8 \oplus x_{15}$ | |
| $(i)\ x_8 \oplus x_9 \oplus x_{14}$ | |
| $(j)\ x_9 \oplus x_{10} \oplus x_{13}$ | |
| $(k)\ x_{10} \oplus x_{11} \oplus x_{12}$ | |
| $(l)\ \neg x_1 \oplus x_{11} \oplus x_{12}$ | |

Since our goal is to clarify CBP and CBCA, we will solve the problem using the basic DPLL algorithm and only apply those two heuristics. That means we will not use the lookahead procedures. The first step now is to create the clusters. It can be seen in the vig that the longest shortest path can be found between for example $x_{11}$ and $x_{18}$ and has

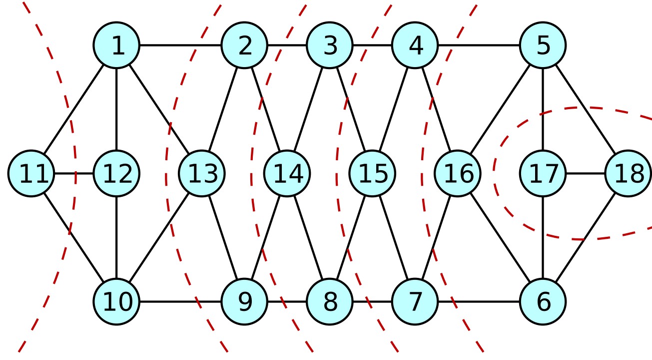


**Figure 2.** The vig of dubois6 and its division in clusters.

length 6. When creating the clusters we will use the first method described in (see Section 3) to create $C_0$, which becomes $x_{11}$. The other clusters are created like described in Section 3, this clustering results in the clusters as shown in Figure 5.

Table 2 shows how dubois6 is solved using CBP and CBCA. The setting of branch variables is shown in bold font, the variables that are set due to IUP are shown in plain font. When an assignment is falsified, this is indicated by an X. A plain X means that this variable must be set to both a 1 and a 0 at the same time due to IUP and a bold X means that this branch variable leads to an empty clause both when it is set to 1 and when it is set to 0. Added conflict clauses are shown in the column *added cc*, their number is shown in the column *cc id*. The column $C_c$ shows the current cluster, and the column $x_b$ shows the chosen branch variable.

In this example resolution is used for conflict clause minimization. Should resolution occur then this is indicated by the ids of the conflict clauses that were used in the resolution.

Table 2: Full description of the solving process for the dubois6 benchmark.

| $C_c$ | $x_b$ | $C_1$ | $C_2$ | | | $C_3$ | | | $C_4$ | | | $C_5$ | | | $C_6$ | | | $C_7$ | | **added cc** | **cc id** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **11** | **1** | **12** | **10** | **2** | **13** | **9** | **3** | **14** | **8** | **4** | **15** | **7** | **5** | **16** | **6** | **17** | **18** | | |
| $C_1$ | 11 | **1** | | | | | | | | | | | | | | | | | | | |
| $C_2$ | 1 | **1** | **1** | 0 | 0 | | | | | | | | | | | | | | | | |
| $C_3$ | 2 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | | | | | | | | | | | | | |
| $C_4$ | 3 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | | | | | | | | | | |
| $C_5$ | 4 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | | | | | | | |
| $C_6$ | 5 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | | | | |
| $C_7$ | 17 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | X | | |
| $C_7$ | -17 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | X | | |
| $C_7 \rightarrow C_6$ | | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **X** | | (-5 6 -16) | (1) |
| $C_6$ | -5 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | | | | |
| $C_7$ | 17 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **1** | X | | |
| $C_7$ | -17 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **0** | X | | |
| $C_7 \rightarrow C_6$ | | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **X** | | (5 -6 16) | (2) |
| $C_6 \rightarrow C_5$ | | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **X** | | | | | (-4 7 -15) | (3) |
| $C_5$ | -4 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | | | | | | | |
| $C_6$ | 5 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **1** | 0 | 0 | | | | |
| $C_7$ | 17 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **1** | 0 | 0 | **1** | X | | |
| $C_7$ | -17 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **1** | 0 | 0 | **0** | X | | |
| $C_7 \rightarrow C_6$ | | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **1** | 0 | 0 | **X** | | (-5 6 16) | (4) |
| | | | | | | | | | | | | | | | | | | | | (-5 6) | (5) (1+4) |
| $C_6$ | -5 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **0** | 1 | 1 | | | | |
| $C_7$ | 17 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **0** | 1 | 1 | **1** | X | | |
| $C_7$ | -17 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **0** | 1 | 1 | **0** | X | | |
| $C_7 \rightarrow C_6$ | | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **0** | 1 | 1 | **X** | | (5 -6 -16) | (6) |
| | | | | | | | | | | | | | | | | | | | | (5 -6) | (7) (2+6) |
| $C_6 \rightarrow C_5$ | | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **X** | | | | | (4 -7 15) | (8) |
| $C_5 \rightarrow C_4$ | | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **1** | 1 | 0 | **X** | | | | | | | | (-3 8 -14) | (9) |
| $C_4$ | -3 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | | | | | | | | | | |

Table 2: Full description of the solving process for the dubois6 benchmark.

