Computers and Operations Research 84 (2017) 1–15

Contents lists available at ScienceDirect

Computers and Operations Research

journal homepage: www.elsevier.com/locate/cor

On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem

Chu-Min Li[a,b], Hua Jiang[a,*], Felip Manyà[c]

[a] Huazhong University of Sciences and Technology (HUST), China
[b] MIS, Université de Picardie Jules Verne, France
[c] Artificial Intelligence Research Institute (IIIA, CSIC), Spain

ARTICLE INFO

Article history:
Received 23 June 2015
Revised 22 February 2017
Accepted 23 February 2017
Available online 24 February 2017

Keywords:
Maximum clique problem
Branch-and-bound
Branching ordering
Incremental MaxSAT Reasoning

ABSTRACT

When searching for a maximum clique in a graph G, branch-and-bound algorithms in the literature usually focus on the minimization of the number of branches generated at each search tree node. We call *dynamic strategy* this minimization without any constraint, because it induces a dynamic vertex ordering in G during the search. In this paper, we introduce a *static strategy* that minimizes the number of branches subject to the constraint that a static vertex ordering in G must be kept during the search. We analyze the two strategies and show that they are complementary. From this complementarity, we propose a new algorithm, called MoMC, that combines the strengths of the two strategies into a single algorithm. The reported experimental results show that MoMC is generally better than the algorithms implementing a single strategy.

© 2017 Elsevier Ltd. All rights reserved.

1. Introduction

The maximum clique problem (MaxClique) is a relevant NP-hard problem with real-world applications in fields such as fault diagnosis (Berman and Pelc, 1990), bioinformatics and chemoinformatics (Barnes et al., 2005; Ravetti and Moscato, 2008), coding theory (Etzion and Östergård, 1998), economics (Boginski et al., 2006), and social network analysis (Balasundaram et al., 2011). A huge amount of effort has been devoted to solve MaxClique and, as a result, there exist two main types of algorithms (also called solvers when the algorithms are implemented): heuristic (e.g., Benlic and Hao, 2013; Grosso et al., 2008; Pullan and Hoos, 2006; Pullan et al., 2011) and exact, including algorithms based on the Branch-and-Bound (BnB) scheme (e.g., Babel and Tinhofer, 1990; Carraghan and Pardalos, 1990; Fahle, 2002; Konc and Janežič, 2007; Li et al., 2013; Li and Quan, 2010a; 2010b; Östergård, 2002; Régin, 2003; Segundo et al., 2011; Tomita and Kameda, 2007; Tomita and Seki, 2003; Tomita et al., 2010). See Wu and Hao (2015) for a recent survey. In addition to the previous algorithms, there are also other related approaches that can be used to solve MaxClique. For example, the dynamic programming algorithm using Boolean-width in Sharmin (2014) and the branch-and-cut algorithms in Rebennack et al. (2012), which solve the equivalent maximum independent set (MIS) problem.

In this paper we focus on BnB MaxClique algorithms. Given a graph $G = (V, E)$, the objective of these algorithms is to find a clique of maximum size in G by efficiently traversing the search space formed by all the proper subsets of the set of vertices V. At each node of the search space, and knowing that the largest clique found so far has size r, the algorithm partitions V into a set A, such that the subgraph induced by A has a maximum clique of size not greater than r, and a set $B = \{b_1, b_2, \ldots, b_{|B|}\}$ of branching vertices. Then, the algorithm recursively searches for a clique containing $b_i \in B$, of size greater than r, in the subgraphs induced by $\{b_i, b_{i+1}, \ldots, b_{|B|}\} \cup A$ for $i = |B|, |B| - 1, \ldots, 1$. The algorithm returns the largest clique found after traversing all the search space.

The main issue addressed in our work is the generation of the sets A and B in such a way that the set of branching vertices B is minimized. The common approach in the literature, which we call *dynamic strategy*, minimizes B as much as possible, inducing a dynamic vertex ordering over the vertices of G. In this paper, we show how to reduce the cardinality of B by applying incremental MaxSAT reasoning, and introduce a *static strategy* that reduces the cardinality of B subject to the constraint that the vertices of B are smaller than the vertices of A w.r.t. a static ordering defined at the beginning of the search and kept during the search process. Then, we compare the performance of the static strategy with the dynamic strategy. The comparison suggests that the

* Corresponding author.
E-mail addresses: chu-min.li@u-picardie.fr (C.-M. Li), jh_hgt@163.com (H. Jiang), felip@iiia.csic.es (F. Manyà).

http://dx.doi.org/10.1016/j.cor.2017.02.017
0305-0548/© 2017 Elsevier Ltd. All rights reserved.

dynamic and static strategies are complementary. So, we propose to combine them for developing BnB MaxClique algorithms.

This paper develops further the work reported in Li et al. (2015), which is an extended abstract that roughly describes two BnB MaxClique algorithms: DoMC, which implements a dynamic strategy when minimizing the set of branching vertices B, and SoMC, which implements a static strategy. It also shows that SoMC dominates DoMC. The analysis of the poor performance of DoMC led us to describe two new variants in this paper: DoMC2 and SoMC2. They improve DoMC and SoMC by efficiently handling the adjacency matrix and integrating an incremental upper bound. The conducted experiments indicate that DoMC2 and SoMC2, with the new solving techniques, are complementary. Consequently, we develop MoMC, which is a highly competitive algorithm combining techniques of both DoMC2 and SoMC2.

The paper is organized as follows. Section 2 presents the preliminaries. Section 3 describes algorithm DoMC_0, a basic BnB algorithm for MaxClique. Sections 4 and 5 describe DoMC and SoMC, respectively, and detail the incremental MaxSAT reasoning that those algorithms implement. Section 6 empirically compares the functions used by DoMC_0, DoMC and SoMC to generate the set of branching vertices B. Section 7 describes algorithms DoMC2_0, DoMC2 and SoMC2, which are improved variants of DoMC_0, DoMC and SoMC, respectively. Section 8 describes algorithm MoMC, which combines the strengths of DoMC2 and SoMC2. Section 9 analyzes the experimental investigation, considering harder instances and more algorithms than the ones reported in Li et al. (2015). Section 10 contains the concluding remarks.

2. Preliminaries

In this section, we define some basic concepts about graphs, and then the MaxClique and graph coloring problems. Finally, we define the MaxSAT problem, two MaxSAT encodings of MaxClique, and introduce unit propagation and failed literal detection.

2.1. Graph problems

Let $G = (V, E)$ be an undirected graph, where V is a set of n vertices $\{v_1, v_2, \ldots, v_n\}$ and E is a set of m edges. Two vertices v_i and v_j of V are adjacent if edge $(v_i, v_j) \in E$. The set of adjacent vertices of a vertex v in G is denoted by $\Gamma(v) = \{v' | (v, v') \in E\}$ and are called the neighbors of v. The cardinality of $\Gamma(v)$, denoted by $|\Gamma(v)|$, is the degree of v. Let V' be a subset of V, the subgraph of G induced by V', denoted by $G[V']$, is defined as $G[V'] = (V', E')$, where $E' = \{(v_i, v_j) \in E | v_i, v_j \in V'\}$. The density of a graph with n vertices and m edges is $2m/(n(n-1))$. An independent set of vertices is a subset D of V in which no two vertices are adjacent. An independent set partition of V is a partition of the vertices of V into independent sets such that each vertex belongs to exactly one independent set.

A clique in a graph G is a subset C of V such that every two vertices in C are adjacent. The cardinality of C, denoted by $|C|$, is the size of the clique. The Maximum Clique problem (MaxClique for short) for G is to find a clique of maximum size $\omega(G)$ in G. Note that a graph can have several different maximum cliques.

The graph coloring problem (GCP) for an undirected graph G is to assign a color to each vertex of G in such a way that adjacent vertices are assigned different colors. The chromatic number of G, denoted by $\chi(G)$, is the minimum number of colors needed to color G. It holds that $\chi(G)$ is an upper bound of $\omega(G)$; i.e., $\omega(G) \leq \chi(G)$. Recall that coloring a graph is equivalent to partitioning its vertices into independent sets in such a way that each independent set is formed by all vertices with the same color. In the following, we assume that colors are represented by consecutive integers starting with 1. Given a vertex v_i and a graph coloring algorithm, function color(v_i) returns the color assigned to v_i by the algorithm.

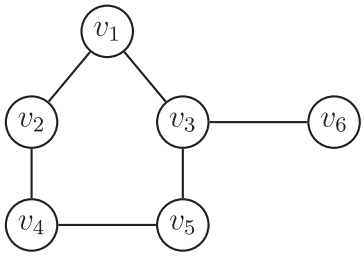

Fig. 1. A graph with $\chi(G)$=3 and $\omega(G)$=2.

2.2. MaxSAT and its relation with MaxClique

A literal is a propositional variable x or its negation $\bar{x}$, a clause is a disjunction of literals, and a conjunctive normal form (CNF) formula is a conjunction of clauses. A CNF formula is also represented as the set of its clauses, and a clause as the set of its literals. A truth assignment is a mapping of each propositional variable to true (1) or false (0). A truth assignment I satisfies a literal x ($\bar{x}$) if variable x is assigned the value 1 (0), satisfies a clause c (i.e., $I(c) = 1$) if it satisfies at least one of its literals, and satisfies a CNF formula if it satisfies all its clauses. A partial MaxSAT instance is a set of clauses in which some clauses are declared to be hard and the other are declared to be soft. Given a partial MaxSAT instance, the partial MaxSAT problem is to find a truth assignment that satisfies all the hard clauses and the maximum number of soft clauses.

MaxClique can be reduced to partial MaxSAT. Given a graph $G = (V, E)$, we define a propositional variable x_i for each vertex $v_i \in V$ with the intended meaning that x_i is true (false) if v_i belongs (does not belong) to the maximum clique C, and derive the partial MaxSAT instance ϕ that contains (i) a hard clause $\bar{x}_i \vee \bar{x}_j$ for each pair of non-adjacent vertices v_i and v_j, stating that v_i and v_j cannot be both in C, and (ii) a soft unit clause x_i for each vertex $v_i \in V$. Maximizing the number of satisfied soft clauses while satisfying all hard clauses gives C. Given an optimal assignment of ϕ, the derived maximum clique is formed by all vertices for which the variables are assigned the value 1, which means true. Li and Quan (2010b) improved this naive encoding by computing a partition of G into independent sets, and instead of adding a soft clause for every vertex, they added a soft clause for each independent set in the partition, which is the disjunction of the variables for the vertices in the independent set. Note that at most one vertex in each independent set can belong to C. In the rest of the paper, we always use the improved encoding, and refer to it as the Partial MaxSAT encoding of MaxClique.

Example 1 (from Li and Quan (2010b))**.** Consider the graph in Fig. 1, whose vertices can be partitioned into three independent sets: $\{v_1, v_4, v_6\}$, $\{v_2, v_3\}$, $\{v_5\}$. The naive encoding is formed by the hard clauses: $\{\bar{x}_1 \vee \bar{x}_4, \bar{x}_1 \vee \bar{x}_5, \bar{x}_1 \vee \bar{x}_6, \bar{x}_2 \vee \bar{x}_3, \bar{x}_2 \vee \bar{x}_5, \bar{x}_2 \vee \bar{x}_6, \bar{x}_3 \vee \bar{x}_4, \bar{x}_4 \vee \bar{x}_6, \bar{x}_5 \vee \bar{x}_6\}$, and the soft clauses $\{x_1, x_2, x_3, x_4, x_5, x_6\}$. The improved encoding is like the naive encoding but with the following soft clauses: $\{x_1 \vee x_4 \vee x_6, x_2 \vee x_3, x_5\}$.

Deciding the satisfiability of a CNF formula is an NP-complete problem. So, to detect contradictions in a set of clauses within a reasonable time, we apply unit propagation and failed literal detection, which have polynomial time complexity. Note that these two methods are incomplete because they do not guarantee the detection of all the contradictions in any set of clauses.

The unit resolution rule states that a CNF formula containing a unit clause x_i ($\bar{x}_i$) is satisfiable iff the CNF formula obtained by removing all the occurrences of $\bar{x}_i$ (x_i) is satisfiable. Unit propagation (UP) applies the unit resolution rule until the empty clause is derived (in this case the formula is declared to be unsatisfiable), or a saturation state is reached.

Given a CNF formula ϕ, a literal l fails if UP determines that $\phi \cup \{l\}$ is unsatisfiable. Given a clause $c = l_1 \vee \cdots \vee l_k$, if $\phi \cup \{l_i\}$ fails for $i = 1, \ldots, k$, then the CNF formula $\phi \cup \{c\}$ is unsatisfiable.

3. DoMC$_0$: a basic BnB algorithm for MaxClique

The search space of a BnB algorithm that finds a maximum clique in a graph $G = (V, E)$ is formed by all the subsets of V. The main difference among the existing BnB MaxClique algorithms lies in the way the search space is explored.

BnB MaxClique algorithms maintain a global variable $C_{\max}$ that stores the largest clique found so far in G. At every search tree node, they try to find a clique of size greater than $|C_{\max}|$, stopping the search when the search space is exhausted and backtracking when the search space below the current node does not contain any clique of size greater than $|C_{\max}|$. To this end, the algorithms partition the set of vertices V into two sets, A and B, in such a way that the size of a maximum clique in A is not greater than $|C_{\max}|$, and $B = V \setminus A = \{b_1, b_2, \ldots, b_{|B|}\}$ is called the set of branching vertices. If B is empty, the search is pruned. Otherwise, the algorithms recursively search for a maximum clique containing $b_i \in B$ in the subgraphs induced by $\{b_i, b_{i+1}, \ldots, b_{|B|}\} \cup A$ for $i = |B|, |B| - 1, \ldots, 1$. State-of-the-art BnB MaxClique algorithms focus on the minimization of B to reduce the search space.

