Fast Blocked Clause Decomposition with High Quality

Jingchao Chen

School of Informatics, Donghua University
2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China
chen-jc@dhu.edu.cn

Abstract. Any CNF formula can be decomposed two blocked subsets such that both can be solved by BCE (Blocked Clause Elimination). To make the decomposition more useful, one hopes to have the decomposition as unbalanced as possible. It is often time consuming to achieve this goal, since it has been proven to be NP-hard. So far there have been several decomposition and post-processing algorithms such as PureDecompose, QuickDecompose, EagerMover etc. We found that these existing algorithms are often either inefficient or low-quality decomposition. To build an efficient and high-quality algorithm, we improve the existing BCE, and present two new variants of PureDecompose, a new heuristic decomposition called LessInterfereDecompose, and a new post-processing algorithm called RsetGuidedDecompose. Combining these new techniques results in a new algorithm called MixDecompose. In our experiments, there is no application formula where the quality of PureDecompose+EagerMover is better than MixDecompose. In terms of speed, MixDecompose is also very fast. Our average runtime is a little longer, but the worst-case runtime is shorter. In theory, our two variants of PureDecompose requires linear time in the number of clauses. By limiting the size of the touch list used by BCE, we can guarantee always that MixDecompose runs in linear time.

Keywords: Blocked Clause Elimination, Blocked Clause Decomposition, CNF preprocessing

1 Introduction

Recently, one found that blocked clause decomposition can not only efficiently find backbone variables [1] and implied binary equivalences through SAT sweeping, but also improve the performance of the state-of-the-art SAT solvers such as Lingeling [2] on hard application benchmarks [3,4]. Due to its importance, many researchers have been attracted to pay attention to this subject.

A set of clauses is said to a blocked set if it can be removed it completely by Blocked Clause Elimination (BCE) [5]. Any CNF formula can be decomposed into two blocked subsets. To make a blocked clause decomposition more useful, one wants always to have two blocked subsets as unbalanced as possible. The problem is that it is not easy to find the most unbalanced subsets. In theory,
one has proven that finding a maximal blocked subset of a CNF formula with
the largest cardinality (MaxBS for short) is NP-hard [3]. In other words, it is
impossible to find the best decomposition in polynomial time unless P = NP.
So far a few decomposition algorithms were proposed. However, no algorithm
achieves optimization in all terms. PureDecompose [3] is the fastest, but its
quality is poor. To improve the quality, Heule et al [4] presented QuickDecom-
pose. However, QuickDecompose is time-consuming. Soon after, to improve the
speed, Balyo et al [4] developed a post-processing algorithm called EagerMover.
Through a exhaustive series of experiments, we noted that althou gh the de-
composition quality of PureDecompose + EagerMover (PureEager for short) and
QuickDecompose can outperform PureDecompose, their quality is not high yet.

This paper aims to present an algorithm for finding efficiently a high-qu ality
decomposition. To achieve this goal, we present two new variants of PureDecom-
pose, a new decomposition algorithm based on clause correlation degree, and
a new post-processing algorithm. In addition, we improve the existing BCE to
speed up the decomposition. The algorithm resulting from integrating these new
techniques is called MixDecompose, which can improve significantly the qual-
ity of decomposition. On application instances, the decomposition quality of
MixDecompose is better than that of PureEager. there is no application formula
where the quality of PureEager is better than MixDecompose. In terms of speed,
MixDecompose is still fast. On average, it took 8.97 seconds on our machine,
which is a little slower than PureEager took 7.41 seconds. However, in the worst
case, MixDecompose was faster than PureEager. The latter exceeded 300 seconds
in some cases, whereas the former took at most 110 seconds.

2 Preliminaries

In this section, we present basic concepts that will be used in subsequent algo-
rithms for blocked clause decomposition.