| $C_c$ | $x_b$ | $C_1$ | $C_2$ | | | $C_3$ | | | $C_4$ | | | $C_5$ | | | $C_6$ | | | $C_7$ | | **added cc** | **cc id** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **11** | **1** | **12** | **10** | **2** | **13** | **9** | **3** | **14** | **8** | **4** | **15** | **7** | **5** | **16** | **6** | **17** | **18** | | |
| $C_5$ | 4 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **1** | 0 | 0 | | | | | | | |
| $C_6$ | 5 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **1** | 0 | 0 | **1** | 1 | X | | | | |
| $C_6$ | -5 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **1** | 0 | 0 | **0** | 0 | X | | | | |
| $C_6 \rightarrow C_5$ | | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **1** | 0 | 0 | **X** | | | | | (-4 7 15) | (10) |
| | | | | | | | | | | | | | | | | | | | | (-4 7) | (11) (3+10) |
| $C_5$ | -4 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **0** | 1 | 1 | | | | | | | |
| $C_6$ | 5 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **0** | 1 | 1 | **1** | 0 | X | | | | |
| $C_6$ | -5 | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **0** | 1 | 1 | **0** | 1 | X | | | | |
| $C_6 \rightarrow C_5$ | | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **0** | 1 | 1 | **X** | | | | | (4 -7 -15) | (12) |
| | | | | | | | | | | | | | | | | | | | | (4 -7) | (13) (8+12) |
| $C_5 \rightarrow C_4$ | | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **0** | 0 | 1 | **X** | | | | | | | | (3 -8 14) | (14) |
| $C_4 \rightarrow C_3$ | | **1** | **1** | 0 | 0 | **1** | 1 | 0 | **X** | | | | | | | | | | | (-2 9 -13) | (15) |
| $C_3$ | 2 | **1** | **1** | 0 | 0 | **0** | 0 | 1 | | | | | | | | | | | | | |
| $C_4$ | 3 | **1** | **1** | 0 | 0 | **0** | 0 | 1 | **1** | 0 | 0 | | | | | | | | | | |
| $C_5$ | 4 | **1** | **1** | 0 | 0 | **0** | 0 | 1 | **1** | 0 | 0 | **1** | 1 | X | | | | | | | |
| $C_5$ | 4 | **1** | **1** | 0 | 0 | **0** | 0 | 1 | **1** | 0 | 0 | **0** | 0 | X | | | | | | | |
| $C_5 \rightarrow C_4$ | | **1** | **1** | 0 | 0 | **0** | 0 | 1 | **1** | 0 | 0 | **X** | | | | | | | | (-3 8 14) | (16) |
| | | | | | | | | | | | | | | | | | | | | (-3 8) | (17) (9+16) |
| $C_4$ | -3 | **1** | **1** | 0 | 0 | **0** | 0 | 1 | **0** | 1 | 1 | | | | | | | | | | |
| $C_5$ | 4 | **1** | **1** | 0 | 0 | **0** | 0 | 1 | **0** | 1 | 1 | **1** | 0 | X | | | | | | | |
| $C_5$ | 4 | **1** | **1** | 0 | 0 | **0** | 0 | 1 | **0** | 1 | 1 | **0** | 1 | X | | | | | | | |
| $C_5 \rightarrow C_4$ | | **1** | **1** | 0 | 0 | **0** | 0 | 1 | **0** | 1 | 1 | **X** | | | | | | | | (3 -8 -14) | (18) |
| | | | | | | | | | | | | | | | | | | | | (3 -8) | (19) (14+18) |
| $C_4 \rightarrow C_3$ | | **1** | **1** | 0 | 0 | **0** | 0 | 1 | **X** | | | | | | | | | | | (2 -9 13) | (20) |
| $C_3 \rightarrow C_2$ | | **1** | **1** | 0 | 0 | **X** | | | | | | | | | | | | | | (-1 10 12) | (21) |
| $C_2$ | -1 | **1** | **0** | 1 | 1 | | | | | | | | | | | | | | | | |
| $C_3$ | 2 | **1** | **0** | 1 | 1 | **1** | 0 | 0 | | | | | | | | | | | | | |
| $C_4$ | 3 | **1** | **0** | 1 | 1 | **1** | 0 | 0 | **1** | 1 | X | | | | | | | | | | |
| $C_4$ | -3 | **1** | **0** | 1 | 1 | **1** | 0 | 0 | **0** | 0 | X | | | | | | | | | | |
| $C_4 \rightarrow C_3$ | | **1** | **0** | 1 | 1 | **1** | 0 | 0 | **X** | | | | | | | | | | | (-2 9 13) | (22) |
| | | | | | | | | | | | | | | | | | | | | (-2 9) | (23) (15+21) |
| $C_3$ | -2 | **1** | **0** | 1 | 1 | **0** | 1 | 1 | | | | | | | | | | | | | |
| $C_4$ | 3 | **1** | **0** | 1 | 1 | **0** | 1 | 1 | **1** | 0 | X | | | | | | | | | | |
| $C_4$ | -3 | **1** | **0** | 1 | 1 | **0** | 1 | 1 | **0** | 1 | X | | | | | | | | | | |
| $C_4 \rightarrow C_3$ | | **1** | **0** | 1 | 1 | **0** | 1 | 1 | **X** | | | | | | | | | | | (2 -9 -13) | (24) |
| | | | | | | | | | | | | | | | | | | | | (2 -9) | (25) (20+23) |
| $C_3 \rightarrow C_2$ | | **1** | **0** | 1 | 1 | **X** | | | | | | | | | | | | | | (1 -10 -12) | (26) |
| $C_2 \rightarrow C_1$ | | **1** | **X** | | | | | | | | | | | | | | | | | (-11) | (27) |
| $C_1$ | -11 | **0** | | | | | | | | | | | | | | | | | | | |
| $C_2$ | 1 | **0** | **1** | 1 | 0 | | | | | | | | | | | | | | | | |
| $C_3$ | 2 | **0** | **1** | 1 | 0 | **1** | 1 | X | | | | | | | | | | | | | |
| $C_3$ | -2 | **0** | **1** | 1 | 0 | **0** | 0 | X | | | | | | | | | | | | | |
| $C_3 \rightarrow C_2$ | | **0** | **1** | 1 | 0 | **X** | | | | | | | | | | | | | | (-1 10 -12) | (28) |
| | | | | | | | | | | | | | | | | | | | | (-1 10) | (29) (20+27) |
| $C_2$ | -1 | **0** | **0** | 0 | 1 | | | | | | | | | | | | | | | | |
| $C_3$ | 2 | **0** | **0** | 0 | 1 | **1** | 0 | X | | | | | | | | | | | | | |
| $C_3$ | -2 | **0** | **0** | 0 | 1 | **0** | 1 | X | | | | | | | | | | | | | |
| $C_3 \rightarrow C_2$ | | **0** | **0** | 0 | 1 | **X** | | | | | | | | | | | | | | (1 -10 12) | (30) |
| | | | | | | | | | | | | | | | | | | | | (1 -10) | (31) (25+29) |
| $C_2 \rightarrow C_1$ | | **0** | **X** | | | | | | | | | | | | | | | | | (11) | (32) |
| **UNSAT** (27+32) | | | | | | | | | | | | | | | | | | | | | |

The striking result we see here, is that each cluster is solved in only a small number of steps. With a traditional lookahead solver, the problem will be solved in a number of steps which is exponential in the number of variables. But with the addition of CBCA, we only have to solve one more cluster. As can be seen, this adds only a constant number of steps, and therefore the problem can now be solved in a polynomial number of steps.