For determining A and B, and reducing the cardinality of B, BnB algorithms like MCS (Tomita et al., 2010), MaxCliqueDyn (Konc and Janežič, 2007) and MaxCLQ (Li and Quan, 2010a; 2010b) compute an independent set partition of V using a greedy graph coloring algorithm that successively assigns the smallest possible color to each vertex in V based on a predefined ordering, ensuring that adjacent vertices are assigned different colors. Then, $A = \{v | v \in V \text{ and } color(v) \leq |C_{\max}|\}$ and $B = V \setminus A$. Since a clique of size $|C_{\max}|$ needs at least $|C_{\max}|$ colors to be colored and A can be colored using $|C_{\max}|$ colors, the size of a maximum clique in $G[A]$ is never greater than $|C_{\max}|$.

In MCS, the cardinality of B is further reduced, during the coloring process, using a procedure called *Re-NUMBER*, that re-colors some vertices in order to increase the number of vertices with a color not greater than $|C_{\max}|$. Recently, an approach inspired by constraint programming techniques has been proposed to reduce the branching in BnB MaxClique algorithms McCreesh and Prosser (2014).

In this paper, we present several functions to minimize B. Each function, when implemented in the generic BnB scheme MC described in Algorithm 1, results in a different BnB MaxClique solver. The template of these functions is GetBranches(G, r, O_0), where O_0 is a predefined vertex ordering and r is an integer related to $|C_{\max}|$. All the functions return a set B of branching vertices by showing that $A = V \setminus B$ does not contain any clique of size greater than r. For finding a maximum clique in a graph G, the initial call to algorithm MC should be MC(G, V, O_0, $\emptyset$, $\emptyset$). The candidate set P, which is initialized to V, contains vertices that may potentially be added to the current growing clique C.

Algorithm 2 defines the first function GetBranches of this paper, called *GetBranches*$_{d0}$. It was inspired by the way of computing A and B in the BnB algorithms MCS, BBMCL (Segundo and Tapia, 2014) and IncMaxCLQ (Li et al., 2013). Given a vertex ordering O_0, GetBranches$_{d0}$ returns a set of branching vertices by maximizing the number of vertices that are colored with a color not greater than r. *Re-NUMBER* works as follows: when a vertex v cannot be colored with a color not greater than r during the coloring process, the procedure checks whether there exists a vertex u, and two colors c_1 and c_2 not greater than r, such that u is the only neighbor of v colored with c_1 but u might be colored with c_2. In this case, u is re-colored with c_2 and v is colored with c_1, obtaining in this way an additional vertex in A and reducing the number of branching vertices in B.

Algorithm 1: MC($G, P, O_0, C, C_{\max}$), a generic BnB algorithm for MaxClique.

Input: $G = (V, E)$, a candidate set P, an ordering O_0 over P, the current growing clique C, and the largest clique $C_{\max}$ found so far in G.
Output: $C \cup C'$, where C' is a maximum clique of $G[P]$, the subgraph of G induced by P, if $|C \cup C'| > |C_{\max}|$; $C_{\max}$ otherwise.

```
1  begin
2      if P = ∅ then
3          return C;
4      B ← GetBranches(G[P], |C_max| − |C|, O_0);
5      if B = ∅ then
6          return C_max;
7      A ← P \ B;
8      Let B = {b_1, b_2, ..., b_|B|} and b_1 < b_2 < ··· < b_|B| w.r.t. O_0;
9      for i := |B| downto 1 do
10         C_1 ←
           MC(G, Γ(b_i) ∩ ({b_{i+1}, ..., b_|B|} ∪ A), O_0, C ∪ {b_i}, C_max) ;
11         if |C_1| > |C_max| then
12             C_max ← C_1;
13     return C_max;
```

Example 2. Consider the graph in Fig. 2, and assume that the vertex ordering O_0 is $v_1 < v_2 < v_3 < \cdots < v_{10}$, $|C_{\max}| = 2$ and $|C| = 0$. GetBranches$_{d0}$(G, 2, O_0) successively inserts v_{10}, v_9, v_8 and v_7 into two independent sets $D_1 = \{v_{10}, v_8\}$ and $D_2 = \{v_9, v_7\}$. Then, vertex v_6 cannot be inserted into D_1 and D_2, because it is adjacent to v_8 in D_1 and to v_9 in D_2. The Re-NUMBER procedure is applied to move v_8 from D_1 to D_2, and to insert v_6 into D_1. Afterwards, GetBranches$_{d0}$(G, 2, O_0) fails to insert v_5, v_4, v_3 and v_2 even with the Re-NUMBER procedure, before inserting v_1 into D_1. Finally, GetBranches$_{d0}$(G, 2, O_0) returns $B_{d0} = \{v_2, v_3, v_4, v_5\}$, and $A_{d0} = \{v_1, v_6, v_7, v_8, v_9, v_{10}\}$. Note that v_1 in A_{d0} is smaller than all the branching vertices in B_{d0} w.r.t. O_0, while the other vertices in A_{d0} are greater than all vertices in B_{d0} w.r.t. O_0.

The vertex ordering O_0 is a key component of efficient BnB MaxClique algorithms. A typical vertex ordering $v_1 < v_2 < v_3 < \cdots < v_n$, proposed in Carraghan and Pardalos (1990) and called *degeneracy ordering* in Eppstein and Darren (2011), is computed as follows: v_1 is the vertex with the smallest degree in G, v_2 is the vertex with the smallest degree in G after removing v_1, and so on. The degeneracy ordering (or a refinement of it) has shown to be effective in computing good quality upper bounds in many state-of-the-art BnB algorithms. In MCR (Tomita and Kameda, 2007), MCS, and BBMC (Segundo et al., 2011), the ordering is derived at the beginning of the search in a preprocessing step, and is then used to compute A and B in every recursive call of the algorithm (i.e., in every search tree node). In MaxCliqueDyn and MaxCLQ, the ordering is dynamically re-computed in the search tree nodes near the tree root to better minimize B there.

Another effective vertex ordering, proposed in Li et al. (2013) and called *MIS ordering*, is computed by repeatedly find-

Algorithm 2: GetBranches$_{d0}(G, r, O_0)$, for a BnB algorithm searching for a maximum clique containing more than r vertices in G.

Input: a graph $G = (V, E)$, an integer r and a vertex ordering O_0 over V
Output: a set B_{d0} of branching vertices

```
1  begin
2      B_d0 ← ∅; Π ← ∅; /* Π will be an independent set partition of V */
3      while V is not empty do
4          v ← the greatest vertex of V w.r.t. the ordering O_0;
5          remove v from V;
6          if there is an independent set D in Π in which v is not adjacent to any vertex then
7              insert v into D;
8          else
9              if |Π| < r then
10                 create a new independent set D = {v}; Π ← Π ∪ {D};
11             else
12                 /* Re-NUMBER */
13                 if there is a D in which v has only one adjacent vertex u, and u can be inserted into another independent set D' then
14                     insert u into D'; insert v into D;
15                 else
16                     insert v into B_d0;
17     return B_d0;
```

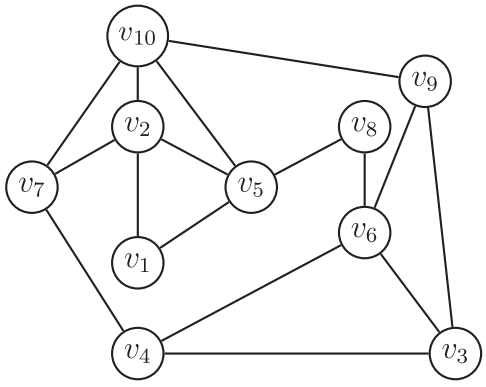

Fig. 2. The graph for Examples 2, 4, 6 and 7.

ing a maximum independent set S_i in G and removing S_i from G for $i = 1, 2, \ldots$ until G becomes empty. Then, the MIS ordering is defined as follows: given two vertices v_i and v_j, $v_i < v_j$ iff $v_i \in S_p$ and $v_j \in S_q$ and ($p > q$ or ($p = q$ and $v_i < v_j$ w.r.t. the degeneracy ordering)).

IncMaxCLQ automatically switches between the degeneracy ordering and the MIS ordering as follows:

1. If the density of G is smaller than 0.7, then O_0 is the degeneracy ordering, because computing the MIS ordering is hard in this case.
2. Otherwise, compute the MIS ordering by searching for successive maximum cliques S_1, S_2, ..., in the complementary graph $\overline{G}$ of G as follows. First, sort the vertices of $\overline{G}$ in the degeneracy ordering. Second, search for the first maximum clique S_1 of $\overline{G}$ using IncMaxCLQ, and then search for the second maximum clique S_2 using IncMaxCLQ in the reduced $\overline{G}$ after removing the vertices of S_1, and so on, until $\overline{G}$ becomes empty. The obtained S_1, S_2, ... are the successive maximum independent sets of G. If there are at least two maximum independent sets containing exactly one vertex, then O_0 is the degeneracy ordering, otherwise O_0 is the MIS ordering. Since the search is performed in the complementary graphs, whose densities are below 0.3, computing the MIS ordering takes less than one second in all the graphs considered in the experiments. Thus, the overhead of computing the MIS ordering is negligible compared with the total solving time.

In the following, when we mention algorithm DoMC$_0$, we refer to algorithm MC implementing function GetBranches$_{d0}$. DoMC$_0$ stands for "Dynamic ordering MaxClique solver", because the vertices in B_{d0} are not always smaller than the vertices in $A_{d0} = V \setminus B_{d0}$ w.r.t. O_0, so that the ordering between them should be dynamically re-defined.

4. DoMC: a BnB algorithm for MaxClique with incremental MaxSAT reasoning

We first present a state-of-the-art approach to improve an upper bound of $\omega(G)$ based on applying MaxSAT reasoning. Then, we define a new way of applying MaxSAT reasoning that, besides improving the quality of the upper bound, enables to reduce the cardinality of the set of branching vertices. As a result, we describe a new and improved variant of GetBranches: GetBranches$_d$.

4.1. MaxSAT reasoning to improve an upper bound of ω(G)

Algorithms such as MCQ (Tomita and Seki, 2003), MCR (Tomita and Kameda, 2007), MaxCliqueDyn (Konc and Janežič, 2007), MCS (Tomita et al., 2010) and MaxCLQ (Li and Quan, 2010a; 2010b) compute a partition of the set of vertices V of a graph G into r independent sets and use r as an upper bound of $\omega(G)$. Nevertheless, this upper bound may not be tight enough, even when r is the chromatic number of the graph. For example, $\{\{v_1, v_4, v_6\}, \{v_2, v_3\}, \{v_5\}\}$ is an optimal independent set partition of the vertices of the graph in Fig. 1, but its maximum cliques have size two.

A subset of q independent sets is said *conflicting* if the q independent sets cannot form a clique of size q. A suitable approach to improve that upper bound r is to apply MaxSAT reasoning to detect disjoint conflicting subsets of independent sets (Li et al., 2013; Li and Quan, 2010a; 2010b), which was shown to be very effective in MaxCLQ and IncMaxCLQ. It holds that (Li and Quan, 2010b):

Proposition 1. *Let G be a graph that can be partitioned into r independent sets. If t disjoint conflicting subsets of independent sets can be detected, then $\omega(G) \leq r - t$.*

Proof. Let G be partitioned into $\Pi = \{D_1, D_2, \ldots, D_r\}$, where each D_i ($1 \leq i \leq r$) is an independent set, let $\Pi_1, \Pi_2, \ldots, \Pi_t$ be the t detected disjoint conflicting subsets of Π, and let C be a maximum clique of G. Without loss of generality, let us write Π_i ($1 \leq i \leq t$) as $\{S_{i1}, S_{i2}, \ldots, S_{i|\Pi_i|}\}$ and let $|\Pi_1| + |\Pi_2| + \cdots + |\Pi_t| = p$ with $p \leq r$ and $\Pi_1 \cup \Pi_2 \cup \cdots \cup \Pi_t = \{D_1, D_2, \ldots, D_p\}$. Since Π_i is conflicting, $\sum_{j=1}^{|\Pi_i|} |C \cap S_{ij}| \leq |\Pi_i| - 1$, because there exists some j such that $|C \cap S_{ij}| = 0$. So, $\sum_{i=1}^{t} \sum_{j=1}^{|\Pi_i|} |C \cap S_{ij}| \leq \sum_{i=1}^{t} (|\Pi_i| - 1) = p - t$, and $|C| = \sum_{j=1}^{r} |C \cap D_j| = \sum_{i=1}^{t} \sum_{j=1}^{|\Pi_i|} |C \cap S_{ij}| + \sum_{j=p+1}^{r} |C \cap D_j| \leq (p - t) + (r - p) = r - t$. □

Given a graph $G = (V, E)$ and an independent set partition of V, we first create the partial MaxSAT encoding defined in Section 2, and then apply failed literal detection to identify disjoint conflicting subsets of independent sets. Actually, the MaxSAT encoding

is implicit in MaxCLQ and IncMaxCLQ. The vertices are directly treated as propositional variables, and the independent sets as soft clauses. In addition, the non-adjacency relations between vertices (each vertex is associated with the list of its non-adjacent vertices during the preprocessing) represent the hard clauses. In this way, once G is partitioned into independent sets, failed literal detection can be applied to the obtained MaxSAT instance.

A clause C containing $w+1$ literals has more possibilities to be satisfied than a clause C' containing w literals, because C is satisfied when any of its $w+1$ literals is satisfied. Hence, C is considered to be *weaker* than C'. A unit clause l has only one possibility to be satisfied and is considered to be the strongest case, because the satisfaction of l assigns a fixed value to the variable in l, removes $\bar{l}$ from the clauses containing $\bar{l}$, and derives a unit clause if $\bar{l}$ is removed from a clause with two literals. The unit clause is then recorded as the reason for the value of the variable. Unit propagation consists in repeatedly satisfying unit clauses until an empty clause is found, or there are no more unit clauses.