CNF. It is short for conjunctive normal form. A formula in CNF is formulated
as a conjunction of clauses, where each clause is a disjunction of literals, each
literal being either a Boolean variable or its negation. The negation of a variable
x is denoted by \( \bar{x} \) or \( \neg x \). In general, a clause \( C \) is written as
\( C = x_1 \lor \cdots \lor x_m \),
where \( x_i (1 \leq i \leq m) \) is a literal. A formula \( F \) is is written as
\( F = C_1 \land \cdots \land C_n \),
where \( C_i (1 \leq i \leq n) \) is a clause. The symbols \( \text{var}(F) \) and \( \text{lit}(F) \) denote the sets
of variables and literals occurring in a formula \( F \), respectively.

Resolution. Given two clauses \( C_1 = (l \lor a_1 \lor \cdots \lor a_m) \) and \( C_2 = (\bar{l} \lor b_1 \lor \cdots \lor b_n) \),
the clause \( C = (l \lor a_1 \lor \cdots \lor a_m \lor b_1 \lor \cdots \lor b_n) \) is called the resolvent of \( C_1 \) and
\( C_2 \) on the literal \( l \), which is denoted by \( C = C_1 \otimes_l C_2 \).

Blocked Clauses. Given a CNF formula \( F \), a clause \( C \), a literal \( l \in C \) is said
to block \( C \) w.r.t. \( F \) if (i) \( C \) is a tautology w.r.t. \( l \), or (ii) for each clause \( C' \in F \)
with \( \bar{l} \in C' \), \( C' \otimes \bar{l} C \) is a tautology. A clause is a tautology if it contains both \( x \)
and \( \bar{x} \) for some variable \( x \). When \( l \) blocks \( C \) w.r.t. \( F \), the literal \( l \) and the clause
\( C \) are called a blocking literal and a blocked clause, respectively.
BCE. It is short for blocked clause elimination, which removes blocked clauses from CNF formulas. By BCE($F$) we mean the CNF formula resulting from repeating the following operation until fixpoint: If there is a blocked clause $C \in F$ w.r.t. $F$, let $F := F - \{C\}$. It is said that BCE can solve a formula $F$ if and only if BCE($F$) = $\emptyset$. The seminal work in BCE is due to Kullmann [5].

3 Blocked Clause Decomposition

In theory, any CNF formula can be decomposed into two blocked subsets. However, not all the decompositions are effective. In general, the larger one of the blocked sets is, the better the decomposition quality is, since the larger it is, the more it resembles the original formula. Therefore, the size difference of the two sets is considered as a measure of the decomposition quality. Nevertheless, computing the largest blocked set from a CNF formula is NP-hard. Hence, we here aim to find a fast decomposition with higher quality, rather than the highest quality.

The simplest decomposition way is called pure decomposition, which is shown in Fig. 1. Let the symbols $L$ and $R$ denote the left(large) subset and the right (remainder) subset, respectively. For each variable $x$, this algorithm adds always the larger of $F_x$ and $F_\bar{x}$ to $L$ and the smaller to $R$, where $F_x$ ($F_\bar{x}$) is the set of clauses of $F$ where $x$ occurs positively (negatively). At the termination of this algorithm, we have $F = L \cup R$ with $|L| \geq |R|$. In Fig. 1, $\max\{F_x, F_\bar{x}\}$ means the set with the larger cardinality between $F_x$ and $F_\bar{x}$.

**PureDecompose**($F$)

$L := \emptyset$

for each variable $x \in \text{var}(F)$ do

$L := L \cup \max\{F_x, F_\bar{x}\}$

$F := F - (F_x \cup F_\bar{x})$

return $L$.

Fig. 1. Pseudo-code of PureDecompose algorithm

The advantage of this algorithm is that it can be easily implemented to run in linear time in the size of $F$, using a standard structure of occurrence lists. Therefore it is very fast. The drawback is that its decomposition quality is not high on many formulas. For this reason, we will improve it by combining some variants of it and the other algorithms.