## 6. March_cluster architecture

The implementation of March_cluster is done by adjusting March_ks. In this section we describe the basic architecture used by March_cluster.

The architecture of March_cluster is based on the recursive, or lookahead, DPLL structure as described in section 2. There are two major changes to the original algorithm. The first one is the preselection of variables through CBP and the second one is the use of conflict clauses by means of CBCA. The locations of these changes are shown in algorithm 10.

The main solve function of the March_cluster implementation is a 2 step phase. The function starts with the creation of clusters, followed by the call of the first cluster_DPLL function. This is described in algorithm 9:

---

**Algorithm 9** March_cluster solve($\mathcal{F}$)

---

1: $createClusters(\mathcal{F})$
2: **return** $cluster_DPLL(\mathcal{F},\ 0)$

---

**Algorithm 10** $cluster_DPLL(\mathcal{F},\ oldCluster)$

---

1: $\mathcal{F} \leftarrow IterativeUnitPropagation(\mathcal{F})$
2: **if** $\mathcal{F} = \emptyset$ **then**
3: **return** SAT
4: **else if** $\mathcal{F}$ *contains empty clause* **then**
5: **return** UNSAT
6: **end if**
7: $currentCluster \leftarrow DetermineCurrentCluster()$
8: variable set $\mathcal{P} \leftarrow Preselect(currentCluster)$
9: $x \leftarrow GetBranchVariable(\mathcal{P})$
10: **if** $cluster_DPLL(\mathcal{F} \cup \{x\} = SAT)$ **then**
11: **return** SAT
12: **else if** $checkConsistency() = UNSAT$ **then**
13: **return** UNSAT
14: **end if**
15: **if** $cluster_DPLL(\mathcal{F} \cup \{\neg x\} = SAT)$ **then**
16: **return** SAT
17: **end if**
18: **while** $currentCluster > oldCluster$ **do**
19: $currentCluster \leftarrow currentCluster - 1$
20: $addConflictClause(currentCluster)$
21: **end while**
22: **return** UNSAT

---

## 7. Evaluation

To evaluate the effectiveness of CBP and CBCA, we selected several benchmarks for which we expected the performance of March_cluster to be better than that of March_ks. As mentioned before the aim of March_cluster is to improve the performance of lookahead solvers on problems with large diameters. Therefore the chosen benchmarks were selected based primarily on their diameter. We also selected some benchmarks based on the performance of other solvers and based on the characteristics of certain families of problems.

To measure the improvements of the discussed heuristics, we tested each benchmark with five different versions of March_ks.

### 7.1 Test Setup

The five versions that were used during the benchmarking phase are chosen as follows. The first version is the reference version for the adapted versions. It is the original version of March_ks, but without the heuristics equivalence reasoning and distribution branching, because they violated some of the assumptions of our new heuristics. The other four versions are different combinations of the new heuristics. The CBP heuristic is either based on forward or reversed clustering, but is always turned on. CBCA can be turned on or off. A summary of the versions can be found in Table 7.1.

| Version | Description |
|---|---|
| $v_n$ | The original version of March_ks. |
| $v_{f-}$ | March_cluster using forward clustering based CBP and no CBCA. |
| $v_{b-}$ | March_cluster using reversed clustering based CBP and no CBCA. |
| $v_{f+}$ | March_cluster using forward clustering based CBP and CBCA. |
| $v_{b+}$ | March_cluster using reversed clustering based CBP and CBCA. |

The system on which the tests were conducted was a Intel Core 2 Duo E6400 (2.13 GHz) with 2GB memory[4.]. As the solver is purely single threaded, only one core of the processor was used.

### 7.2 Benchmarks

The tested benchmarks were selected from the union of the SATLib [3], SAT competition 2005 and SAT competition 2007 benchmarks. Because the largest improvement gained by CBP and CBCA is expected to occur on problems with a large diameter, we selected the largest 10% of the benchmarks based on diameter, these problems have a diameter of 16 or higher. We also chose the ferry and hanoi benchmarks. A complete list of all the 179 selected benchmarks can be found in appendix A. Due to time constraints, the statistics presented below do not include data from the dubois, pret and xor benchmarks.

4. Other components of the test setup included a Asus P5B motherboard and a Maxtor 250GB SATA2 hard disk.

### 7.3 Results

The number of solved benchmarks for each version can be found in table 3. From these results, we selected only benchmarks that gave sensible results for further analysis. We left out benchmarks that were not solved by any of the adapted versions within 1200 seconds, problems that were solved under 1 second by all versions and different configurations of the same problems that gave no additional insight in the differences between versions. The list of the times needed to solve these benchmarks can be found in table 4. For convenience the diameter is also mentioned in that table.

**Table 3.** Percentages of solved benchmarks within the time limit of 1200 seconds.

| Version | Benchmarks solved |
|---|---|
| $v_n$ | 36% |
| $v_{f-}$ | 19% |
| $v_{b-}$ | 16% |
| $v_{f+}$ | 40% |
| $v_{b+}$ | 36% |

To analyze the theoretic speedup caused by clustering and conflict clauses, we need a more abstract method of measuring the performance. This is done with the *node-count*. The node count indicates how many branches of the search space were visited [2]. The expectation is that the node count will drop when conflict clauses are used, because that heuristic will prevent that identical branches in the search space will be investigated multiple times. For measuring the impact of conflict clauses on the computation, we recorded the following statistics: how many conflict clauses were added initially and added and removed during solving, the average size of the conflict clauses as a percentage of the size of the clusters and how many backjumps were made. These variables will show how well the heuristic performs. In table 5 these statistics are listed for version $v_{f+}$.

The columns *initial*, *added* and *removed* indicate statistics from the conflict clauses. All clauses that are fully contained within a cluster are added as conflict clauses before solving starts, these are listed in *initial*. All conflict clauses that are added during solving are listed under *added*. All conflict clauses that are removed due to minimization are listed under *removed*. The average size of conflict clauses, as a percentage of the clustersize, is listed under *size (%)*. The number of times the solver backjumped due to CBCA is listed under $\curvearrowleft$.