Example 3 (from Li and Quan (2010b))**.** Consider the graph G in Fig. 1, and the partial MaxSAT encoding in Example 1. We apply unit propagation to prove that G has no clique of size three because the set formed by the three independent sets is conflicting. The satisfaction of the unit clause x_5 sets $x_5 = 1$, and the clause x_5 is recorded as the reason for $x_5 = 1$. Since $x_5 = 1$ cannot satisfy the hard clauses $\bar{x}_1 \vee \bar{x}_5$, $\bar{x}_2 \vee \bar{x}_5$ and $\bar{x}_5 \vee \bar{x}_6$, the occurrences of $\bar{x}_5$ are removed from these clauses and they become unit clauses. In other words, there is only one way to satisfy these hard clauses after setting x_5 to 1. They are then satisfied and recorded as the reason for $x_1 = 0$, $x_2 = 0$ and $x_6 = 0$, respectively. Since $x_1 = 0$, $x_2 = 0$ and $x_6 = 0$ cannot satisfy the soft clauses $x_1 \vee x_4 \vee x_6$ and $x_2 \vee x_3$, the occurrences of x_1, x_2 and x_6 are removed from the two clauses and they become unit clauses. The satisfaction of the new unit clauses sets $x_4 = 1$ and $x_3 = 1$, and the two clauses are recorded as the reason for $x_4 = 1$ and $x_3 = 1$, respectively. However, $x_4 = 1$ and $x_3 = 1$ do not satisfy the hard clause $\bar{x}_3 \vee \bar{x}_4$, which becomes empty. This contradiction implies that the set of clauses involved in the derivation of the empty clause cannot be satisfied simultaneously, because the clauses can only be satisfied in a unique way before reaching the empty clause. We now collect all the clauses involved in the derivation of the empty clause with unit propagation and push them into a queue Q, starting with the clause $\bar{x}_3 \vee \bar{x}_4$ that became empty.

In a real implementation of unit propagation, literals are marked as removed but are not physically deleted. So, we easily find the two literals $\bar{x}_3$ and $\bar{x}_4$ occurring in the empty clause. The reasons for $x_3 = 1$ and $x_4 = 1$ are $x_2 \vee x_3$ and $x_1 \vee x_4 \vee x_6$, respectively, and both clauses are pushed into Q. We now process the second clause in Q: $x_2 \vee x_3$. The reason for $x_2 = 0$ is $\bar{x}_2 \vee \bar{x}_5$, which is pushed into Q, and the reason for $x_3 = 1$ is $x_2 \vee x_3$, which is ignored because it is already in Q. Next, we process the third clause in Q: $x_1 \vee x_4 \vee x_6$. The reasons for $x_1 = 0$, $x_4 = 1$ and $x_6 = 0$ are $\bar{x}_1 \vee \bar{x}_5$, $x_1 \vee x_4 \vee x_6$ and $\bar{x}_5 \vee \bar{x}_6$, respectively, thus $\bar{x}_1 \vee \bar{x}_5$ and $\bar{x}_5 \vee \bar{x}_6$ are pushed into Q ($x_1 \vee x_4 \vee x_6$ is already in Q and is ignored). We process the fourth clause in Q: $\bar{x}_2 \vee \bar{x}_5$. The reason for $x_2 = 0$ is $\bar{x}_2 \vee \bar{x}_5$, which is ignored because it is already in Q, and the reason for $x_5 = 1$ is x_5, which is pushed into Q. We process the fifth clause in Q: $\bar{x}_1 \vee \bar{x}_5$. The reasons for $x_1 = 0$ and $x_5 = 1$ are already in Q, and are both ignored. We process the sixth clause in Q: $\bar{x}_5 \vee \bar{x}_6$. The reasons for $x_5 = 1$ and $x_6 = 0$ are already in Q, and are both ignored. Finally, we process the last clause in Q: x_5. The reason for $x_5 = 1$ is already in Q, and is ignored. Hence, the set of clauses involved in the derivation of the empty clause with unit propagation is $Q = \{\bar{x}_3 \vee \bar{x}_4, x_2 \vee x_3, x_1 \vee x_4 \vee x_6, \bar{x}_2 \vee \bar{x}_5, \bar{x}_1 \vee \bar{x}_5, \bar{x}_5 \vee \bar{x}_6, x_5\}$. Since the hard clauses must be satisfied by any feasible assignment, we conclude that the three soft clauses $x_1 \vee x_4 \vee x_6$, $x_2 \vee x_3$ and x_5 cannot be satisfied simultaneously, implying that the upper bound of $\omega(G)$ is 2, instead of 3.

Table 1
Collecting the clauses making $\bar{x}_3 \vee \bar{x}_4$ empty in Example 3.

Treated clause	Treated Vars.	New reasons in Q	Vars. already treated
		$\bar{x}_3 \vee \bar{x}_4$	
$\bar{x}_3 \vee \bar{x}_4$	x_3, x_4	$x_2 \vee x_3$, $x_1 \vee x_4 \vee x_6$	
$x_2 \vee x_3$	x_2	$\bar{x}_2 \vee \bar{x}_5$	x_3
$x_1 \vee x_4 \vee x_6$	x_1, x_6	$\bar{x}_1 \vee \bar{x}_5$, $\bar{x}_5 \vee \bar{x}_6$	x_4
$\bar{x}_2 \vee \bar{x}_5$	x_5	x_5	x_2
$\bar{x}_1 \vee \bar{x}_5$			x_1, x_5
$\bar{x}_5 \vee \bar{x}_6$			x_5, x_6
x_5			x_5

Table 1 shows the evolution of Q. Each line gives a processed clause, the variables whose reason must be pushed into Q, and the variables that are ignored because their reason was already pushed into Q. Note that unit propagation can be reproduced using the clauses in the column "New reasons in Q" from bottom to top.

Failed literal detection consists in adding a unit clause l to a MaxSAT instance and then apply unit propagation. If unit propagation derives an empty clause, then l is a *failed literal*. Given a soft clause c, adding a literal l of c to the MaxSAT instance is equivalent to testing one possibility to satisfy c. If l is a failed literal, c cannot be satisfied by l. If all the literals of c are failed (i.e., all possibilities to satisfy c fail), the union of the clauses making each literal of c failed, together with c, cannot be satisfied simultaneously, and all the soft clauses involved in the derived contradictions give a conflicting subset of independent sets. In Example 3, $\{\{v_1, v_4, v_6\}, \{v_2, v_3\}, \{v_5\}\}$ is a conflicting subset of independent sets. Once such a subset has been detected, the soft clauses must be weakened before detecting further conflicts.

Definition 1. Let ϕ be a MaxSAT instance, and let $U = \{c_1, c_2, \ldots, c_q\}$ be a conflicting subset of soft clauses of ϕ. Weakening the clauses in U refers to the following procedure: If $q = 2$, replace the clauses c_1 and c_2 with $c_1 \vee c_2$. If $q > 2$, add a fresh propositional variable z_i to each clause c_i, for $i = 1, \ldots, q$, and add the hard constraint $z_1 + z_2 + \cdots + z_q = 1$ to the MaxSAT instance ϕ.

For example, weakening the clauses of $U = \{x_1 \vee x_4 \vee x_6, x_2 \vee x_3, x_5\}$ returns $U' = \{x_1 \vee x_4 \vee x_6 \vee z_1, x_2 \vee x_3 \vee z_2, x_5 \vee z_3, z_1 + z_2 + z_3 = 1\}$, whose clauses are weaker and easier to satisfy. It holds that:

Proposition 2. *Let ϕ be a MaxSAT instance, and let $U = \{c_1, c_2, \ldots, c_q\}$ be a conflicting subset of soft clauses of ϕ. If the minimum number of false soft clauses in ϕ is s, then the minimum number of false soft clauses in the MaxSAT instance ϕ' obtained from ϕ after weakening U is $s - 1$.*

Proof. Let I be an optimal assignment of ϕ and let s be the number of soft clauses that I falsifies in ϕ. Note that $s > 0$ because U is conflicting. We prove that $s - 1$ is the minimum number of false soft clauses in ϕ', by distinguishing two cases:

(i) $q = 2$: In this case, $\phi' = (\phi \setminus \{c_1, c_2\}) \cup \{c_1 \vee c_2\}$. If I falsifies c_1 and c_2, then I falsifies $c_1 \vee c_2$ and $s - 2$ soft clauses of $\phi \setminus \{c_1, c_2\}$. So, I falsifies $s - 1$ soft clauses of ϕ'. If I falsifies either c_1 or c_2 (but not both), then I falsifies $s - 1$ soft clauses of $\phi \setminus \{c_1, c_2\}$ but satisfies $c_1 \vee c_2$. So, I falsifies $s - 1$ soft clauses of ϕ'.

(ii) $q > 2$: In this case, $\phi' = (\phi \setminus \{c_1, c_2, \ldots, c_q\}) \cup \{z_1 + z_2 + \cdots + z_q = 1, c_1 \vee z_1, c_2 \vee z_2, \ldots, c_q \vee z_q\}$. As $z_1 + z_2 + \cdots + z_q = 1$ is a hard constraint, any feasible assignment of ϕ' satisfies exactly one of the literals in $\{z_1, z_2, \ldots, z_q\}$. If I is extended by setting $I(z_i) = 1$ for exactly

one $i \in \{1, 2, \ldots, q\}$ such that $I(c_i) = 0$, and $I(z_j) = 0$ for each $j \in (\{1, 2, \ldots, q\} \setminus \{i\})$, then I satisfies the hard constraint and forces the satisfaction of $c_i \vee z_i$ but $I(c_j \vee z_j) = I(c_j)$ for all $j \neq i$. Thus, I falsifies $s - 1$ soft clauses of ϕ'.

Note that s is the minimum number of soft clauses that can be false in ϕ. By construction of ϕ', it has exactly one false soft clause less than ϕ in both cases. So, $s - 1$ is the minimum number of false soft clauses in ϕ'. □

For instance, in Example 3, the minimum number of false soft clauses in the conflicting subset $U = \{x_1 \vee x_4 \vee x_6, x_2 \vee x_3, x_5\}$ is 1, because all the hard clauses and the two first soft clauses are satisfied when $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 0, x_6 = 0$. However, the minimum number of false soft clauses in the weakened subset $U' = \{x_1 \vee x_4 \vee x_6 \vee z_1, x_2 \vee x_3 \vee z_2, x_5 \vee z_3, z_1 + z_2 + z_3 = 1\}$ is 0, because all the hard clauses, all the weakened soft clauses and $z_1 + z_2 + z_3 = 1$ are satisfied when $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 0, x_6 = 0, z_1 = 0, z_2 = 0, z_3 = 1$.

Since the clause weakening procedure removes one conflict from ϕ, the additional conflicts detected in ϕ' are independent of the removed conflict. If t conflicts are detected among the r soft clauses, we know that at least t soft clauses cannot be satisfied by a truth assignment of the original variables, because the t soft clauses have to be satisfied by t fresh variables (note that each fresh variable occurs exactly in one clause). As a result, the upper bound of $\omega(G)$ is improved to $r - t$. Hard clauses are not weakened because they must be satisfied by the original variables in every feasible solution of the MaxSAT instance.

4.2. Incremental MaxSAT reasoning to reduce the set of branching vertices

We apply incremental MaxSAT reasoning (Li et al., 2015) to reduce the set of branching vertices B_{d0}. After the coloring process, V is partitioned into A and B, and A is partitioned into r independent sets. Then, we define a Boolean variable v for each vertex $v \in V$, and the following partial MaxSAT encoding: there is a soft clause for each independent set resulting from the partition of A, and a hard clause $\bar{v}_i \vee \bar{v}_j$ for each pair of non-adjacent vertices v_i and v_j. For convenience, we directly treat each vertex as a Boolean variable and each independent set as a soft clause. Finally, we add a vertex of B_{d0} as a soft unit clause u, and apply unit propagation to determine whether the literal in u fails. If so, the vertex is removed from B_{d0}, and the soft clauses involved in the conflict are weakened. This process is repeated until the algorithm fails to detect a conflict or there are no more vertices in B_{d0}. If conflicts are detected for the vertices $b_{i_1}, \ldots, b_{i_k}$, then the subgraph induced by $A \cup \{b_{i_1}, \ldots, b_{i_k}\}$ does not contain any clique of size greater than r and, therefore, $b_{i_1}, \ldots, b_{i_k}$ can be removed from B_{d0} and added to A.

Function *IncMaxSAT*, defined in Algorithm 3, implements the described incremental MaxSAT reasoning approach, and returns a possibly reduced set of branching vertices. It uses a stack S to store all the unit clauses of ϕ, and applies unit propagation until an empty clause is derived or S is empty. If an empty clause is derived, the soft clauses used to derive the empty clause are collected in lines 15 to 19 by retracing the reason for each falsified literal in the empty and unit clauses. Example 3 and Table 1 illustrate how unit propagation derives an empty clause and how the clauses used to derive the conflict are collected. Then, the collected soft clauses are weakened. The constraint $z_b + z_{c_1} + \cdots + z_{c_q} = 1$ is treated as follows: if one of these fresh variables is in a unit clause, the other variables are negated (i.e. they are assigned false) and are pushed into S as unit clauses.

Let n be the number of vertices of graph G. Each vertex v occurs in exactly one soft clause but in $O(n)$ hard clauses, because the number of non-neighbors of v is in $O(n)$ and each non-neighbor u of v results in a hard clause $\bar{u} \vee \bar{v}$. So, unit resolution for one unit clause $\{l\}$ requires time $O(n)$, and unit propagation for one vertex b in B has a time complexity in $O(n^2)$; the number of unit clauses pushed into the stack S during unit propagation is in $O(n)$. The time complexity of algorithm IncMaxSAT to remove $O(n)$ vertices from B is in $O(n^3)$.