By a few empirical observations, we found that the performance of PureDecompose rely significantly on the order in which variables are eliminated. Here, we present two variants of PureDecompose with different variable elimination ordering heuristics. The first variant is called min pure decomposition, which is shown in Fig. 2. One fifth of variable eliminations are to be eliminated in the same order as PureDecompose. The remaining variables are to be eliminated in order from the lowest occurrence of literals to the highest. If there are multiple literals with the lowest occurrence, the literal with the minimum total number
of clauses containing it is eliminated first. The total clause size of a literal $x$ can be formulated as $\sum_{C \in F_x} |C|$. 

$$\text{MinPureDecompose}(F)$$

$L := \emptyset$

$k := 0$

while $F \neq \emptyset$ do

if $k \mod 5 = 0$ then

select $u \in \text{vars}(F)$ in the order of variable No.

else $m = \min_{x \in \text{lit}(F)} |F_x|$

$u := \arg \min_{|F_x| = m} \sum_{C \in F_x} |C|$

$L := L \cup \max\{F_u, F_u\}$

$F := F - (F_u \cup F_u)$

$k := k + 1$

return $L$.

Fig. 2. Pseudo-code of MinPureDecompose algorithm

Compared to PureDecompose, this algorithm adds only the search of variables to be eliminated. This search can be done in $O(n \log n)$ time, using an order heap, where $n$ is the number of variables. In the actual implementation, the number of variables in $\min_{x \in \text{lit}(F)} |F_x|$ is limited to 30000. This can guarantee that MinPureDecompose is still very fast even if $n$ is very large. In terms of decomposition quality, this algorithm is superior to the other algorithms on some application instances such as ctl\{291\_567\_unsat\_pre.

Now we consider the second variant of PureDecompose. The order of its variable eliminations is opposite to that of the first variant. We call this variant max pure decomposition, which is shown in Fig. 3. It always eliminate first a literal with the highest occurrence. When multiple literals have the same highest occurrence, we select a variable with the lowest difference of its two literal occurrences. This can be done by computing $\min_{|F_x| = m} ||F_x| - |\bar{F}_x||$, where $m$ is defined as $\max_{x \in \text{lit}(F)} |F_x|$. Actually, the first variant can introduce also this tie-break method.

$$\text{MaxPureDecompose}(F)$$

$L := \emptyset$

while $F \neq \emptyset$ do

$m := \max_{x \in \text{lit}(F)} |F_x|$

$u := \arg \min_{|F_x| = m} ||F_x| - |\bar{F}_x||$

$L := L \cup \max\{F_u, F_u\}$

$F := F - (F_u \cup F_u)$

return $L$.

Fig. 3. Pseudo-code of MaxPureDecompose algorithm

Unlike the first variant, MaxPureDecompose needn’t compute the total clause size of each literal. So it should run faster than the first variant. In order to ensure that is still very fast even if the number of variables is very large, the range for finding a literal with the highest occurrence is limited to 5000 variables. The
decomposition quality of this algorithm is superior to that of the other algorithms on some application instances such as complete-500.

\[ \text{BCE} \text{(touched clauses } T, \text{ formula } F, \text{ blocked set } L) \]
for each clause \( C \in T \land F \) do
  for \( l \in C \) with \(|F| < 2 \) or \(|F| < 200000 \) do
    if all resolvent \( C \) on \( l \) in are tautologies, i.e., \( C \) is blocked then
      \( L := L \cup \{C\} \)
      \( F := F - \{C\} \)
      \( T := T \cup \text{touch}(C, F) \)
      continue with next \( C \) in outer loop

return \( L \)

Fig. 4. Pseudo-code of \( \text{BCE} \) algorithm

\[ \text{LessInterfereDecompose}(F) \]
\( L := R := S := \emptyset \)
\( \text{BCE}(F, F, L) \)
while \( F \neq \emptyset \) do
  if \( S \cap F = \emptyset \) then
    \( m = \min_{x \in \text{lit}(F)} |F_x| \)
    for each clause \( C \in F \) do
      for each clause \( e \in F_i \) with \( l \in C \) and \(|F_i| = m \) do
        \( \text{score}[e] := \text{score}[e] + 1 \)
      \( S := \{x | \text{score}[x] \geq \alpha, \text{ where the } p\text{-th highest score is } \alpha \} \)
      select a clause \( C \in S \cap F \)
      \( F := F - \{C\} \)
      \( \text{BCE} \text{(touch}(C, F), F, L) \)
  return \( L \)