**Table 4.** Benchmarks with a diameter of 16 or higher. All results are in seconds.
∅= diameter - = timeout (1200 seconds)

| benchmark name | ∅ | $v_n$ | $v_{f-}$ | $v_{b-}$ | $v_{f+}$ | $v_{b+}$ |
|---|---|---|---|---|---|---|
| SAT instances | | | | | | |
| depots3_ks99i.renamed-as | 23 | 0.578 | - | - | 10.437 | - |
| driverlog2_ks99i.renamed | 18 | 0.343 | - | - | - | - |
| driverlog5_ks99i.renamed | 16 | 0.562 | - | - | - | - |
| ferry10_ks99a.renamed | 39 | - | - | - | 910.312 | - |
| ferry5_ks99i.renamed | 37 | 8.093 | - | - | 0.671 | 0.937 |
| ferry5_v01i.renamed | 19 | 25.796 | 91.828 | - | 0.406 | 2.250 |
| ferry6_ks99a.renamed | 23 | 2.062 | 9.687 | - | 0.453 | 1.796 |
| ferry6_ks99i.renamed | 45 | 527.671 | - | - | 3.046 | 4.875 |
| ferry6_v01a.renamed | 12 | 34.140 | 6.281 | 329.421 | 0.562 | 3.921 |
| ferry6_v01i.renamed | 23 | - | - | - | 2.187 | 16.156 |
| ferry7_ks99a.renamed | 27 | 16.359 | 511.343 | - | 2.203 | 12.265 |
| ferry7_ks99i.renamed | 53 | - | - | - | 18.093 | 29.421 |
| ferry7_v01a.renamed | 14 | - | 376.515 | - | 3.843 | 25.359 |
| ferry7_v01i.renamed | 27 | - | - | - | 12.000 | 104.453 |
| ferry8_ks99a.renamed | 31 | 463.750 | - | - | 15.687 | 112.296 |
| ferry8_ks99i.renamed | 61 | - | - | - | 101.437 | 207.484 |
| ferry8_v01a.renamed | 16 | - | - | - | 27.828 | 158.953 |
| ferry8_v01i.renamed | 31 | - | - | - | 67.656 | 659.359 |
| ferry9_ks99a.renamed | 35 | - | - | - | 123.500 | - |
| ferry9_v01a.renamed | 18 | - | - | - | 210.796 | 995.140 |
| rovers5_ks99i.renamed | 17 | 0.718 | - | - | - | - |
| satellite2_ks99i.renamed | 23 | 0.562 | - | - | 10.640 | - |
| strips-gripper-08t15.shuffled | 28 | - | - | - | 584.703 | 265.109 |
| bw_large.c | 16 | 1.500 | - | - | - | - |
| bw_large.d | 20 | 67.640 | - | - | - | - |
| hanoi4 | 17 | 6.359 | 11.421 | - | 218 | - |
| hanoi5 | 33 | - | - | - | 3.421 | - |
| hanoi6 | 65 | - | - | - | 68.203 | - |
| logistics.d | 17 | 2.906 | - | - | - | - |
| par16-1 | 26 | 0.906 | 2.062 | 4.906 | 2.218 | 8.531 |
| par16-2 | 25 | 1.437 | 0.750 | 3.546 | 0.828 | 3.828 |
| par16-4 | 26 | 0.796 | 0.750 | 23.421 | 0.796 | 1.084.953 |
| par16-5 | 25 | 0.671 | 0.109 | 19.140 | 0.109 | 505.187 |
| UNSAT instances | | | | | | |
| strips-gripper-08t14.shuffled | 26 | - | - | - | 985.343 | 218.953 |
| emptyroom-4-h21-unsat | 46 | 896.250 | - | - | - | - |
| dubois6 | 7 | 0.015 | 0.062 | 0.062 | 0.046 | 0.093 |
| dubois20 | 21 | 4.281 | 2.578 | 2.468 | 0.015 | 0.015 |
| dubois21 | 22 | 3.484 | 5.187 | 5.062 | 0.015 | 0.031 |
| dubois22 | 23 | 18.546 | 10.125 | 9.953 | 0 | 0 |
| dubois23 | 24 | 41.578 | 20.609 | 19.531 | 0 | 0 |
| dubois24 | 25 | 56.328 | 41.093 | 39.406 | 0 | 0 |
| dubois25 | 26 | 78.015 | 82.968 | 79.375 | 0 | 0 |
| dubois26 | 27 | 77.671 | 165.625 | 159.875 | 0.015 | 0 |
| dubois27 | 28 | 645.875 | 326.484 | 323.875 | 0.015 | 0.015 |
| dubois28 | 29 | 331.703 | 663.953 | 651.109 | 0 | 0 |
| dubois29 | 30 | 1199.031 | - | - | 0 | 0 |
| dubois30 | 31 | - | - | - | 0 | 0 |
| dubois50 | 51 | - | - | - | 0 | 0 |
| dubois100 | 101 | - | - | - | 31 | 31 |
| pret150_25 | 16 | - | - | - | - | 581.468 |
| pret150_40 | 16 | - | - | - | - | 582.203 |
| pret150_60 | 16 | - | - | - | - | 582.968 |
| pret150_75 | 16 | - | - | - | - | 586.953 |
| x1_16.shuffled | 7 | 359 | 93 | 62 | 140 | 46 |
| x1_24.shuffled | 7 | 163.656 | 51.515 | 56.031 | - | - |
| x2_16.shuffled | 7 | 0.468 | 0.156 | 0.171 | 0.171 | 0.546 |
| x2_48.shuffled | 10 | - | - | - | - | 0 |

To get more insight in the number of conflict clauses that are removed due to minimization, table 6 lists percentages of how many conflict clauses were removed. Per interval of percentage removed conflict clauses, the table shows in how many benchmarks that amount of conflict clauses were removed.