As presented in Li and Quan (2010a; 2010b), standard MaxSAT reasoning partitions the entire set of vertices V into r independent sets, and applies unit propagation and failed literal detection to the MaxSAT encoding that has those r independent sets as soft clauses. If MaxSAT reasoning detects t conflicts and the obtained upper bound is not greater than $|C_{\max}|$, the search is pruned. However, if $r - t$ is not good enough to prune the search, all the effort spent in MaxSAT reasoning is useless. Initially, incremental MaxSAT reasoning only partitions A, instead of V, and derives a MaxSAT instance ϕ that has the independent sets of A as soft clauses. Then,

Algorithm 3: IncMaxSAT(G, O_0, A, B), incremental MaxSAT reasoning to reduce the set of branching vertices B.

Input: a graph $G = (V, E)$ and an ordering O_0 over V. V is partitioned into A and B. In addition, A is partitioned into independent sets.
Output: a set of branching vertices.

```
1  begin
2      φ ← Partial MaxSAT encoding of G without including the
       soft clauses for the vertices in B;
3      while B is not empty do
4          b ← the greatest vertex in B w.r.t. O_0;
5          add soft unit clause {b} into φ and push {b} into an
           empty stack S;
6          while S is not empty and an empty clause is not derived
           do
7              pop a unit clause u from S;
8              ℓ ← the only literal in u; record u as the reason for
               the value of the variable in ℓ satisfying ℓ;
9              foreach clause c ∈ φ that contains ℓ̄ do
10                 remove ℓ̄ from c;
11                 if c becomes a unit clause then push c into S;
12                 if c becomes empty then
13                     B ← B \ {b}; /* Branching on b is not
                       necessary */
14                     /* Search for clauses inducing the conflict */
15                     push c into an empty queue Q;
16                     for each clause c' in the order they were
                       pushed in Q do
17                         foreach removed literal ℓ' of c' do
18                             if the reason r for literal ℓ' is not in Q
                               then
19                                 push r into Q;
20                     restore all removed literals into their clauses;
21                     {{b}, c_1, ..., c_q} ← the set of soft clauses in
                       Q;
22                     let z_b, z_{c_1}, ..., z_{c_q} be new variables;
23                     φ ← (φ \ {{b}, c_1, ..., c_q})
                           ∪({b ∨ z_b} ∪ {c_1 ∨ z_{c_1}, ..., c_q ∨ z_{c_q}})
                           ∪ {z_b + z_{c_1} + ··· + z_{c_q} = 1};
24                     break the foreach loop;
25         if no empty clause is derived then return B;
26     return B;
```

Algorithm 4: GetBranches$_d(G, r, O_0)$, for a BnB algorithm searching for a maximum clique containing more than r vertices in G.

Input: a graph $G = (V, E)$, an integer r, and an ordering O_0 over V
Output: a set of branching vertices

```
1 begin
2     B_d0 ← GetBranches_d0(G, r, O_0);
3     if B_d0 is empty then
4         return the empty set;
5     else
6         A_d0 ← V \ B_d0;
7         B_d ← IncMaxSAT(G, O_0, A_d0, B_d0);
8         return B_d;
```

Algorithm 5: GetBranches$_s(G, r, O_0)$ for a BnB algorithm searching for a maximum clique containing more than r vertices in G.

Input: a graph $G = (V, E)$, an integer r and an ordering O_0 over V
Output: a set of branching vertices

```
1 begin
2     B_d ← GetBranches_d(G, r, O_0);
3     if B_d is empty then
4         return the empty set;
5     else
6         v ← the greatest vertex in B_d w.r.t. the ordering O_0;
7         B_s ← {u | u∈V, u≤v w.r.t. O_0};
8         return B_s;
```

it inserts one by one the vertices of $B = V \setminus A$ as soft unit clauses into ϕ and derives a conflict for each inserted vertex. If incremental MaxSAT reasoning is able to insert all vertices of B into ϕ, the search is pruned. Otherwise, B is generally significantly reduced. Thus, incremental MaxSAT reasoning, unlike standard MaxSAT reasoning, almost always has a positive impact.

Algorithm 4 defines function GetBranches$_d$, which is a new variant of GetBranches. It first calls GetBranches$_{d0}$ and computes a set B_{d0} of branching vertices. Then, it applies incremental MaxSAT reasoning to B_{d0} and obtains a reduced set B_d of branching vertices.

Example 4. Consider the graph in Fig. 2, and assume that the vertex ordering O_0 is $v_1 < v_2 < v_3 < \cdots < v_{10}$, $|C_{\max}| = 2$ and $|C| = 0$. GetBranches$_{d0}(G, 2, O_0)$ returns $B_{d0} = \{v_2, v_3, v_4, v_5\}$, as illustrated in Example 2. Note that $D_1 = \{v_{10}, v_6, v_1\}$, $D_2 = \{v_9, v_8, v_7\}$ and $A_{d0} = D_1 \cup D_2$.

Function GetBranches$_d$ applies incremental MaxSAT reasoning to reduce B_{d0} as follows. IncMaxSAT inserts v_5 into A as a new independent set D_3=$\{v_5\}$. The adjacent vertices to v_5 in D_1 are v_{10} and v_1. However, v_8, which is the only adjacent vertex to v_5 in D_2, is not adjacent to v_{10} and v_1. So the set $\{D_1, D_2, \{v_5\}\}$ cannot form a clique of size 3 and is conflicting, which is weakened to $\{D_1$=$\{v_{10}, v_6, v_1, z_1\}$, D_2=$\{v_9, v_8, v_7, z_2\}$, D_3=$\{v_5, z_3\}\}$ together with the hard constraint $z_1 + z_2 + z_3 = 1$. Then, IncMaxSAT inserts v_4 into A as a new independent set D_4=$\{v_4\}$. Since v_5 in D_3 is not adjacent to v_4, z_3 should be 1, forcing z_1 and z_2 to be 0 because of the hard constraint. The only adjacent vertex to v_4 in D_1 is v_6, which is not adjacent to v_7, the only adjacent vertex to v_4 in D_2. So, the set $\{D_1, D_2, \{v_5, z_3\}, \{v_4\}\}$ cannot form a clique of size 4 and is conflicting. This set is weakened to $\{D_1$=$\{v_{10}, v_6, v_1, z_1, z_4\}$, D_2=$\{v_9, v_8, v_7, z_2, z_5\}$, D_3=$\{v_5, z_3, z_6\}$, D_4=$\{v_4, z_7\}\}$ together with the new hard constraint $z_4 + z_5 + z_6 + z_7 = 1$. IncMaxSAT fails to derive a conflict when inserting vertex v_3. Finally, GetBranches$_d$ returns the set B_d=$\{v_2, v_3\}$, which has two vertices fewer than B_{d0}.

In the following, when we mention algorithm DoMC, we refer to algorithm MC implementing function GetBranches$_d$.

5. SoMC: a BnB algorithm for MaxClique with a static ordering for branching

In the previous GetBranches functions, the only objective is that B should be as small as possible. Until now, we used a static vertex ordering O_0 for partitioning the set of vertices V, and minimized B as much as possible, inducing a dynamic vertex ordering between vertices in A and vertices in B. We now add a constraint: all the vertices of B should be smaller than all the vertices of A w.r.t. the ordering O_0, so that when the BnB algorithm branches on a vertex $b \in B$, it is guaranteed that b is smaller than all the vertices in the candidate set P w.r.t. O_0.

Algorithm 5 defines function GetBranches$_s$, which is based on GetBranches$_d$, and simply returns the set B_s of vertices that are smaller than or equal to the greatest vertex of B_d w.r.t. O_0. In this way, all vertices in B_s are smaller than all vertices in $A = V \setminus B_s$ w.r.t. O_0.

Example 5. Consider again Example 4, where GetBranches$_d$ returns the set $B_d = \{v_2, v_3\}$. Since the greatest vertex in B_d is v_3, GetBranches$_s$ moves the vertex v_1 of $A_d = V \setminus B_d = \{v_1, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$, which is smaller than v_3, back into B_d, and returns $B_s = \{v_1, v_2, v_3\}$. It returns the set of vertices that are smaller than or equal to v_3.

In the following, when we mention algorithm SoMC, we refer to algorithm MC implementing function GetBranches$_s$. SoMC stands for "Static ordering MaxClique solver", because the vertex ordering between B_s and $A_s = V \setminus B_s$ is always consistent with the initial ordering O_0.

In Example 5, B_s contains one more branching vertex than B_d: v_1. However, when SoMC branches on v_2 or v_3, the candidate set P does not contain v_1, as opposed to what happens in DoMC. Our first intuition for SoMC could be described as follows. Since v_1 is the smallest vertex, searching for a maximum clique in $G[\Gamma(v_1)]$ probably is much easier than in $G[\Gamma(v_2)]$ and in $G[\Gamma(v_3)]$. So, it might be advantageous to branch on v_1 separately, because removing v_1 from the candidate set P might significantly reduce the complexity of branching on v_2 and v_3 and largely compensate the cost of branching on v_1.

6. Comparison of the GetBranches functions

We conducted experiments to compare the mean number of branching vertices of the three GetBranches functions when solving a subset of DIMACS instances, excluding the instances that are either too easy or too hard. We ran DoMC$_0$ to solve each DIMACS instance. At each search tree node, we computed the number of branching vertices returned by each GetBranches function. Then, DoMC$_0$ effectively branched on every vertex in the set returned by GetBranches$_{d0}$. Finally, the mean number of branching vertices was computed for every GetBranches function by dividing the total number of branching vertices in the search tree by the search tree size.

Table 2 shows the mean number of branching vertices computed by each GetBranches function. The comparison is meaningful because the three functions are called with the same graph and the same r, as well as the same vertex ordering O_0, at every search tree node. DoMC$_0$ was chosen because GetBranches$_{d0}$

Table 2
Mean number of branching vertices of three GetBranches functions over a set of DIMACS graphs, excluding the instances that are either too easy or too hard.

Instance	GetBranches$_{d0}$	GetBranches$_{d}$	GetBranches$_{s}$
brock400_1	3.54	0.61	0.86
brock400_2	3.44	0.55	0.80
brock400_3	3.22	0.52	0.89
brock400_4	3.38	0.55	0.95
C250.9	2.48	0.09	0.16
DSJC1000_5	5.64	2.59	4.30
gen200_p0.9_55	1.73	0.98	1.08
gen400_p0.9_55	1.90	<0.01	<0.01
gen400_p0.9_65	1.71	0.02	0.02
gen400_p0.9_75	1.87	0.23	0.24
hamming10-2	1.00	1.00	1.01
keller5	3.96	0.21	0.26
MANN_a27	1.47	1.10	10.14
MANN_a45	2.29	1.15	1.24
p_hat1000-2	4.40	0.53	1.38
p_hat1500-1	8.23	3.91	7.65
p_hat300-3	4.04	0.46	1.00
p_hat500-2	3.73	0.98	2.34
p_hat500-3	3.98	0.39	1.01
p_hat700-2	4.69	0.69	1.68
p_hat700-3	4.31	0.29	0.77
san400_0.7_3	3.06	0.18	0.21
sanr200_0.9	2.53	0.10	0.19
sanr400_0.7	3.50	0.71	1.03

presumably computes the largest set of branching vertices. Thus, the three GetBranches functions compute the number of branching vertices for the maximum number of subgraphs: each branching vertex in DoMC$_0$ corresponds to a subgraph, and the three GetBranches functions are called at each subgraph.

As expected, GetBranches$_{d0}$ computes the largest set of branching vertices, providing evidence of the positive impact of incremental MaxSAT reasoning. For GetBranches$_d$ and GetBranches$_s$, the mean number of branching vertices is often smaller than one, because they return an empty set of branching vertices at many search tree nodes where GetBranches$_{d0}$ returns a nonempty set of branching vertices. GetBranches$_s$ computes more branching vertices than GetBranches$_d$ but preserves the static vertex ordering O_0 in the graph. Experimental results reported in Li et al. (2015) indicate that SoMC clearly dominates DoMC.

7. Improved algorithms: DoMC2$_0$, DoMC2 and SoMC2

A weak point of DoMC$_0$, DoMC and SoMC is that they do not exploit sufficiently the results of the previous search. To overcome this lack of incrementality we incorporated into them a more efficient representation of the adjacency matrix, and an incremental upper bound computation method. As a result, we developed the new algorithms DoMC2$_0$, DoMC2 and SoMC2 that outperform DoMC$_0$, DoMC and SoMC, respectively.

In this section we first describe how to improve the representation of the adjacency matrix, and the new incremental upper bound. Then, we define a new generic algorithm, MC2, that implements the previously mentioned improvements. Finally, we explain how to derive DoMC2$_0$, DoMC2 and SoMC2 from MC2.

7.1. Reconstruction of adjacency matrix

Algorithm MC stores the graph $G = (V, E)$ in an adjacency matrix M of size $|V| \times |V|$ where, for each row i and each column j, $M[i, j] = 1$ if and only if vertex v_i and vertex v_j are adjacent. Note that M is usually huge. For example, if G has 1000 vertices, then M usually needs 1,000,000 memory bytes. In general, M is supposed not to be loaded entirely into the cache memory.

Ideally, consecutive rows and columns of M correspond to consecutive vertices of G w.r.t. a fixed ordering. If the branching vertices in B are all smaller than the vertices in A, the sub-matrix of M storing the subgraph induced by A grows incrementally along with successive branchings on $b_{|B|}$, $b_{|B|-1}$, ..., b_1 in B, providing efficient cache memory utilization. So, static orderings for branching lead to an incremental increase of the size of the matrix and provide better cache memory utilization. However, using the dynamic ordering of DoMC, A can contain vertices that are smaller (and greater) than the branching vertices of B. In this case, the rows and columns of the sub-matrix corresponding to vertices in A can spread widely in M, producing frequent cache failures.