Fig. 5. Pseudo-code of \( \text{LessInterfereDecompose} \) algorithm

The above two algorithms both are based on the order of variable elimination. However, in some cases, the algorithm obtained by optimizing the order of variable elimination is not necessarily optimal. Therefore, one must find more efficient algorithms to replace them. Blow we present a new algorithm called \( \text{less interfere decomposition} \), which is based on the order of clause elimination. Its pseudo-code is shown in Fig. 5. This algorithm may sketch the basic outline as follows: move blocked clauses in \( F \) to \( L \) by BCE, compute the candidate set \( S \), move each clause \( C \in S \cap F \) to \( R \). These steps are repeated until \( F \) is empty. The computation of the candidate set \( S \) is based on the notion of interfering degree. The interfering degree of a clause \( C \) can be defined as \( \sum_{C' \in F \land l \in \text{var}(C') \cap \text{var}(C)} \text{Ntaut}(C' \otimes l) \), where \( \text{Ntaut}(X) \) is zero if \( X \) is a tautology clause, and one otherwise. The probability that \( C' \otimes l \) is not a tautology clause is very high. To save the computing cost, we may approximate the interfering degree as \( \sum_{C' \in F} |\text{var}(C') \cap \text{var}(C)| \). \( \text{LessInterfereDecompose} \) in Fig. 5 uses this approximation version to compute the interfering degree, and call this
measure score, i.e., $\text{score}[C] = \sum_{C' \in F} |\text{var}(C') \cap \text{var}(C)|$. To get clauses with the maximum score, all the clauses in $F$ are traversed. If only one clause with the maximum score is moved to $R$ from $F$ each time $F$ is traversed, it is time consuming. So we decide to move $p$ clauses to $R$ one time, where $p = \left\lfloor \frac{|F|}{\theta} \right\rfloor$, where $\theta$ is a constant. In our experiment, when $|F| < 8 \times 10^4$, $\theta$ is set to 2300. Otherwise, $\theta$ is set to 400 for random instances, and 100 for non-random instances. When $\frac{|F|}{\theta} < 18$, $p$ is set to 18. As shown in Fig. 5, the clauses with the first $p$ highest scores are stored in $S$ as the candidate clauses to be moved to $R$. In order to save time further, we compute the interfering degree produced by only literals with the lowest occurrence, not all literals.

\textit{LessInterfereDecompose} calls to BCE shown in Fig. 4 to move blocked clauses to $L$. BCE here is different from that presented in \cite{6}. BCE in \cite{6} is based on a literal-based priority queue, while our BCE is based on a clause-based linear linked list. Another important difference from the usual BCE is that we do not try to test whether each literal $l$ in $C$ is a blocking literal when $|F| \geq 200000$. We test only literal $l$ with $|F_l| < 2$. That is, we replace the statement "for $l \in C$ do " in the usual BCE with the statement "for $l \in C$ with $(|F_l| < 2$ or $|F| < 200000)$ do". Due to this simplification, our BCE is much faster than the usual BCE. Surprisingly, the decomposition quality keep unchanged in most cases. Even if it is changed, its change is still very small. In addition, our touch function is different from that in \cite{6}. Here is our definition about it.

$$
touch(C, F) = \begin{cases} 
\bigcup_{x \in C \setminus \{l\}} F_x & |F| < 600000 \\
\bigcup_{x \in C \setminus \{l\}} F_x & \text{otherwise}
\end{cases}
$$

When $|F| \geq 600000$, we consider only the clauses touched by the negation of literals with the number of occurrences $< 2$. This can speed up the decomposition of large instances. For example, using the above touch, the runtime of decomposing on \textit{q-query}\_\textit{L90} can be reduced from 600 seconds to 9 seconds. The decomposition quality keep unchanged still.