**Table 5.** Detailed results of the benchmarks for version $v_{f+}$.
∅= diameter ↶ = backjumps

| benchmark name | ∅ | nodes $v_n$ | nodes $v_{f+}$ | initial | added | removed | size (%) | ↶ |
|---|---|---|---|---|---|---|---|---|
| SAT instances | | | | | | | | |
| depots3_ks99i.renamed | 23 | 21 | 87040 | 25344 | 7640 | 2116 | 19.75 | 0 |
| driverlog2_ks99i.renamed | 18 | 12 | 1464020 | 11492 | 131003 | 26561 | 69.04 | 0 |
| driverlog5_ks99i.renamed | 16 | 41 | 1903586 | 17262 | 169651 | 31376 | 73.39 | 3 |
| ferry10_ks99a.renamed | 39 | 1877164 | 589116 | 24961 | 169719 | 1774 | 80.87 | 0 |
| ferry5_ks99i.renamed | 37 | 10056 | 8406 | 6479 | 3021 | 371 | 29.53 | 277 |
| ferry5_v01i.renamed | 19 | 56447 | 4157 | 7207 | 1489 | 0 | 21.10 | 0 |
| ferry6_ks99a.renamed | 23 | 3015 | 7935 | 5249 | 1863 | 40 | 25.34 | 0 |
| ferry6_ks99i.renamed | 45 | 713821 | 27853 | 11217 | 9386 | 921 | 40.74 | 744 |
| ferry6_v01a.renamed | 12 | 60611 | 8360 | 5581 | 1590 | 0 | 25.05 | 0 |
| ferry6_v01i.renamed | 23 | 3948889 | 18710 | 12313 | 6205 | 0 | 35.67 | 0 |
| ferry7_ks99a.renamed | 27 | 36534 | 27009 | 8437 | 6745 | 80 | 40.08 | 0 |
| ferry7_ks99i.renamed | 53 | 4204702 | 79281 | 17801 | 27826 | 2916 | 54.23 | 2500 |
| ferry7_v01a.renamed | 14 | 2474061 | 42057 | 8901 | 7047 | 0 | 44.85 | 0 |
| ferry7_v01i.renamed | 27 | 2462051 | 69413 | 19339 | 21318 | 0 | 52.92 | 0 |
| ferry8_ks99a.renamed | 31 | 920788 | 81111 | 12685 | 21431 | 232 | 56.31 | 0 |
| ferry8_ks99i.renamed | 61 | 2997697 | 221587 | 26531 | 75310 | 7249 | 67.06 | 6070 |
| ferry8_v01a.renamed | 16 | 6187553 | 175761 | 13303 | 25552 | 0 | 64.37 | 0 |
| ferry8_v01i.renamed | 31 | 2895612 | 223611 | 28585 | 65152 | 0 | 68.68 | 0 |
| ferry9_ks99a.renamed | 35 | 2492076 | 224798 | 18143 | 62242 | 714 | 70.56 | 0 |
| ferry9_v01a.renamed | 18 | 2722668 | 592791 | 18937 | 81414 | 0 | 78.36 | 0 |
| rovers5_ks99i.renamed | 17 | 61 | 7034258 | 20274 | 135304 | 77638 | 42.20 | 0 |
| satellite2_ks99i.renamed | 23 | 21 | 87040 | 25344 | 7640 | 2116 | 19.75 | 0 |
| strips-gripper-08t15.shuffled | 28 | 9316158 | 814159 | 17124 | 140801 | 3795 | 82.74 | 534 |
| bw_large.c | 16 | 7 | 981218 | 26451 | 93467 | 16558 | 63.40 | 96 |
| bw_large.d | 20 | 341 | 593295 | 66969 | 45605 | 2256 | 32.48 | 67 |
| hanoi4 | 17 | 29924 | 2493 | 990 | 888 | 1 | 47.35 | 0 |
| hanoi5 | 33 | 5751590 | 28356 | 3008 | 12117 | 1 | 73.60 | 0 |
| hanoi6 | 65 | 3155028 | 165354 | 10006 | 92458 | 212 | 81.47 | 3 |
| logistics.d | 17 | 107 | 2389991 | 10343 | 55244 | 48 | 57.51 | 0 |
| par16-1 | 26 | 5134 | 39053 | 151 | 2773 | 4 | 94.68 | 0 |
| par16-2 | 25 | 8069 | 14700 | 369 | 1673 | 0 | 80.31 | 0 |
| par16-4 | 26 | 7443 | 15651 | 235 | 552 | 0 | 70.78 | 0 |
| par16-5 | 25 | 4531 | 660 | 210 | 173 | 0 | 56.53 | 0 |
| UNSAT instances | | | | | | | | |
| emptyroom-4-h21-unsat | 46 | 1 | 708591 | 52409 | 0 | 0 | 0.08 | 0 |
| strips-gripper-08t14.shuffled | 26 | 17128663 | 970716 | 15120 | 183279 | 4562 | 86.34 | 802 |
| dubois6 | 7 | 63 | 21 | 0 | 12 | 3 | 3 | 0 |
| dubois20 | 21 | 360198 | 77 | 0 | 68 | 3 | 3 | 0 |
| dubois21 | 22 | 336932 | 81 | 0 | 72 | 3 | 3 | 0 |
| dubois22 | 23 | 1286227 | 85 | 0 | 76 | 3 | 3 | 0 |
| dubois23 | 24 | 3641284 | 89 | 0 | 80 | 3 | 3 | 0 |
| dubois24 | 25 | 5174198 | 93 | 0 | 84 | 3 | 3 | 0 |
| dubois25 | 26 | 6301452 | 97 | 0 | 88 | 3 | 3 | 0 |
| dubois26 | 27 | 6519577 | 101 | 0 | 92 | 3 | 3 | 0 |
| dubois27 | 28 | 49563771 | 105 | 0 | 96 | 3 | 3 | 0 |
| dubois28 | 29 | 28260439 | 109 | 0 | 100 | 3 | 3 | 0 |
| dubois29 | 30 | 112160669 | 113 | 0 | 104 | 3 | 3 | 0 |
| dubois30 | 31 | 94537917 | 117 | 0 | 108 | 3 | 3 | 0 |
| dubois50 | 51 | 83552370 | 197 | 0 | 188 | 3 | 3 | 0 |
| dubois100 | 101 | 46511128 | 397 | 0 | 388 | 3 | 3 | 0 |
| x1_16.shuffled | 7 | 43771 | 4287 | 4 | 2180 | 259 | 259 | 0 |
| x1_44.shuffled | 7 | 57400595 | 151 | 12 | 14 | 3 | 7 | 39 |
| x2_16.shuffled | 7 | 36863 | 16447 | 0 | 2132 | 3 | 3 | 0 |

# 8. Conclusion and Future Work

## 8.1 Conclusions

Based on the results in section 7, we have reached the following conclusions, each of which will be explained in the following paragraphs.