Example 6. Refer to Example 5, where $A_s = \{v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$ and $B_s = \{v_1, v_2, v_3\}$. The sub-matrix storing the subgraph induced by A_s consists of consecutive rows and columns of M, and has more chance to be loaded into the cache memory. Then, v_3, v_2 and v_1 are added successively to A_s, incrementally increasing the sub-matrix and favoring the cache memory utilization. However, in DoMC, the sub-matrix storing $A_d = \{v_1, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$ does not consist of consecutive rows and columns of M, and has less chance to be loaded into the cache memory for successive branchings.

To overcome this drawback in DoMC, we adapted an optimization technique from MCS that reconstructs the adjacency matrix, so that vertices in A correspond to consecutive rows and columns in M. However, the reconstruction is time-consuming and is thus limited to the root in MCS. We extended the reconstruction to the direct child nodes of the root of the search tree (i.e., when $|C| \leq 1$ in Algorithm MC). Our tests show that the reconstruction limited to such nodes makes DoMC about 5% to 10% faster, and that applying the reconstruction to additional search tree nodes does not improve the performance.

Observe that although the reconstruction of the adjacency matrix is mainly introduced for DoMC, it is also useful for SoMC, because after branching on b_i in SoMC, the vertices in the candidate set $P = \Gamma(b_i) \cap (\{b_{i+1}, \ldots, b_{|B|}\} \cup A)$ are generally not consecutive in matrix M (though less widely spread than in DoMC). This reconstruction makes these vertices consecutive in the matrix and improves the effective utilization of the cache memory.

7.2. Incremental upper bound

In order to exploit the results obtained in previous search, we use an array called vertexUB$[v_i, O]$, as in Li et al. (2013), to store an upper bound on the size of a maximum clique containing v_i in the subgraph induced by v_i and all the vertices greater than v_i w.r.t. the ordering O. Formally, let $v_1 < v_2 < v_3 < \cdots < v_n$ be a vertex ordering O over the vertices of graph $G = (V, E)$, and let $V_i = \{v_i, v_{i+1}, \ldots, v_n\}$ for $i = 1, \ldots, n$ (i.e., V_i contains v_i and all vertices greater than v_i w.r.t. O). We store in vertexUB$[v_i, O]$ an upper bound on the size of a maximum clique containing v_i in $G[V_i]$.

As shown in Li et al. (2013), vertexUB$[v_i, O]$ is very effective in pruning the search. In fact, when a BnB algorithm branches on v_i, its purpose is to search for a maximum clique containing v_i in $G[V_i]$. However, if the largest clique found so far in G has size s and vertexUB$[v_i, O] \leq s$, then the branch can be pruned.

The following rules are introduced in Li et al. (2013) to define vertexUB$[v_i, O]$.

- **Inheritance rule:** If vertexUB$[v_i, O]$ is defined on a set U_i and $V_i \subseteq U_i$, then vertexUB$[v_i, O]$ can be defined on V_i with the same value as in U_i.
- **Incremental rule:** We define a function called IncUB(v_i, O) as follows. If $V_i \cap \Gamma(v_i) = \emptyset$, then IncUB$(v_i, O) = 1$. Otherwise, IncUB$(v_i, O) = 1 + \max_{u \in V_i \cap \Gamma(v_i)} vertexUB[u, O]$, provided that

vertexUB[u, O] was already defined for each $u \in V_i \cap \Gamma(v_i)$ in the previous search. Obviously, vertexUB[v_i, O] can be defined to be IncUB(v_i, O). In other words, vertexUB[v_i, O] can be defined to be the maximum vertexUB of its neighbors in V_i plus 1, because any clique containing v_i is formed by v_i and some of its neighbors.

Coloring rule: If V_i can be partitioned into r independent sets, then vertexUB[v_i, O] can be defined to be r.

MaxSAT rule: If V_i can be partitioned into r independent sets and t disjoint conflicting subsets of independent sets can be detected using MaxSAT reasoning, then vertexUB[v_i, O] can be defined to be $r - t$.

At the beginning of the search, IncMaxCLQ (Li et al., 2013) initializes vertexUB[v, O] for each v in G using the incremental rule. The inheritance rule is used to initialize vertexUB[v, O] for each v in a subgraph of G. Before branching on v_i, IncMaxCLQ applies the incremental rule, the coloring rule and the MaxSAT rule. Let UB_{inc}, UB_{color} and UB_{MaxSAT} be the upper bounds derived from the incremental rule, the coloring rule and the MaxSAT rule, respectively, and let $\beta = \min(\text{UB}_{inc}, \text{UB}_{color}, \text{UB}_{MaxSAT})$. IncMaxCLQ improves vertexUB[v_i, O] to be $\min(\beta$, vertexUB[v_i, O]). If the improved vertexUB[v_i, O] does not allow IncMaxCLQ to prune the branch, IncMaxCLQ branches on v_i, with a growing clique C and the largest clique $C_{\max}$ found so far, to obtain a clique C_1 which is either a maximum clique containing v_i in $G[V_i]$ such that $|C_1| > |C_{\max}|$ or $C_{\max}$ if such a clique does not exist. In both cases, IncMaxCLQ applies the following rule to improve vertexUB[v_i, O].

Branching rule: vertexUB[v_i, O] can be defined to be $|C_1| - |C|$ after a BnB algorithm branches on v_i with the growing clique C to obtain a clique C_1.

Example 7. Let O be the ordering $v_1 < v_2 < v_3 < \cdots < v_{10}$ for the vertices in the graph G of Fig. 2. Using the incremental rule, vertexUB[v_i, O] ($1 \le i \le 10$) can be initialized as follows:

- vertexUB[v_{10}, O] = 1, because $V_{10} \cap \Gamma(v_{10}) = \emptyset$;
- vertexUB[v_9, O] = 2, initialized with 1+vertexUB[v_{10}, O];
- vertexUB[v_8, O] = 1, because $V_8 \cap \Gamma(v_8) = \emptyset$;
- vertexUB[v_7, O] = 2, initialized with 1+vertexUB[v_{10}, O];
- vertexUB[v_6, O] = 3, initialized with 1+vertexUB[v_9, O];
- vertexUB[v_5, O] = 2, initialized with 1+vertexUB[v_{10}, O];
- vertexUB[v_4, O] = 4, initialized with 1+vertexUB[v_6, O];
- vertexUB[v_3, O] = 5, initialized with 1+vertexUB[v_4, O];
- vertexUB[v_2, O] = 3, initialized with 1+vertexUB[v_7, O];
- vertexUB[v_1, O] = 4, initialized with 1+vertexUB[v_2, O].

At the beginning, the growing clique C is empty. Before branching on v_i, IncMaxCLQ improves vertexUB[v_i, O] using the incremental rule, the coloring rule and the MaxSAT rule. If the minimum value obtained is not sufficient to prune the branch, IncMaxCLQ branches on v_i and applies the branching rule. For instance, the branching on v_3 gives the clique $\{v_3, v_6, v_9\}$. Thus $|C_{\max}| = 3$ and vertexUB[v_3, O] = 3. Before branching on v_2, IncMaxCLQ checks vertexUB[v_2, O] = $3 = |C_{\max}| - |C|$ and prunes the branch, because vertexUB[v_2, O] = 3 means that a clique containing v_2 and larger than 3 does not exist in $V_2 = \{v_2, v_3, \ldots, v_{10}\}$.

Note that vertexUB[v_i, O] depends on the ordering O. Let O' be $v_{10} < v_9 < v_8 < \cdots < v_1$ in G and $V_i' = \{v_i, v_{i-1}, \ldots, v_1\}$. Using the incremental rule, we can define vertexUB[v_1, O'] = 1 ($V_1' \cap \Gamma(v_1) = \emptyset$), vertexUB[$v_2, O'$] = 2 (1+vertexUB[$v_1, O'$]), etc. These values are different from those obtained for O, meaning that a vertexUB value obtained from an ordering cannot generally be used to prune the search in another ordering.

Nevertheless, given two orderings O and O', if v_i is smaller than or equal to all the vertices in V_i w.r.t. both O and O', we can define vertexUB[v_i, O'] = vertexUB[v_i, O]. Formally, we have:

Compatibility rule: Let O and O' be two different vertex orderings. Let v_i be a vertex such that vertexUB[v_i, O] has been defined. If it holds that $\{u \mid v_i < u \text{ w.r.t. } O'\} \subseteq \{u \mid v_i < u \text{ w.r.t. } O\}$, then vertexUB[$v_i, O'$] can be defined to be vertexUB[v_i, O].

In other words, vertexUB[v_i, O] can be used to prune the search in the ordering O' if O and O' are compatible at v_i. The compatibility rule is new in this paper. The inheritance rule is a special case of the compatibility rule when $O = O'$.

In this paper, we do not use the coloring rule and the MaxSAT rule to define vertexUB, because the GetBranches function already uses graph coloring and MaxSAT reasoning to partition a candidate set into A and B. We only use the compatibility rule, the incremental rule and the branching rule to define vertexUB, as described in the next subsection.

7.3. A new generic BnB algorithm

Algorithm 6 describes the integration of the adjacency matrix reconstruction and the incremental upper bound in Algorithm MC, giving MC2. MC2 uses the array vertexUB to store, for a vertex ordering O and for each vertex v, an upper bound on the size of a maximum clique containing v in the set $\{u \in P \mid u \ge v \text{ w.r.t. } O\}$, and calls function GetBranches to partition P into A and B according to the initial vertex ordering O_0 (line 3 and line 5), so that A cannot form a clique of size greater than $|C_{\max}| - |C|$. Note that O_0 is static and is never changed.

If B is not empty, let $A = \{a_1, a_2, \ldots, a_{|A|}\}$ and $B = \{b_1, b_2, \ldots, b_{|B|}\}$ both in increasing order w.r.t. O. MC2 defines a new vertex ordering O': $b_1 < b_2 < \cdots < b_{|B|} < a_1 < a_2 < \cdots < a_{|A|}$. O' may be different from O because there may exist two vertices a_i and b_j such that $a_i < b_j$ w.r.t. O. So, MC2 should define vertexUB[v, O'] for each vertex in P, because it will branch on vertices of B w.r.t. O'.

For each vertex a_i in A, since $a_1 < a_2 < \cdots < a_{|A|}$ w.r.t. O' is consistent with O and $A \subseteq P$, vertexUB[a_i, O'] can be defined to be vertexUB[a_i, O] using the compatibility rule. In addition, vertexUB[a_i, O'] can also be defined using the incremental rule. Moreover, the GetBranches function has proved that a maximum clique in A cannot contain more than $|C_{\max}| - |C|$ vertices. So, vertexUB[a_i, O'] should not be greater than $|C_{\max}| - |C|$. Therefore, vertexUB[a_i, O'] = $\min$(vertexUB[a_i, O], $|C_{\max}| - |C|$, IncUB(a_i, O')) (line 10).

For each vertex b_i in B, vertexUB[b_i, O'] is first initialized with the incremental rule. If vertexUB[b_i, O'] $\le |C_{\max}| - |C|$, the branching on b_i is pruned. Otherwise, if b_i is smaller than all vertices in $\{b_{i+1}, \ldots, b_{|B|}\} \cup A$ w.r.t. O and vertexUB[b_i, O] $\le |C_{\max}| - |C|$, the branching on b_i is also pruned and vertexUB[b_i, O'] is improved to vertexUB[b_i, O] using the compatibility rule (line 19). Otherwise, MC2 has to branch on b_i to compute a new clique C_1 which is either $C_{\max}$ or a larger clique (line 22). As a result, vertexUB[b_i, O'] is defined to be $|C_1| - |C|$ (line 23) using the branching rule.

When MC2 branches on b_i to search for a maximum clique in $P' = \Gamma(b_i) \cap (\{b_{i+1}, \ldots, b_{|B|}\} \cup A)$, it uses the new ordering O'. Note that vertexUB[v, O'] was already tightly defined for every vertex v in $\{b_{i+1}, \ldots, b_{|B|}\} \cup A$ and can be used to prune the search in the recursive call w.r.t. O'.

For finding a maximum clique in a graph $G = (V, E)$, the initial call of Algorithm 6 should be MC2(G, V, O_0, $\emptyset$, $\emptyset$, O_0). Before the initial call, using the incremental rule, vertexUB[v_i, O] is initialized to 1 if $V_i \cap \Gamma(v_i) = \emptyset$, and to $1 + \max_{u \in V_i \cap \Gamma(v_i)} vertexUB[u, O]$ otherwise.

Algorithm 6: MC2(G, P, O_0, C, $C_{\max}$, O), a new generic BnB algorithm for MaxClique.