The runtime of \textit{LessInterfereDecompose} consists of three parts: BCE, computing scores and determining $S$'s. The runtime of computing scores is $O\left(\frac{|F|^2}{p}\right) = O(\theta|F|)$. If scores are given, determining a $S$ can be done in a linear time in $|F|$, since there exist linear time algorithms for finding the $p$-th highest score \cite{7,8}. The total runtime of computing scores plus determining $S$'s does not exceed $O(\theta|F|)$. If the number of clauses touched by each clause does not exceed a constant $\delta$, where $\delta$ is certainly smaller than the maximal number of literal occurrences times the maximal size of clauses, i.e., $\max_{x \in \text{lit}(F)} |F_x| \times \max_{C \in F} |C|$. The total time required by all BCE's is at most $O(\delta|F|)$. Thus, the total runtime of \textit{LessInterfereDecompose} is at most $O((\delta + \theta)|F|)$). In practice, $\delta$ is generally very small. Should $\delta$ is very large, we can remove a part of touched clauses to reduce the time required by BCE to test whether a clause in the touch list is blocked, or limit the size of the touch list a small constant, say 2000. Using such a policy can guarantee that the time complexity of \textit{LessInterfereDecompose} is
linear in $|F|$. Compared with $EagerMover$ in [4], the runtime of $BCE$ in $LessInterfereDecompose$ is smaller than that in $EagerMover$. $EagerMover$ calls at least four times $BCE$ on a subset with the size of $0.75|F|$. All calls to $BCE$ on each clause $C$ in $F$ can be viewed as a call to $BCE$ on the whole $F$. Thus, the total runtime of $BCE$ in $LessInterfereDecompose$ corresponds to double the runtime of $BCE$ on a $F$. As long as the runtime of computing scores and determining $S$’s is smaller than the runtime of $BCE$ on a $F$, $LessInterfereDecompose$ should be faster than $EagerMover$. In fact, that is true. On some instances, the former are indeed faster than the latter.

In general, the above algorithms do not achieve maximal blocked set decomposition. However, they can be improved further by post-processing. The post-processing often used is $MoveBlockedClause$ algorithm shown in Fig. 7, which is to move blocked (with respect to the current $L$) clauses from $R$ to $L$. We noted that even if this post-processing algorithm is applied, the decomposition quality can be improved still. For this reason, we present a new post-processing algorithm, called $Right set guided decomposition$, which is shown in Fig. 6. It is a simplified version of $LessInterfereDecompose$. Replacing $S$ with $R$ results in this algorithm. This algorithm requires that the right blocked set $R$ must be given in advance. Hence, it is used generally as post-processing. It is faster than $LessInterfereDecompose$, since it need not computing $R$. Its time complexity depends mainly on that of $BCE$. For some benchmarks, this algorithm can improve significantly their decomposition quality. For example, to decompose SAT.dat.k75-24.rule.3 using $MinPureDecompose$, the fractions of the large subset (i.e., $|L|/|F|$) with $RsetGuidedDecompose$ and without it are 83.9% and 69.9%, respectively. If replacing $MinPureDecompose$ with $LessInterfereDecompose$, their fractions are 87.8% and 87.3%, respectively. $RsetGuidedDecompose$ raises still the quality by 0.5%. However, the speed difference among the three algorithms is big. On this instance, $LessInterfereDecompose$, $RsetGuidedDecompose$ and $MinPureDecompose$ spent 25, 4 and 1 seconds, respectively. The slowest $LessInterfereDecompose$ is not suitable for huge instances with ten millions of clauses.