1. March_cluster (using forward CBP and CBCA) solves slightly more benchmarks than March_ks.

2. The effectiveness of CBP is greatly increased when CBCA is applied.

3. Benchmarks that are solved by at least one solver of March_ks and March_cluster are solved faster by March_cluster.

**Table 6.** Percentage of benchmarks with removed conflict clauses.

| % of removed conflict clauses | $v_{f+}$ | $v_{b+}$ |
|---|---|---|
| 0 - 10 | 46.27% | 32.84% |
| 10 - 20 | 4.48% | 4.48% |
| 20 - 30 | 3.73% | 2.24% |
| 30 - 40 | 0.75% | 2.24% |
| 40 - 50 | 0% | 3.73% |
| 50 - 60 | 14.93% | 4.48% |
| 60 - 70 | 0% | 3.73% |
| 70 - 80 | 0.75% | 2.99% |
| 80 - 90 | 0.75% | 0.75% |
| 90 - 100 | 0.75% | 1.49% |

4. The choice of the clustering technique is important, since there is a significant difference in running time between the different clustering techniques.

5. CBCA creates on average only 10% extra clauses that can be removed due to minimization.

6. Backjumping does not occur often in both $v_{f+}$ and $v_{b+}$ and when it does it does not significantly shorten the search space.

Conclusions 1 and 2 follow directly from table 3. Although March_cluster has better results, some benchmarks remain where it is slower. When looking at table 5, we see that in all these benchmarks, the node-count of March_cluster increases significantly. This shows that the slowdown is caused by CBP, because CBCA will never be responsible for increasing the node-count.

A comparison between the results of version $v_n$ and $v_{f+}$ led us to conclusion 3. When all results are compared in which at least version $v_n$ or $v_{f+}$ solved the benchmark in time, it turns out that 71% is solved faster by $v_{f+}$ and only 18% is solved by $v_n$ and not by $v_{f+}$.

Conclusion 4 can be seen from the time difference between the forward and reversed CBP versions, and because different configurations of the same benchmark are sometimes solved fastest by forward CBP, and sometimes solved fastest by reversed CBP. This can be explained by the way in which problems are constructed. When a problem is translated to a satisfiability problem, the CNF exhibits a clear structure. In most cases this structure is best used with forward clustering. When the CNF is shuffled, the vig may become mirrored and reversed clustering becomes the best technique.

Conclusion 5 follows from table 6. As can be seen, in both version $v_{f+}$ and $v_{b+}$ most benchmarks are in the first interval, where less than 10% of the added conflict clauses is removed.

The conclusion from 6 requires further research, since we do not know whether it is a good or a bad result. Although the search space is not significantly reduced by this technique, it might indicate that conflicts in CBP occur soon after the responsible variables

are set, thus reducing wasted search space. However it might also indicate that our conflict clauses contain redundant literals, that should have been removed by minimization.

### 8.2 Future Work

In our research we have come along many issues in the CBP and CBCA heuristics that can be improved. Here we will discuss the most important of these issues.

First of all, further research has to be done on the effectiveness of conflict clauses. In this paper we have focussed on the number of removed conflict clauses during minimization, while a better benchmark would be the number of times a conflict clause became unit.

Secondly we have always used the clustering algorithm as stated in algorithm 4. Other techniques are possible, for example by making a distinction between different kinds of variables, based on there significance for the problem.

A more structural problem in March_cluster is the occurrence of useless assignments. When a cluster is not yet fully assigned, but all clauses in the cluster are already satisfied, March_cluster will still try to assign the unassigned variables, although they will no longer have any effect. The next time we backtrack to this cluster, these redundant variables are the first to be reassigned in an attempt to satisfy $\mathcal{F}$.

Another problem is that an assignment in, say, the first cluster may lead to an assignment in, say, the last clusters due to unit propagation. When this particular assignment will eventually cause a conflict in the last cluster, we will backtrack to the last but one cluster. A new assignment is tried and when we come the last cluster again, the conflict will be detected again. The solver actually had to backtrack to the first place where the conflict originated, instead of to the previous cluster.

## References

[1] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[2] M.J.H. Heule. March, towards a lookahead sat solver for general purposes. Master's thesis, Delft University of Technology, 2004.

[3] Holger H. Hoos and Thomas Stützle. Satlib: An online resource for research on sat. In *SAT 2000*, pages 283–292. IOS Press, 2000.

[4] Sharad Malik Lintao Zhang. The quest for efficient boolean satisfiability solvers. In *CADE: International Conference on Automated Deduction*, 2002.

[5] Niklas Srensson Niklas Een. Minisat a sat solver with conflict-clause minimization. 2005.

[6] Decomposing Satisfiability Problems. P.r. herwig. Master's thesis, Delft University of Technology, 2006.