Input: $G = (V, E)$, a candidate set P, an initial vertex ordering O_0 over G, the current growing clique C, the largest clique $C_{\max}$ found so far in G, another vertex ordering O in P
Output: $C \cup C'$, where C' is a maximum clique of $G[P]$, if $|C \cup C'| > |C_{\max}|$; $C_{\max}$ otherwise

```
1  begin
2      if P = ∅ then return C;
3      B ← GetBranches(G[P], |C_max| − |C|, O_0);
4      if B = ∅ then return C_max;
5      A ← P \ B;
6      Let A = {a_1, a_2, ..., a_|A|} and B = {b_1, b_2, ..., b_|B|} in the
       increasing ordering w.r.t. O;
7      Define the new vertex ordering O' in P:
       b_1 < b_2 < ··· < b_|B| < a_1 < a_2 < ··· < a_|A|;
8      for i := |A| downto 1 do
9          /* compatibility rule and incremental rule
           */
10         vertexUB[a_i, O'] ← min(vertexUB[a_i, O], |C_max| − |C|,
11                                 IncUB(a_i, O'));
12     if |C| ≤ 1 then
13         reconstruct the adjacency matrix to make the vertices
           of G[P] consecutive in the matrix w.r.t. the ordering O';
14     for i := |B| downto 1 do
15         vertexUB[b_i, O'] ← IncUB(b_i, O'); /* incremental
           rule */
16         if vertexUB[b_i, O'] > |C_max| − |C| then
17             if b_i is smaller than all vertices in {b_{i+1}, ..., b_|B|} ∪ A
               w.r.t. O and vertexUB[b_i, O] ≤ |C_max| − |C| then
18                 /* compatibility rule */
19                 vertexUB[b_i, O'] ← vertexUB[b_i, O];
20             else
21                 P' ← Γ(b_i) ∩ ({b_{i+1}, ..., b_|B|} ∪ A);
22                 C_1 ← MC2(G, P', O_0, C ∪ {b_i}, C_max, O');
23                 vertexUB[b_i, O'] ← |C_1| − |C|; /* branching
                   rule */
24                 if |C_1| > |C_max| then C_max ← C_1;
25     return C_max;
```

7.4. Algorithms DoMC2$_0$, DoMC2 and SoMC2

We obtain the new BnB algorithm DoMC2$_0$ (resp. DoMC2 and SoMC2) if function GetBranches is replaced with GetBranches$_{d0}$ (resp. GetBranches$_d$ and GetBranches$_s$) in MC2. Observe that in SoMC2, the vertex orderings O and O' are always identical to O_0, which is not the case in DoMC2$_0$ and DoMC2.

Table 3 compares the performance of algorithms DoMC2$_0$, DoMC2 and SoMC2 on the instances of Table 2. We can see that MaxSAT reasoning makes DoMC2 and SoMC2 significantly better than DoMC2$_0$ in terms of both runtime and search tree size, and that DoMC2 is superior to SoMC2 on some instances whereas SoMC2 is superior to DoMC2 on other instances. So, the results indicate that DoMC2 and SoMC2 are complementary.

8. Algorithm MoMC: combining the dynamic and static strategies

SoMC2 and DoMC2 are both based on function GetBranches$_d$, although SoMC2 re-inserts some vertices back into the set of branching vertices to keep the ordering between the branching and non-branching vertices. Since SoMC2 and DoMC2 are complementary, we propose a new function in Algorithm 7, called GetBranches$_m$, that combines the performance of SoMC2 and DoMC2.

Table 3
Comparison of three new solvers in terms of search tree sizes in thousands and runtimes in seconds on DIMACS graphs, excluding too easy and too hard instances.

Instance	DoMC2$_0$		DoMC2		SoMC2	
	Time	Tree	Time	Tree	Time	Tree
brock400_1	333.1	26,829	193.1	1801	**128.5**	**1097**
brock400_2	886.2	79,641	492.0	4605	**100.1**	**907.8**
brock400_3	254.0	21,329	141.6	1289	**76.13**	**745.2**
brock400_4	142.2	9472	74.42	564.2	**20.87**	**202.2**
C250.9	1038	77,526	**143.9**	**1035**	193.9	1060
DSJC1000_5	143.1	8934	155.7	1866	**130.8**	**1389**
gen200_p0.9_55	**0.04**	2.77	**0.04**	**1.48**	**0.04**	**1.48**
gen400_p0.9_55	130.5	9699	**1.09**	**1.83**	**1.09**	**1.83**
gen400_p0.9_65	2.71	321.0	**0.32**	**2.15**	**0.32**	**2.15**
gen400_p0.9_75	0.48	29.06	**0.21**	**2.78**	**0.21**	**2.78**
hamming10-2	**7.90**	**130.8**	35.43	**130.8**	34.88	**130.8**
keller5	1392	40,720	253.2	848.4	**181.3**	**745.7**
MANN_a27	0.37	12.60	**0.18**	**8.20**	0.25	8.57
MANN_a45	75.05	238.6	**7.84**	**62.88**	10.60	63.15
p_hat1000-2	52.22	1402	**19.69**	47.72	26.26	**46.15**
p_hat1500-1	**2.65**	139.7	3.32	45.58	3.42	**42.91**
p_hat300-3	0.53	19.13	**0.28**	1.28	0.32	**1.11**
p_hat500-2	0.29	3.54	**0.28**	**0.82**	**0.28**	**0.82**
p_hat500-3	33.71	949.1	**9.98**	**25.65**	14.45	26.78
p_hat700-2	1.48	25.56	**0.92**	1.56	1.08	**1.45**
p_hat700-3	498.6	9352	**113.7**	198.0	157.1	**171.0**
san400_0.7_3	0.25	11.07	**0.20**	**0.25**	**0.20**	**0.25**
sanr200_0.9	12.40	946.9	1.93	15.29	**1.90**	**11.75**
sanr400_0.7	207.0	20,894	148.4	1660	**63.14**	**568.1**

Algorithm 7: GetBranches$_m$(G, r, O_0), for a BnB algorithm searching for a maximum clique containing more than r vertices in G.

Input: a graph $G = (V, E)$, an integer r and a vertex ordering O_0
Output: a set of branching vertices

```
1  begin
2      B_d ← GetBranches_d(G, r, O_0);
3      if B_d is empty then
4          return B_d;
5      else
6          v ← the greatest vertex in B_d w.r.t. O_0;
7          B_s ← {u | u ∈ V, u ≤ v w.r.t. O_0};
8          if |B_d|/|B_s| < α then
9              return B_d;
10         else
11             return B_s;
```

Using GetBranches$_m$ in Algorithm 6, we obtain a BnB algorithm called MoMC (for Mixed ordering MaxClique solver). GetBranches$_m$ uses a parameter α ($0 \le \alpha \le 1$) to control the choice of either GetBranches$_d$ or GetBranches$_s$. If $\alpha = 0$, GetBranches$_m$ is simply GetBranches$_s$; and if $\alpha = 1$, GetBranches$_m$ is GetBranches$_d$. When $0 < \alpha < 1$, GetBranches$_m$ dynamically chooses either B_d or B_s according to the ratio $|B_d|/|B_s|$. If B_d is much smaller than B_s (i.e., if $|B_d| < \alpha|B_s|$), B_d is returned, inducing a new vertex ordering over the vertices of G. Otherwise, B_s is returned and the branching vertices are all smaller than the non-branching vertices w.r.t. the initial vertex ordering O_0 (O' is simply O_0 in this case in MoMC). Our

experiments show that the solver MoMC achieves the best overall results when $\alpha = 0.6$.

Observe that GetBranches$_d$ is the most time-consuming computation of GetBranches$_m$. The overhead of the other lines is negligible.

9. Experimental analysis

We conducted experiments to evaluate the algorithms DoMC2$_0$, DoMC2, SoMC2 and MoMC, and compare them with several state-of-the-art exact algorithms. DoMC2$_0$, DoMC2, SoMC2 and MoMC were implemented in C and compiled using GNU gcc −O3. Vertices are represented by integers, and we used standard integer arrays in the implementation of the algorithms, except for the adjacency matrix which is stored in a 2-dimension array. For example, we use integer arrays to store the list of neighbors and the list of non-neighbors of each vertex, the independent sets, vertexUB, the stack S for unit propagation, the queue Q in the IncMaxSAT algorithm, and the stacks to control backtracking. Recall that hard clauses and soft clauses in the MaxSAT encoding of a MaxClique instance are respectively represented by the lists of non-neighbors and the independent sets. This implementation might be improved using special data structures like bit strings as in BBMC (Segundo et al., 2011).

To search for a maximum clique in a graph G, all these algorithms begin by automatically determining the initial vertex ordering O_0, which is either the degeneracy ordering or the MIS ordering, in a preprocessing step. The MIS ordering is computed in the same manner as IncMaxCLQ does (see Section 3). The time spent for the preprocessing is included in the whole runtime of each algorithm. Since the performance of DoMC2$_0$ is always worse than the performance of DoMC2 and SoMC2, we do not report its results in this section.

The experiments were performed on an Intel Xeon CPU X5460@3.16GHz under Linux with 16 GB of memory and 12 MB of cache memory (shared by 8 running cores). The *dfmax* program (compiled using GNU gcc -O3) needs 0.18, 1.04 and 3.95 s to solve the DIMACS benchmark graphs r300.5, r400.5 and r500.5, respectively, in our machine.

In the rest of this section, we first describe the benchmarks used and the solvers that we compare, and then discuss and analyze the experimental results.

9.1. Benchmarks

We solved random graphs, DIMACS graphs[1] and BHOSLIB graphs[2] to evaluate our solvers, which are essential benchmarks used in Prosser (2012) to study MaxClique solvers. DIMACS graphs and random graphs are standard benchmarks used to evaluate state-of-the-art exact MaxClique solvers (Carraghan and Pardalos, 1990; Fahle, 2002; Konc and Janežič, 2007; Li and Quan, 2010a; 2010b; Östergård, 2002; Régin, 2003; Segundo et al., 2011; Tomita and Kameda, 2007; Tomita and Seki, 2003; Tomita et al., 2010).

DIMACS graphs fall into three categories:

1. Graphs generated using randomness and some properties, including `p_hat`*, graphs with wide node degree range; `genn_p`*, graphs with a unique known maximum clique; `brock`*, graphs with a unique known maximum clique hidden in vertices of low degrees; `C`* and `DSJC`*, uniform random graphs; `san`* and `sanr`*, graphs randomly generated using different methods.
2. Crafted graphs with special structures, including: `MANN`*, Steiner triple graphs; `keller`*, generated by Keller's conjecture on tiling graphs (Lagarias and Shor, 1992); and `johnson`$l-w-d$, generated using binary vectors of length l and weight w, where two vertices are adjacent if the Hamming distance between them is at least d. The crafted graphs are often too easy (all the solvers solved `keller4`, `johnson8-2-4`, `johnson8-4-4`, and `johnson16-2-4` within 10 s), or too hard (no exact solver can solve `johnson32-2-4` and `keller6` within a reasonable time to the best of our knowledge).
3. Graphs from real world problems, including: `hamming`$t-d$ from coding theory, where two t-bit words are adjacent if they are at least at Hamming distance d from each other; `c-fat`*, graphs from distributed fault diagnosis (Berman and Pelc, 1990). Unfortunately, all these graphs are too easy except for `hamming10-4`. In the latter case, no exact solver can solve it within a reasonable time.

The BHOSLIB graphs are recommended in Cai et al. (2011); Grosso et al. (2008) to evaluate MaxClique and VC (Vertex Cover) algorithms. They are encoded from CSP instances generated using the CSP model RB (Xu et al., 2007), and are harder than most DIMACS graphs.

9.2. Solvers

We compared DoMC2, SoMC2 and MoMC with the following solvers:

1. IncMaxCLQ, described in Li et al. (2013) and available at http://home.mis.u-picardie.fr/~cli/EnglishPage.html : This algorithm branches on every vertex of the graph in the ordering O_0.
2. MaxCLQ[3]: We used the improved version in Li and Quan (2010a).
3. MaxCliqueDyn[4], version dec. 2012: It improves the 2010 version, which was better than MCR (Li and Quan, 2010b).
4. MCS (Tomita et al., 2010): MCS is improved from MCR. The original codes of MCS are not open. We used a version implemented by Wang Kai, whose performance is similar to the performance of the original version (Wang, 2013).

9.3. Experimental results

Table 4 compares the mean runtimes of MaxCliqueDyn, MCS, MaxCLQ, IncMaxCLQ, DoMC2, SoMC2 and MoMC, as well as the mean search tree sizes of DoMC2, SoMC2 and MoMC, on random graphs with different numbers of vertices (n) and different densities (d). One advantage of random graphs is that they show the asymptotic behavior of the solvers. At each point (n, d), 51 graphs were generated. Each graph was generated in such a way that two vertices are adjacent with probability d. When less than 51 instances were solved within the cutoff time of 5000 s, the total number of solved graphs is shown in the next row. To save space, we do not display the results for densities that are too easy for a given n (i.e., when the mean runtime is less than 0.1 s for all the tested solvers).

DoMC2, SoMC2 and MoMC are the best performing solvers in Table 4, except for three easy points (500, 0.5), (1000, 0.3) and (2000, 0.2), which are solved faster with MCS. Although DoMC2 generally produces the smallest number of branches at a search

[1] available at http://cs.hbg.psu.edu/txn131/clique.html.

[2] available at http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm.

[3] available at http://home.mis.u-picardie.fr/~cli/EnglishPage.html.

[4] available at http://www.sicmm.org/~konc/maxclique/.