In general, we do not know which algorithm is the best. Because each of the previous three algorithms is very fast, running the three algorithms one after another does not lose much time. Thus, we can construct an algorithm with high speed and high performance by combining the algorithms above-mentioned. The detailed implementation is shown in Fig. 7. We call this algorithm $MixDecom-

$$
RsetGuidedDecompose(formula F, \text{right set } R) \\
L := \emptyset \\
BCE(F, F, L) \\
\textbf{while} \ F \neq \emptyset \ \textbf{do} \\
\quad \text{select a clause } C \in (R \cap F) \\
\quad F := F - \{C\} \\
\quad BCE(touch(C, F), F, L) \\
\textbf{return} \ L
$$

Fig. 6. Pseudo-code of $RsetGuidedDecompose$ algorithm

In general, we do not know which algorithm is the best. Because each of the previous three algorithms is very fast, running the three algorithms one after another does not lose much time. Thus, we can construct an algorithm with high speed and high performance by combining the algorithms above-mentioned. The detailed implementation is shown in Fig. 7. We call this algorithm $MixDecom-$
pose. Its basic idea is to take the maximum from three left sets outputted by three algorithms as the initial $L$ first. If the formula to be decomposition is not large, say the number of clauses is less than $5 \times 10^6$, we invoke \textit{LessInterfereDecompose} to get a larger $L$. Finally, we enlarge the size of $L$ by calling two post-processing algorithms: \textit{RsetGuidedDecompose} and \textit{MoveBlockedClauses}. Like the usual post-processing, the task of \textit{MoveBlockedClauses} is to move blocked clauses from $R$ to $L$. Notice, if $F$ is large, say $|F| > 10^7$, the last post-processing can be canceled to save the running time.

\begin{verbatim}
MoveBlockedClauses(left blocked set L, right set R)
    for each clause C \in R do
        if BCE(L \cup \{C\}) = \emptyset then
            L := L \cup \{C\}
    return L

MixDecompose(formula F)
    L_1 := PureDecompose(F)
    L_2 := MinPureDecompose(F)
    L_3 := MaxPureDecompose(F)
    L := \max\{L_1, L_2, L_3\}
    if |F| < 5 \times 10^6 then
        L_4 := LessInterfereDecompose(F)
        L := \max\{L, L_4\}
    L := \max\{L, \text{RsetGuidedDecompose}(F, F - L)\}
    L := \text{MoveBlockedClauses}(L, F - L)
    return L
\end{verbatim}

Fig. 7. Pseudo-code of \textit{MoveBlockedClauses}, \textit{MixDecompose} algorithm

According to whether both subsets can be solved by BCE, a blocked clause decomposition can be classified as \textit{symmetric} or \textit{asymmetric}. If yes, it is symmetric. If only one of the subsets can be solved by BCE, it is asymmetric. Clearly, our two variants of \textit{PureDecompose} are symmetric. Like \textit{EagerMover} \footnote{EagerMover}, if blockable clauses (whose definition is given below) are allowed to move to $L$, \textit{RsetGuidedDecompose} are asymmetric, since it cannot guarantee that the left set can be solved by BCE. However, by our observation, on almost all application instances, they are symmetric.

\section{Empirical evaluation}

We evaluated the performance of decomposition algorithms under the following experimental platform: Intel Core 2 Quad Q6600 CPU with speed of 2.40GHz and 2GB memory.

Because \textit{QuickDecompose} \footnote{QuickDecompose} requires more time than \textit{EagerMover} \footnote{EagerMover} for many instances, we decided to select more competitive \textit{PureDecompose+EagerMover} \footnote{PureEager} (\textit{PureEager} for short) as our comparison object. \textit{PureEager} and \textit{MixDecompose} both are written in C.

The large set $L$ obtained by \textit{PureEager} contains blockable clauses in addition to blocked clauses. A clause $C$ is said to be blockable w.r.t. a blocked set $L$ if
each literal \( l \in C \) is not a blocking literal of any clause in \( L \). The reason why blockable clauses are add to the blocked set is that they do not destroy the blocked property. That is, blocked sets containing blockable clauses are still satisfiable. To keep identical with the performance evaluation of PureEager, the large set \( L \) of our MixDecompose contains also blockable clauses. In addition, before calling MixDecompose, we use the same unit decomposition policy given in [3] as EagerMover to preprocess the input formula.