[7] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

## Appendix A. Used benchmarks

Table 7: List of used benchmarks

| Portfolio | Benchmark name |
|---|---|
| SATCompetition2002 | x1.shuffled.cnf |
| SATCompetition2002 | x1_128.shuffled.cnf |
| SATCompetition2002 | x1_16.shuffled.cnf |
| SATCompetition2002 | x1_24.shuffled.cnf |
| SATCompetition2002 | x1_32.shuffled.cnf |
| SATCompetition2002 | x1_36.shuffled.cnf |
| SATCompetition2002 | x1_40.shuffled.cnf |
| SATCompetition2002 | x1_44.shuffled.cnf |
| SATCompetition2002 | x1_48.shuffled.cnf |
| SATCompetition2002 | x1_56.shuffled.cnf |
| SATCompetition2002 | x1_64.shuffled.cnf |
| SATCompetition2002 | x1_72.shuffled.cnf |
| SATCompetition2002 | x1_80.shuffled.cnf |
| SATCompetition2002 | x1_96.shuffled.cnf |
| SATCompetition2002 | x2_128.shuffled.cnf |
| SATCompetition2002 | x2_16.shuffled.cnf |
| SATCompetition2002 | x2_24.shuffled.cnf |
| SATCompetition2002 | x2_32.shuffled.cnf |
| SATCompetition2002 | x2_36.shuffled.cnf |
| SATCompetition2002 | x2_40.shuffled.cnf |
| SATCompetition2002 | x2_44.shuffled.cnf |
| SATCompetition2002 | x2_48.shuffled.cnf |
| SATCompetition2002 | x2_56.shuffled.cnf |
| SATCompetition2002 | x2_64.shuffled.cnf |
| SATCompetition2002 | x2_72.shuffled.cnf |
| SATCompetition2002 | x2_80.shuffled.cnf |
| SATCompetition2002 | x2_96.shuffled.cnf |
| SATCompetition2005 | driverlog1_ks99i.shuffled-as.sat05-4018.cnf |
| SATCompetition2005 | driverlog1_ks99i.renamed-as.sat05-3951.cnf |
| SATCompetition2005 | satellite2_ks99i.renamed-as.sat05-3983.cnf |
| SATCompetition2005 | depots3_ks99i.renamed-as.sat05-3945.cnf |
| SATCompetition2005 | depots3_ks99i.shuffled-as.sat05-4012.cnf |
| SATCompetition2005 | driverlog2_ks99i.renamed-as.sat05-3952.cnf |
| SATCompetition2005 | driverlog2_ks99i.shuffled-as.sat05-4019.cnf |
| SATCompetition2005 | driverlog5_ks99i.renamed-as.sat05-3955.cnf |
| SATCompetition2005 | driverlog5_ks99i.shuffled-as.sat05-4022.cnf |
| SATCompetition2005 | grid-pbl-0060.shuffled-as.sat05-1333.shuffled-as.sat05-1333.cnf |
| SATCompetition2005 | grid-pbl-0060.shuffled-as.sat05-1342.shuffled-as.sat05-1342.cnf |
| SATCompetition2005 | grid-pbl-0070.shuffled-as.sat05-1334.shuffled-as.sat05-1334.cnf |

| Portfolio | Benchmark name |
|---|---|
| SATCompetition2005 | grid-pbl-0070.shuffled-as.sat05-1343.shuffled-as.sat05-1343.cnf |
| SATCompetition2005 | grid-pbl-0080.shuffled-as.sat05-1335.shuffled-as.sat05-1335.cnf |
| SATCompetition2005 | grid-pbl-0080.shuffled-as.sat05-1344.shuffled-as.sat05-1344.cnf |
| SATCompetition2005 | grid-pbl-0090.shuffled-as.sat05-1336.shuffled-as.sat05-1336.cnf |
| SATCompetition2005 | grid-pbl-0090.shuffled-as.sat05-1345.shuffled-as.sat05-1345.cnf |
| SATCompetition2005 | grid-pbl-0100.shuffled-as.sat05-1337.shuffled-as.sat05-1337.cnf |
| SATCompetition2005 | grid-pbl-0100.shuffled-as.sat05-1346.shuffled-as.sat05-1346.cnf |
| SATCompetition2005 | grid-pbl-0150.shuffled-as.sat05-1338.shuffled-as.sat05-1338.cnf |
| SATCompetition2005 | grid-pbl-0150.shuffled-as.sat05-1347.shuffled-as.sat05-1347.cnf |
| SATCompetition2005 | rovers5_ks99i.renamed-as.sat05-3974.cnf |
| SATCompetition2005 | rovers5_ks99i.shuffled-as.sat05-4041.cnf |
| SATCompetition2005 | satellite2_ks99i.shuffled-as.sat05-4050.cnf |
| SATCompetition2005 | strips-gripper-08t14.shuffled-as.sat05-1156.cnf |
| SATCompetition2005 | strips-gripper-08t15.shuffled-as.sat05-1149.cnf |
| SATCompetition2005 | strips-gripper-10t18.shuffled-as.sat05-1150.cnf |
| SATCompetition2005 | strips-gripper-10t19.shuffled-as.sat05-1143.cnf |
| SATCompetition2005 | strips-gripper-12t23.shuffled-as.sat05-1144.cnf |
| SATCompetition2005 | ferry10_ks99a.renamed-as.sat05-3992.cnf |
| SATCompetition2005 | ferry10_ks99a.shuffled-as.sat05-4059.cnf |
| SATCompetition2005 | ferry10_v01a.renamed-as.sat05-3993.cnf |
| SATCompetition2005 | ferry10_v01a.shuffled-as.sat05-4060.cnf |
| SATCompetition2005 | ferry5_ks99i.renamed-as.sat05-3994.cnf |
| SATCompetition2005 | ferry5_ks99i.shuffled-as.sat05-4061.cnf |
| SATCompetition2005 | ferry5_v01i.renamed-as.sat05-3995.cnf |
| SATCompetition2005 | ferry5_v01i.shuffled-as.sat05-4062.cnf |
| SATCompetition2005 | ferry6_ks99a.renamed-as.sat05-3996.cnf |
| SATCompetition2005 | ferry6_ks99a.shuffled-as.sat05-4063.cnf |
| SATCompetition2005 | ferry6_ks99i.renamed-as.sat05-3997.cnf |
| SATCompetition2005 | ferry6_ks99i.shuffled-as.sat05-4064.cnf |
| SATCompetition2005 | ferry6_v01a.renamed-as.sat05-3998.cnf |
| SATCompetition2005 | ferry6_v01a.shuffled-as.sat05-4065.cnf |
| SATCompetition2005 | ferry6_v01i.renamed-as.sat05-3999.cnf |
| SATCompetition2005 | ferry6_v01i.shuffled-as.sat05-4066.cnf |
| SATCompetition2005 | ferry7_ks99a.renamed-as.sat05-4000.cnf |
| SATCompetition2005 | ferry7_ks99a.shuffled-as.sat05-4067.cnf |
| SATCompetition2005 | ferry7_ks99i.renamed-as.sat05-4001.cnf |
| SATCompetition2005 | ferry7_ks99i.shuffled-as.sat05-4068.cnf |
| SATCompetition2005 | ferry7_v01a.renamed-as.sat05-4002.cnf |
| SATCompetition2005 | ferry7_v01a.shuffled-as.sat05-4069.cnf |
| SATCompetition2005 | ferry7_v01i.renamed-as.sat05-4003.cnf |
| SATCompetition2005 | ferry7_v01i.shuffled-as.sat05-4070.cnf |
| SATCompetition2005 | ferry8_ks99a.renamed-as.sat05-4004.cnf |
| SATCompetition2005 | ferry8_ks99a.shuffled-as.sat05-4071.cnf |
| SATCompetition2005 | ferry8_ks99i.renamed-as.sat05-4005.cnf |