Table 4
Mean runtimes in seconds and tree sizes in thousands for random graphs with different numbers of vertices (n) and different densities (d); 51 graphs were solved at each point. The points where no graph is solved within 5000 s are marked with "> 5000"'. When less than 51 graphs are solved, the total number of solved graphs (#solved) is shown in the next row. "Dyn" stands for MaxCliqueDyn and "IncMC" stands for IncMaxCLQ.

n	d	Dyn	MCS	MaxCLQ	IncMC	DoMC2		SoMC2		MoMC	
		Time	Time	Time	Time	Time	Tree	Time	Tree	Time	Tree
200	0.70	0.28	0.22	0.24	0.22	0.16	2.23	0.16	**1.96**	**0.14**	2.03
200	0.80	5.22	2.28	1.61	1.44	0.89	11.45	0.91	**9.50**	**0.77**	10.01
200	0.90	65.34	37.61	9.55	6.56	4.28	36.41	4.64	**31.11**	**3.63**	31.19
200	0.95	34.84	23.76	1.69	0.95	0.60	5.08	0.67	4.83	**0.53**	**4.71**
300	0.60	0.63	0.54	0.78	0.67	0.51	6.22	0.48	**5.10**	**0.44**	5.20
300	0.70	7.88	6.37	6.27	5.78	3.91	49.80	3.54	**40.44**	**3.13**	42.12
300	0.80	270.0	209.6	116.3	119.7	72.07	735.3	70.40	**584.7**	**59.39**	610.0
300	0.90	> 5000	> 5000	3951	3576	3255	19,866	3153	**14915**	**2879**	18,116
	#solved	0	0	4	20		46		46		48
400	0.60	4.67	4.21	5.89	4.81	3.47	38.17	3.19	**29.80**	**2.87**	31.97
400	0.70	98.92	94.06	87.17	84.03	53.34	555.2	50.68	**446.7**	**43.94**	465.1
400	0.80	> 5000	> 5000	4175	3332	2760	23,394	2721	19,182	**2049**	**17846**
	#solved	0	0	28	21		51		51		51
500	0.50	1.81	**1.59**	2.89	2.34	1.80	18.55	1.83	**16.38**	1.65	16.56
500	0.60	26.26	23.30	30.60	27.11	17.53	228.6	16.02	**181.2**	**13.94**	186.4
500	0.70	1201	915.5	763.4	877.0	507.3	5283	490.4	**4418**	**426.6**	4577
1000	0.30	0.73	**0.67**	2.31	1.12	0.86	8.75	0.81	**8.25**	0.80	**8.25**
1000	0.40	8.54	7.59	19.42	9.24	7.17	73.67	6.30	**52.47**	**5.55**	52.69
1000	0.50	175.6	166.0	300.2	215.1	151.8	1761	135.4	**1433**	**119.0**	1447
2000	0.20	1.15	**1.08**	7.92	2.53	1.52	2.80	1.41	**2.70**	1.47	**2.70**
2000	0.30	23.98	20.59	147.0	31.92	21.33	222.4	20.49	**188.3**	**17.86**	**188.3**
2000	0.40	622.9	611.6	1910	891.6	589.5	6511	573.2	**5927**	**507.7**	6008
5000	0.10	6.81	5.88	56.81	19.55	5.44	**4.26**	**5.01**	**4.26**	5.12	**4.26**
5000	0.20	114.1	109.1	2330	422.5	97.33	779.6	95.66	**747.0**	**86.52**	**747.0**
10000	0.10	162.1	144.8	1235	269.7	78.31	10.57	**60.94**	**10.49**	64.48	**10.49**
10000	0.20	3421	3283	> 5000	4220	3161	5847	2838	**3538**	**2582**	3540
	#solved	3	3	0	1		51		51		51
15000	0.10	750.5	970.5	> 5000	1105	613.8	122.0	405.5	**116.9**	**375.4**	**116.9**
20000	0.05	341.5	251.7	> 5000	190.6	118.7	**17.66**	106.5	**17.66**	**106.2**	**17.66**

tree node, when compared with SoMC2 and MoMC, its performance is the worst among the three solvers in terms of both runtime and search tree size. MoMC combines the strengths of SoMC2 and DoMC2, and is faster than the other solvers.

Table 5 reports the results on standard DIMACS graphs. The instances that are too easy (i.e. *brock200_*, C125.9, san200_**) or too hard (i.e. *C500.9, C1000.9, C2000.9, C4000.5, keller6*) for all the algorithms were excluded. The cutoff time was set to 10^5 s, except for the two very hard instances *MANN_a81* and *p_hat1500-3*, whose time limit was 1 month. Overall, MoMC is the best performing solver in Table 5. For the 8 *brock** graphs, the performance of MoMC is comparable to the performance of SoMC2 in terms of both runtime and search tree size, and is better than the performance of DoMC2 and the other solvers. For example, MoMC and SoMC2 are almost 5 times faster than the other solvers for *brock400_4* and *brock800_3*. For *p_hat1500-2*, MoMC is 17 times faster than MCS, 25 times faster than MaxCLQ, 3.5 times faster than IncMaxCLQ, 44% faster than SoMC2, and 11% faster than DoMC2.

DoMC2 and MoMC solve the very hard instance *MANN_a81* in 19 days, and *p_hat1500-3* in 24 days and 22 days, respectively. McCreesh and Prosser (2013) solved *MANN_a81* in 31 days with an exact multi-thread algorithm with 24 threads on a 12-core hyper-threaded dual Xeon E5645, and *p_hat1500-3* in 128 days with 32 threads on a 16-core hyper-threaded dual Xeon E5-2660. DoMC2 and MoMC are, to the best of our knowledge, the first two single thread exact solvers that solved these two instances. They might be parallelized using the low-overhead, scalable work splitting mechanism proposed in McCreesh and Prosser (2015), to solve these instances in shorter time.

Table 6 compares the runtimes of all the solvers, and the search tree sizes of DoMC2, SoMC2 and MoMC, for the BHOSLIB graphs. DoMC2, SoMC2 and MoMC have comparable performance, and outperform IncMaxCLQ, which was so far the best exact solver on the BHOSLIB graphs.

Table 7 compares both the runtimes and search tree sizes of DoMC2, SoMC2 and MoMC to find an optimal solution and to prove the optimality for the instances of Tables 5 and 6, for which DoMC2, SoMC2 and MoMC are substantially faster than MCS, MaxCliqueDyn, MaxCLQ and IncMaxCLQ. The columns *finding* report the runtimes and search tree sizes of each algorithm to find an optimal solution from scratch (without proving the optimality). The columns *proving* report the runtimes and search tree sizes of each algorithm to prove the optimality after finding an optimal solution. In general, it is harder to find an optimal solution than to prove its optimality for DoMC2, SoMC2 and MoMC.

10. Conclusions

We introduced a static strategy for BnB MaxClique algorithms to minimize the number of branches at a search tree node subject to the constraint that a static vertex ordering in a graph G should be kept during the search. This static strategy was compared with the dynamic strategy intensively used in the state-of-the-art BnB MaxClique algorithms for minimizing the number of branches without any constraint. We designed the following three BnB algorithms from the same generic algorithm: DoMC2$_0$ and DoMC2 using the dynamic strategy, and SoMC2 using the static strategy. These algorithms differ only in the minimization function. DoMC2$_0$ uses a graph coloring process to minimize the number of branches whereas DoMC2 and SoMC2 use, in addition, incremental MaxSAT reasoning to improve the minimization.

The reported experimental results of DoMC2$_0$, DoMC2 and SoMC2 show that the dynamic strategy significantly reduces the number of branches when compared with the static strategy. However, the static strategy allows a more incremental search. By

Table 5
Runtimes in seconds and search tree sizes in thousands for DIMACS graphs that are solved by at least one solver in 10^5 s (except for the two very hard instances *MANN_a81* and *p_hat1500-3* whose time limit is 1 month), excluding the graphs solved by all solvers in 1 s. "–" stands for instances that cannot be solved wthin the cutoff time. "Dyn" stands for MaxCliqueDyn and "IncMC" stands for IncMaxCLQ.

Instance	n	d	Dyn	MCS	MaxCLQ	IncMC	DoMC2		SoMC2		MoMC	
			Time	Time	Time	Time	Time	Tree	Time	Tree	Time	Tree
brock400_1	400	0.74	466.0	379.7	339.2	222.3	193.1	1801	128.5	**1097**	**122.2**	1242
brock400_2	400	0.74	192.1	166.2	105.9	170.2	492.0	4605	**100.1**	**907**	102.4	1055
brock400_3	400	0.74	371.6	256.2	102.4	204.6	141.6	1289	76.13	**745.2**	**74.27**	752.4
brock400_4	400	0.74	185.7	138.2	125.3	159.2	74.42	564.2	20.87	**202.2**	**20.35**	**202.2**
brock800_1	800	0.65	5988	5209	4889	8830	5814	51,917	**1865**	**13669**	1888	**13669**
brock800_2	800	0.65	5349	4686	4857	11,210	9332	70,890	**1867**	**13903**	1913	**13903**
brock800_3	800	0.65	3455	3208	3452	4221	5765	39,014	789.6	**5913**	**756.8**	**5913**
brock800_4	800	0.65	2691	2259	3441	5832	4069	28,964	**1310**	**11103**	1331	**11103**
C2000.5	2000	0.50	–	–	–	61,009	35,480	361,270	37,416	**341639**	**33941**	348,617
C250.9	250	0.89	2376	2074	298.2	278.9	**143.9**	**1035**	193.9	1060	144.4	1039
DSJC1000.5	1000	0.50	185.2	169.6	295.8	226.3	155.7	1866	130.8	**1389**	**113.12**	1394
gen400_p0.9_55	400	0.90	–	37,220	–	1.23	**1.09**	**1.83**	**1.09**	**1.83**	**1.09**	**1.83**
gen400_p0.9_65	400	0.90	–	96,567	26,134	0.34	**0.32**	**2.15**	**0.32**	**2.15**	**0.32**	**2.15**
gen400_p0.9_75	400	0.90	–	–	1372	0.27	**0.21**	**2.78**	**0.21**	**2.78**	**0.21**	**2.78**
hamming10-2	1024	0.99	49.73	0.19	**0.06**	32.65	35.43	**130.8**	34.88	**130.8**	35.37	**130.8**
keller5	776	0.75	–	–	5376	**141.6**	253.2	848.4	181.3	745.6	178.6	**736.7**
MANN_a27	378	0.99	2.81	0.36	**0.12**	0.34	0.18	**8.20**	0.25	8.57	0.18	8.22
MANN_a45	1035	0.99	1712	63.09	20.04	115.8	**7.84**	**62.88**	10.60	63.15	10.21	63.15
MANN_a81	3321	0.99	–	–	–	–	**19 days**		–		**19 days**	
p_hat1000-2	1000	0.49	276.6	131.6	219.9	48.75	19.69	47.72	26.26	**46.15**	**19.53**	49.41
p_hat1000-3	1000	0.75	–	–	–	42,244	17,493	29,806	22,986	**27237**	**16720**	28,494
p_hat1500-1	1500	0.25	2.84	**2.38**	9.75	5.32	3.32	45.58	3.42	**42.91**	3.08	**42.91**
p_hat1500-2	1500	0.51	–	10,448	15,138	2165	674.3	1209	879.9	**1098**	**607.8**	1145
p_hat1500-3	1500	0.75	–	–	–	–	24 days		–		**22 days**	
p_hat300-3	300	0.74	3.06	1.43	1.13	0.37	**0.28**	1.28	0.32	**1.11**	0.30	1.14
p_hat500-2	500	0.50	1.04	0.42	0.66	**0.19**	0.28	**0.82**	0.28	**0.82**	0.28	**0.82**
p_hat500-3	500	0.75	235.1	79.23	81.04	22.02	**9.98**	**25.65**	14.45	26.78	10.12	27.52
p_hat700-2	700	0.49	8.60	3.33	4.61	**0.92**	**0.92**	1.56	1.08	**1.45**	0.94	1.53
p_hat700-3	700	0.75	3946	1586	1009	269.5	113.7	198.0	157.1	**171.0**	**111.2**	187.4
san400_0.7_3	400	0.70	1.50	0.77	3.1	1.91	**0.20**	**0.25**	**0.20**	**0.25**	**0.20**	**0.25**
sanr200_0.9	200	0.90	32.56	19.68	6.08	2.71	1.93	15.29	1.90	**11.75**	**1.47**	**11.75**
sanr400_0.7	400	0.70	110.0	99.69	98.56	104.7	148.4	1660	63.14	**568.1**	**54.86**	591.4

Table 6
Runtimes in seconds and tree sizes in thousands for BHOSLIB instances that are solved by at least one solver in 3600 s. The instances that cannot be solved within 3600 s are marked by "–". "Dyn" stands for MaxCliqueDyn and "IncMC" stands for IncMaxCLQ.

Instance	n	d	Dyn	MCS	MaxCLQ	IncMC	DoMC2		SoMC2		MoMC	
			Time	Time	Time	Time	Time	Tree	Time	Tree	Time	Tree
frb30-15-1	450	0.82	2500	992.8	493.6	0.13	**0.12**	**0.43**	**0.12**	**0.43**	**0.12**	**0.43**
frb30-15-2	450	0.82	–	1432	849.2	0.15	**0.14**	**0.44**	**0.14**	**0.44**	**0.14**	**0.44**
frb30-15-3	450	0.82	2741	766.5	365.5	0.21	**0.19**	**0.45**	0.21	**0.45**	**0.19**	**0.45**
frb30-15-4	450	0.82	–	2620	1058	0.21	0.21	**0.45**	0.21	**0.45**	**0.20**	**0.45**
frb30-15-5	450	0.82	–	1109	663.5	0.16	**0.15**	**0.44**	0.16	**0.44**	**0.15**	**0.44**
frb35-17-1	595	0.84	–	–	–	0.89	**0.80**	**0.71**	0.83	**0.71**	**0.80**	**0.71**
frb35-17-2	595	0.84	–	–	–	1.27	1.14	**0.75**	1.20	**0.75**	**1.13**	**0.75**
frb35-17-3	595	0.84	–	–	–	0.35	0.38	**0.61**	0.31	**0.61**	**0.30**	**0.61**
frb35-17-4	595	0.84	–	–	–	0.38	0.34	**0.61**	0.34	**0.61**	**0.33**	**0.61**
frb35-17-5	595	0.84	–	–	–	0.53	**0.47**	**0.64**	0.49	**0.64**	**0.47**	**0.64**
frb40-19-1	760	0.85	–	–	–	1.62	**1.45**	**0.86**	1.55	**0.86**	**1.45**	**0.86**
frb40-19-2	760	0.85	–	–	–	0.52	**0.44**	**0.79**	0.46	**0.79**	**0.44**	**0.79**
frb40-19-3	760	0.85	–	–	–	1.59	**1.42**	**0.90**	1.51	**0.90**	**1.42**	**0.90**
frb40-19-4	760	0.85	–	–	–	5.83	**5.32**	**1.42**	5.56	**1.42**	**5.32**	**1.42**
frb40-19-5	760	0.85	–	–	–	4.83	**4.36**	**1.32**	4.69	**1.32**	**4.36**	**1.32**
frb45-21-1	945	0.86	–	–	–	105.4	**98.0**	**12.5**	100.5	**12.5**	**98.0**	**12.5**
frb45-21-2	945	0.86	–	–	–	62.36	**62.17**	**6.81**	63.74	**6.81**	62.36	**6.81**
frb45-21-3	945	0.86	–	–	–	37.42	36.03	**4.87**	35.94	**4.87**	**34.54**	**4.87**
frb45-21-4	945	0.86	–	–	–	30.47	**27.56**	**3.58**	29.24	**3.58**	27.69	**3.58**
frb45-21-5	945	0.86	–	–	–	189.1	**176.6**	**16.6**	183.6	**16.6**	177.1	**16.6**
frb50-23-1	1150	0.87	–	–	–	694.7	638.4	**48.5**	668.3	**48.5**	**626.4**	**48.5**
frb50-23-2	1150	0.87	–	–	–	306.7	**283.8**	**20.9**	291.8	**20.9**	284.6	**20.9**
frb50-23-4	1150	0.87	–	–	–	14.76	14.87	**2.36**	14.02	**2.36**	**13.92**	**2.36**
frb50-23-5	1150	0.87	–	–	–	218.1	**203.8**	**17.7**	208.9	**17.7**	205.2	**17.7**
frb53-24-2	1272	0.88	–	–	–	176.8	**169.3**	**13.6**	172.9	**13.6**	170.3	**13.6**
frb53-24-3	1272	0.88	–	–	–	1845	**1699**	**139.1**	1754	**139.1**	1705	**139.1**
frb53-24-4	1272	0.88	–	–	–	–	**3369**	**204.9**	3459	**204.9**	3398	**204.9**
frb53-24-5	1272	0.88	–	–	–	897.1	822.3	**54.6**	837.3	**54.6**	**822.0**	**54.6**
frb56-25-2	1400	0.88	–	–	–	2359	2237	**145.4**	2278	**145.4**	**2226**	**145.4**