We evaluated PureEager and MixDecompose on the 297 instances from the application track of the SAT competition 2014, except for three huge instances. Due to limited space, Table 1 lists only a part of results and a random instance in the last row. Column \(|F|\) is in \( 10^4 \) of clauses. Time is in seconds.

| Instances                        | \(|F|\) | PureEager | MixDecompose |
|----------------------------------|--------|-----------|--------------|
|                                  |        | \( |F| \)  | Time | \( |F| \)  | Time |
| 002-23-96                        | 13     | 97.7%     | 1.4  | 99.3%     | 0.29  |
| aes\textunderscore 24\textunderscore 4\textunderscore keyfind\textunderscore 4  | 1      | 57.5%     | 0.02 | 68%       | 0.11  |
| atco\textunderscore encl\textunderscore opt1\textunderscore 03\textunderscore 56 | 26     | 79.3%     | 0.43 | 83.5%     | 7.89  |
| blocks\textunderscore blocks\textunderscore 36\textunderscore 0.120         | 607    | 92.3%     | 17.4 | 96.4%     | 12.65 |
| complete\textunderscore 500\textunderscore 0.1\textunderscore 17            | 8      | 93.9%     | 2.2  | 96.4%     | 3.04  |
| dated\textunderscore 10\textunderscore 11\textunderscore u                     | 49     | 81.6%     | 1.43 | 82.6%     | 2.91  |
| dimacs                           | 1      | 99.9%     | 0.14 | 99.9%     | 0.04  |
| grid\textunderscore strips\textunderscore grid\textunderscore y\textunderscore 3\textunderscore 0.355    | 167    | 85.1%     | 5.61 | 95.1%     | 4.18  |
| hit2g\textunderscore 7\textunderscore 60\textunderscore 80                   | 3      | 73.8%     | 0.26 | 98.4%     | 1.02  |
| MD5\textunderscore 29\textunderscore 3                                    | 7      | 81.4%     | 0.29 | 99.3%     | 0.51  |
| openstacks\textunderscore p30\textunderscore 3\textunderscore 0.855     | 141    | 93.5%     | 1.73 | 94%       | 3.62  |
| partial\textunderscore 5\textunderscore 17\textunderscore s               | 101    | 74.5%     | 2.3  | 82.1%     | 5.66  |
| query\textunderscore 1\textunderscore 150\textunderscore w\textunderscore coli\textunderscore sat | 217    | 67.9%     | 52.4 | 85.8%     | 12.03 |
| query\textunderscore 1\textunderscore 90\textunderscore w\textunderscore coli\textunderscore sat | 118    | 67.8%     | 15.5 | 88.1%     | 9.04  |
| 9vliw\textunderscore m\textunderscore 8stage\textunderscore s\textunderscore c\textunderscore 1\textunderscore 7 | 1338   | > 300     | 86.8%| 108.8%    |
| 9dlx\textunderscore yliw\textunderscore at\textunderscore b\textunderscore q\textunderscore 6 | 364    | 95.2%     | 14.7 | 96.6%     | 12.05 |
| SAT\textunderscore \textunderscore k75\textunderscore 2\textunderscore k\textunderscore rule\textunderscore u | 415    | 78.9%     | 14.4 | 87.8%     | 33.83 |
| transport\textunderscore 3\textunderscore node\textunderscore 10000\textunderscore 4d | 590    | 92.5%     | 24.4 | 92.9%     | 15.66 |
| uni\textunderscore f\textunderscore k3\textunderscore r3\textunderscore 96\textunderscore v\textunderscore 10000\textunderscore 0\textunderscore c\textunderscore 3960000 | 396    | 76.3%     | 37.96| 83.2%     | 82.18 |

Table 1. We evaluated on the 297 instances from the application track of SAT 2014, except for three huge instances. Due to limited space, we list only a part of results and a random instance in the last row. Column \(|F|\) is in \( 10^4 \) of clauses. Time is in seconds.
huge instances within 110 seconds. However, PureEager was not able to finish on some benchmarks such as 9vliw.m9stage_iq3.C1_b7 within 300 seconds.