| Portfolio | Benchmark name |
|---|---|
| SATCompetition2005 | ferry8_ks99i.shuffled-as.sat05-4072.cnf |
| SATCompetition2005 | ferry8_v01a.renamed-as.sat05-4006.cnf |
| SATCompetition2005 | erry8_v01a.shuffled-as.sat05-4073.cnf |
| SATCompetition2005 | ferry8_v01i.renamed-as.sat05-4007.cnf |
| SATCompetition2005 | ferry8_v01i.shuffled-as.sat05-4074.cnf |
| SATCompetition2005 | ferry9_ks99a.renamed-as.sat05-4008.cnf |
| SATCompetition2005 | ferry9_ks99a.shuffled-as.sat05-4075.cnf |
| SATCompetition2005 | ferry9_v01a.renamed-as.sat05-4009.cnf |
| SATCompetition2005 | ferry9_v01a.shuffled-as.sat05-4076.cnf |
| SATLib | bf1355-075.cnf |
| SATLib | bf1355-638.cnf |
| SATLib | bf2670-001.cnf |
| SATLib | bw_large.c.cnf |
| SATLib | bw_large.d.cnf |
| SATLib | dubois6.cnf |
| SATLib | dubois20.cnf |
| SATLib | dubois21.cnf |
| SATLib | dubois22.cnf |
| SATLib | dubois23.cnf |
| SATLib | dubois24.cnf |
| SATLib | dubois25.cnf |
| SATLib | dubois26.cnf |
| SATLib | dubois27.cnf |
| SATLib | dubois28.cnf |
| SATLib | dubois29.cnf |
| SATLib | dubois30.cnf |
| SATLib | dubois50.cnf |
| SATLib | dubois100.cnf |
| SATLib | hanoi4.cnf |
| SATLib | hanoi5.cnf |
| SATLib | hanoi6.cnf |
| SATLib | logistics.d.cnf |
| SATLib | par16-1.cnf |
| SATLib | par16-2.cnf |
| SATLib | par16-3.cnf |
| SATLib | par16-4.cnf |
| SATLib | par16-5.cnf |
| SATLib | par32-1.cnf |
| SATLib | par32-2.cnf |
| SATLib | par32-3.cnf |
| SATLib | par32-4.cnf |
| SATLib | par32-5.cnf |
| SATLib | par8-1.cnf |
| SATLib | par8-2.cnf |

| Portfolio | Benchmark name |
|---|---|
| SATLib | par8-3.cnf |
| SATLib | par8-4.cnf |
| SATLib | par8-5.cnf |
| SATLib | pret150_25.cnf |
| SATLib | pret150_40.cnf |
| SATLib | pret150_60.cnf |
| SATLib | pret150_75.cnf |
| SATCompetition2007 | 999999000001nw.sat05-447.reshuffled-07.cnf |
| SATCompetition2007 | AProVE07-26.cnf |
| SATCompetition2007 | clauses-2.cnf |
| SATCompetition2007 | dspam_dump_vc1080.cnf |
| SATCompetition2007 | dspam_dump_vc1081.cnf |
| SATCompetition2007 | dspam_dump_vc1093.cnf |
| SATCompetition2007 | dspam_dump_vc949.cnf |
| SATCompetition2007 | dspam_dump_vc950.cnf |
| SATCompetition2007 | dspam_dump_vc962.cnf |
| SATCompetition2007 | emptyroom-4-h21-unsat.cnf |
| SATCompetition2007 | emptyroom-4-h22-sat.cnf |
| SATCompetition2007 | equilarge_l2.sat05-519.reshuffled-07.cnf |
| SATCompetition2007 | equilarge_l3.sat05-520.reshuffled-07.cnf |
| SATCompetition2007 | equilarge_l4.sat05-521.reshuffled-07.cnf |
| SATCompetition2007 | equilarge_l5.sat05-522.reshuffled-07.cnf |
| SATCompetition2007 | grid-pebbling-sat-grid-pbl-0300.sat05-1341.sat05-1341.reshuffled-07.cnf |
| SATCompetition2007 | grid-pebbling-unsat-grid-pbl-0300.sat05-1350.sat05-1350.reshuffled-07.cnf |
| SATCompetition2007 | hsat_vc11935.cnf |
| SATCompetition2007 | hsat_vc11944.cnf |
| SATCompetition2007 | hsat_vc12016.cnf |
| SATCompetition2007 | hsat_vc12062.cnf |
| SATCompetition2007 | hsat_vc12070.cnf |
| SATCompetition2007 | itox_vc1033.cnf |
| SATCompetition2007 | itox_vc1044.cnf |
| SATCompetition2007 | itox_vc1130.cnf |
| SATCompetition2007 | itox_vc1138.cnf |
| SATCompetition2007 | itox_vc1216.cnf |
| SATCompetition2007 | itox_vc909.cnf |
| SATCompetition2007 | itox_vc965.cnf |
| SATCompetition2007 | itox_vc979.cnf |
| SATCompetition2007 | safe-30-h29-unsat.cnf |
| SATCompetition2007 | sat-grid-pbl-0070.sat05-1334.reshuffled-07.cnf |
| SATCompetition2007 | sat-grid-pbl-0200.sat05-1339.reshuffled-07.cnf |
| SATCompetition2007 | sat-strips-gripper-10t19.sat05-1143.reshuffled-07.cnf |
| SATCompetition2007 | sat-strips-gripper-12t23.sat05-1144.reshuffled-07.cnf |
| SATCompetition2007 | sgp_5-5-6.sat05-2675.reshuffled-07.cnf |
| SATCompetition2007 | sortnet-6-ipc5-h11-unsat.cnf |

| Portfolio | Benchmark name |
|---|---|
| SATCompetition2007 | sortnet-7-ipc5-h15-unsat.cnf |
| SATCompetition2007 | strips-gripper-12t22.sat05-1151.reshuffled-07.cnf |
| SATCompetition2007 | strips-gripper-18t35.sat05-1147.reshuffled-07.cnf |
| SATCompetition2007 | unsat-grid-pbl-0080.sat05-1344.reshuffled-07.cnf |
| SATCompetition2007 | uts-l05-ipc5-h26-unsat.cnf |
| SATCompetition2007 | uts-l05-ipc5-h27-unknown.cnf |
| SATCompetition2007 | xinetd_vc56687.cnf |
| SATCompetition2007 | xinetd_vc56703.cnf |