Table 7
Runtimes in seconds and search tree sizes in thousands of DoMC2,SoMC2 and MoMC to find an optimal solution (without proving the optimality) and to prove the optimality for instances of Tables 5 and 6, for which DoMC2, SoMC2 and MoMC are substantially faster than MCS, MaxCliqueDyn, MaxCLQ and IncMaxCLQ.

Instance	DoMC2				SoMC2				MoMC			
	finding		*proving*		*finding*		*proving*		*finding*		*proving*	
	Time	Tree	Time	Tree	Time	Tree	Time	Tree	Time	Tree	Time	Tree
brock400_1	169.3	1627	23.82	173.4	126.4	1082	2.12	14.85	118.8	1216	3.45	25.97
brock400_2	483.0	4548	9.01	56.36	100.1	907.0	<0.01	0.02	102.4	1054	0.03	0.12
brock400_3	116.1	1159	25.53	129.1	75.81	742.7	0.32	2.47	73.95	749.9	0.32	2.47
brock400_4	71.92	553.6	2.50	10.64	20.44	199.5	0.43	2.70	19.93	199.5	0.42	2.70
brock800_1	5489	49,552	324.2	2365	825.0	7055	1040	6614	882.0	6994	1006	6675
brock800_2	9331	70,890	<0.01	0.00	1813	13,589	53.57	313.7	1865	13,589	47.87	313.7
brock800_3	5764	39,014	<0.01	0.00	636.6	5042	153.0	870.3	602.6	5036	154.2	876.8
brock800_4	3113	24,357	955.7	4607	1237	10,589	72.20	513.7	1259	10,589	71.74	513.7
DSJC1000.5	31.40	429.0	124.3	1437	43.21	413.2	87.59	975.8	32.18	423.7	80.94	970.3
gen400_p0.9_55	1.08	1.83	<0.01	0.00	1.08	1.83	<0.01	0.00	1.08	1.83	<0.01	0.00
gen400_p0.9_65	0.31	2.15	<0.01	0.00	0.31	2.15	<0.01	0.00	0.31	2.15	<0.01	0.00
gen400_p0.9_75	0.20	2.78	<0.01	0.00	0.20	2.78	<0.01	0.00	0.20	2.78	<0.01	0.00
MANN_a45	4.71	60.05	3.13	2.83	3.68	60.09	6.92	3.06	4.83	60.09	5.38	3.06
keller5	6.70	3.20	246.5	845.2	42.10	82.80	139.2	662.8	32.90	73.90	145.7	662.8
p_hat1000-2	0.18	1.04	19.51	46.68	0.24	1.04	26.02	45.11	0.35	1.99	19.18	47.42
p_hat1500-2	20.10	3.00	654.2	1206	3.70	4.00	876.2	1094	1.70	1.00	606.1	1144
p_hat700-3	1.90	1.80	111.8	196.2	1.40	1.90	155.7	169.1	2.10	4.00	109.1	183.4
sanr400_0.7	24.40	285.0	124.0	1375	4.44	13.70	58.70	554.4	0.96	14.20	53.90	577.2
frb50-23-1	638.4	48.50	<0.01	0.00	668.3	48.50	<0.01	0.00	626.4	48.50	<0.01	0.00
frb50-23-2	283.8	20.90	<0.01	0.00	291.8	20.90	<0.01	0.00	284.6	20.90	<0.01	0.00
frb50-23-4	14.86	2.36	<0.01	0.00	14.01	2.36	<0.01	0.00	13.91	2.36	<0.01	0.00
frb50-23-5	203.8	17.70	<0.01	0.00	208.9	17.70	<0.01	0.00	205.2	17.70	<0.01	0.00
frb53-24-2	169.3	13.60	<0.01	0.00	172.9	13.60	<0.01	0.00	170.3	13.60	<0.01	0.00
frb53-24-3	1698	139.1	<0.01	0.00	1753	139.1	<0.01	0.00	1704	139.1	<0.01	0.00
frb53-24-4	3368	204.9	<0.01	0.00	3458	204.9	<0.01	0.00	3397	204.9	<0.01	0.00
frb53-24-5	822.3	54.60	<0.01	0.00	837.3	54.60	<0.01	0.00	822.0	54.60	<0.01	0.00
frb56-25-2	2236	145.4	<0.01	0.00	2277	145.4	<0.01	0.00	2225	145.4	<0.01	0.00

integrating adjacency matrix reconstruction and by exploiting an incremental upper bound in the solvers, we showed that the two strategies are complementary.

This observation led us to the design of a new BnB algorithm for MaxClique, called MoMC, that combines the strengths of the two strategies into a single algorithm. More precisely, let $|B_d|$ and $|B_s|$ be the number of branches obtained using the dynamic and static strategy, respectively. If $|B_d|/|B_s| < \alpha$, where α is a parameter of the algorithm, the dynamic strategy is used to minimize the number of branches. Otherwise, the static strategy is used. Our experimental results show that MoMC is generally better than the algorithms implementing a single strategy, and is substantially faster than other state-of-the-art BnB MaxClique algorithms. In particular, MoMC and DoMC2 are, to the best of our knowledge, the first two single thread exact solvers that solved the two very hard DIMACS instances *MANN_a81* and *p_hat1500-3*.

Acknowledgements

We thank specially the area editor and the reviewers for their careful reviews. Their constructive comments and suggestions greatly helped us to improve this paper.

This work was supported by National Natural Science Foundation of China (Grant No. 61472147, No. 61370184 and No. 61272014) and by the MeCS platform of the University of Picardie Jules Verne. The third author was supported by Mobility Grant PRX16/00215 of the Ministerio de Educación, Cultura y Deporte, the Generalitat de Catalunya grant AGAUR 2014-SGR-118, and the MINECO-FEDER project RASO TIN2015-71799-C2-1-P.

References

Babel, L., Tinhofer, G., 1990. A branch and bound algorithm for the maximum clique problem. Methods Models Oper. Res. 34 (3), 207–217.

Balasundaram, B., Butenko, S., Hicks, I.V., 2011. Clique relaxations in social network analysis: the maximum k-plex problem. Oper. Res. 59 (1), 133–142.

Barnes, E., Strickland, D.M., Sokol, J.S., 2005. Optimal protein structure alignment using maximum cliques. Oper. Res. 53, 389–402.

Benlic, U., Hao, J.K., 2013. Breakout local search for maximum clique problems. Comput. Oper. Res. 40 (1), 192–206.

Berman, P., Pelc, A., 1990. Distributed fault diagnosis for multiprocessor systems. In: Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing(FTCS 90). Newcastle, U.K., pp. 340–346.

Boginski, V., Butenko, S., Pardalos, P.M., 2006. Mining market data: a network approach. Comput. Oper. Res. 33 (11), 3171–3184.

Cai, S.W., Su, K.L., Abdul, S., 2011. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. Artif. Intell. 175, 1672–1696.

Carraghan, R., Pardalos, P.M., 1990. An exact algorithm for the maximum clique problem. Oper. Res. Lett. 9 (6), 375–382.

Eppstein, D., Darren, S., 2011. Listing all maximal cliques in large sparse real-world graphs. In: Proceedings of 10th International Symposium on Experimental Algorithms(SEA 2011), LNCS, 6630, pp. 364–375.

Etzion, T., Östergård, P.R.J., 1998. Greedy and heuristic algorithms for codes and colorings. IEEE Trans. Inf. Theory 44 (1), 382–388.

Fahle, T., 2002. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In: Proceedings of the 10th Annual European Symposium on Algorithms(ESA 2002), pp. 485–498.

Grosso, A., Locatelli, M., Pullan, W., 2008. Simple ingredients leading to very efficient heuristics for the maximum clique problem. J. Heuristics 14, 587–612.

Konc, J., Janežič, D., 2007. An improved branch and bound algorithm for the maximum clique problem. Commun. Math. Comput. Chem. 58, 569–590.

Lagarias, J.C., Shor, P.W., 1992. Keller's cube-packing conjecture is false in high dimensions. Bull. AMS (New Series) 27 (2), 279–283.

Li, C.M., Fang, Z.W., Xu, K., 2013. Combining maxSAT reasoning and incremental upper bound for the maximum clique problem. In: Proceedings of the IEEE 25th International Conference on Tools with Artificial Intelligence(ICTAI 2013), pp. 939–946.

Li, C.M., Jiang, H., Xu, R.C., 2015. Incremental maxSAT reasoning to reduce branches in a branch-and-bound algorithm for maxclique. In: Proceedings of Learning and Intelligent OptimizatioN Conference (LION 9), LNCS 8994, pp. 268–274.

Li, C.M., Quan, Z., 2010a. Combining graph structure exploitation and propositional reasoning for the maximum clique problem. In: Proceedings of the IEEE 22th International Conference on Tools with Artificial Intelligence(ICTAI 2010), pp. 344–351.

Li, C.M., Quan, Z., 2010b. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In: Proceedings of the 24th AAAI Conference on Artificial Intelligence(AAAI-10), pp. 128–133.

McCreesh, C., Prosser, P., 2013. Multi-threading a state-of-the-art maximum clique algorithm. Algorithms 6, 618–635.
McCreesh, C., Prosser, P., 2014. Reducing the branching in a branch and bound algorithm for the maximum clique problem. In: Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming(CP 2014), LNCS 8656, pp. 549–563.
McCreesh, C., Prosser, P., 2015. The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound. ACM Trans. Parallel Comput. 2 (1), 1–27.
Östergård, P.R.J., 2002. A fast algorithm for the maximum clique problem. Discrete Appl. Math. 120, 197–207.
Prosser, P., 2012. Exact algorithms for maximum clique: a computational study. Algorithms 2012 5, 545–587.
Pullan, W., Hoos, H.H., 2006. Dynamic local search for the maximum clique problem. J. Artif. Intel. Res. 25, 159–185.
Pullan, W., Mascia, F., Brunato, M., 2011. Cooperating local search for the maximum clique problem. J. Heuristics 17, 181–199.
Ravetti, M.G., Moscato, P., 2008. Identification of a 5-protein biomarker molecular signature for predicting alzheimer's disease. PLoS ONE 3 (9), e3111.
Rebennack, S., Reineltb, G., Pardalos, P.M., 2012. A tutorial on branch and cut algorithms for the maximum stable set problem. Int. Trans. Oper. Res. 19 (1–2), 161–199.
Régin, J.C., 2003. Using constraint programming to solve the maximum clique problem. In: Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming(CP 2003), LNCS 2833, pp. 634–648.
Segundo, P.S., Diego, R.L., Augustin, J., 2011. An exact bit-parallel algorithm for the maximum clique problem. Comput. Oper. Res. 38, 571–581.
Segundo, P.S., Tapia, C., 2014. Relaxed approximate coloring in exact maximum clique search. Comput. Oper. Res. 44, 185–192.
Sharmin, S., 2014. Practical Aspects of the Graph Parameter Boolean-width. University of Bergen, Norway Dissertation for the degree of philosophiae doctor (phd).
Tomita, E., Kameda, T., 2007. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. J. Glob. Optim. 37, 95–111.
Tomita, E., Seki, T., 2003. An efficient branch-and-bound algorithm for finding a maximum clique. In: Proceedings of the 4th International Conference on Discrete Mathematics & Theoretical Computer Science(DMTCS 2003), LNCS 2731, pp. 278–289.
Tomita, E., Sutani, Y., Higashi, T., Takahashi, S., Wakatsuki, M., 2010. A simple and faster branch-and-bound algorithm for finding maximum clique. In: Proceedings of the 4th International Workshop on Algorithms and Computation(WALCOM 2010), LNCS 5942, pp. 191–203.
Wang, K., 2013. Personal communication.
Wu, Q.H., Hao, J.K., 2015. A review on algorithms for maximum clique problems. Eur. J. Oper. Res. 242 (3), 693–709.
Xu, K., Frédéric, B., Fred, H., Christophe, L., 2007. Random constraint satisfaction: easy generation of hard (satisfiable) instances. Artif. Intell. 171, 514–534.