Table 2. Comparing performance of two algorithms on 297 benchmarks from SAT Competition 2014 application track.

| Algorithm     | Ave | # of best | # of eq | Ave Time | Time Out |
|---------------|-----|-----------|---------|----------|----------|
| PureEager     | 87.2% | 0         | 71       | 7.41     | 7        |
| MixDecompose  | 92.2% | 226       | 71       | 8.97     | 0        |

Table 2 shows the empirical result on 297 benchmarks from SAT Competition 2014 application track. Three huge instances zfcp-2.8-u2-nh, esawnuw3.debugged and post-cbmc-zfcp-2.8-u2 were removed. The second column shows the average fraction of the large set. Column ‘# of best’ indicates the number of the best results obtained by an algorithm. Column ‘# of eq’ is the number of results equivalent to another algorithm. On 226 out of 297 benchmarks, the size of the large set obtained by MixDecompose is larger than that obtained by PureEager. On 71 remaining benchmarks, the quality of the two algorithms is identical. There is no application formula where the quality of PureEager is better than MixDecompose. In addition, we conducted also experiments on random benchmarks. We observed that on all random instances, the quality of MixDecompose is strictly better than that of PureEager. As seen from the last row of Table 2, MixDecompose can solve huge random instances with millions of clauses in a reasonable time. For 3-SAT random instances, it can increase the fraction of the large set by 5%.

The fifth column in Table 2 shows the average runtime taken by each algorithm in seconds. Here, computing the average runtime counts only solved instances, excluding timed-out instances. The last column in Table 2, lists the number of times the time-out was hit. The timeout for each algorithm was set to 300 seconds. MixDecompose did not time out on the tested benchmarks, while PureEager did on 7 benchmarks. MixDecompose took at most 110 seconds. Although on average, PureEager run faster than MixDecompose, in this experiment, the worst-case runtime of the former was significantly larger than the latter.

According to results of four algorithms, MixDecompose outputs the best result among them. For 297 application instances, the number of the best results obtained by LessInterfereDecompose, PureDecompose, MaxPureDecompose and MinPureDecompose are 215, 18, 30 and 34, respectively. This empirical result shows that in most cases, LessInterfereDecompose is superior to the other algorithms.
5 Conclusions and Future work

In this paper, we developed a new blocked clause decomposition algorithm by combining several decomposition strategies. The new algorithm not only achieves high quality decomposition, but also is fast. Even for large instances, it can ensure that the decomposition is done within 110 seconds on our machine. Because our machine is slower than the platform of SAT competition 2014, if running on the latter, the speed will be more fast.

So far we know only that we can get a higher quality decomposition than the existing algorithms such as PureEager. However, this does not mean that MixDecompose is the best. How to develop a better and more efficient than MixDecompose will be a future research topic.

References

1. Parkes, A.J.: Clustering at the phase transition, Proc. of the 14th Nat. Conf. on AI, pp. 340–345 (1997)
2. Biere, A.: Lingeling, plingeling and treengeling entering the sat competition 2013, Proceedings of SAT Competition 2013, pp. 51C-52. University of Helsinki (2013)
3. Heule, M.J.H., Biere, A.: Blocked clause decomposition, LPAR-19 2013, LNCS 8312, pp. 423–438, Springer, (2013)
4. Balyo, T., Fröhlich, A., Heule, M.J.H., Biere, A.: Everything you always wanted to know about blocked sets (but were afraid to ask), SAT 2014, LNCS 8561, pp. 317–332 (2014)
5. Kullmann, O.: On a generalization of extended resolution, Discrete Applied Mathematics, 96–97, pp. 149–176 (1999)
6. Järvisalo, M., Biere, A., Heule, M.J.H.: Simulating circuit-level simplifications on cnf, Journal of Automated Reasoning 49(4) pp.583–619 (2012)
7. Musser D.R.: Introspective sorting and selection algorithms, Software Practice and Experience 27(8), pp.983–993 (1997)
8. Valois J.D.: Introspective sorting and selection revisited, Software Practice and Experience 30, pp.617–638 (2000)
9. EagerMover, Java source code: http://ktiml.mff.cuni.cz/~balyo/bcd/
10. SAT 2014 competition web page: http://www.satcompetition.org/2014/