European Journal of Operational Research 270 (2018) 66–77

Contents lists available at ScienceDirect

European Journal of Operational Research

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

Discrete Optimization

A new upper bound for the maximum weight clique problem

Chu-Min Li[a,b], Yanli Liu[a], Hua Jiang[c,*], Felip Manyà[d], Yu Li[b]

[a] Huazhong University of Science and Technology, 1037 Luoyu Road, Wuhan 430074, China
[b] MIS, Université de Picardie Jules Verne, 33 rue Saint Leu, Amiens 80039, France
[c] College of Mathematics and Computer Science, Jianghan University, 8 Sanjiaohu Road, Wuhan 430056, China
[d] Artificial Intelligence Research Institute (IIIA, CSIC), Campus de la UAB, Bellaterra 08193, Spain

ARTICLE INFO

Article history:
Received 6 July 2017
Accepted 12 March 2018
Available online 20 March 2018

Keywords:
Combinatorial optimization
Branch and bound
Maximum weight clique problem
Upper bound
Weight cover

ABSTRACT

The maximum weight clique problem (MWCP) for a vertex-weighted graph is to find a complete subgraph in which the sum of vertex weights is maximum. The main goal of this paper is to develop an efficient branch-and-bound algorithm to solve the MWCP. As a crucial aspect of branch-and-bound MWCP algorithms is the incorporation of a tight upper bound, we first define a new upper bound for the MWCP, called UB_{WC}, that is based on a novel notion called weight cover. The idea of a weight cover is to compute a set of independent sets of the graph and define a weight function for each independent set so that the weight of each vertex of the graph is covered by such weight functions. We then propose a new branch-and-bound MWCP algorithm called WC-MWC that uses UB_{WC} to reduce the number of branches of the search space that must be traversed by incrementally constructing a weight cover for the graph. Finally, we present experimental results that show that UB_{WC} reduces the search space much more than previous upper bounds, and the new algorithm WC-MWC outperforms some of the best performing exact and heuristic MWCP algorithms on both small/medium graphs and real-world massive graphs.

1. Introduction

A *clique* C in an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, is a subset of V such that every pair of vertices in C is connected. The *Maximum Clique Problem* (MCP) for a graph G is to find a clique of maximum size in G, and its size is denoted by $\omega(G)$. Given a vertex-weighted graph $G^w = (V, E, w)$, where w is a weight function that assigns to each $v \in V$ a non-negative integer $w(v)$ called *weight*, the weight of a clique C is the total weight of the vertices in C and is denoted by $w(C)$. The *Maximum Weight Clique Problem* (MWCP) for a vertex-weighted graph G^w is to find a clique of maximum weight in G^w, and its weight is denoted by $\omega(G^w)$.

The MCP and the MWCP are NP-hard (Garey & Johnson, 1979) and have practical applications in domains as diverse as protein structure prediction (Mascia, Cilia, Brunato, & Passerini, 2010), coding theory (Zhian, Sabaei, Javan, & Tavallaie, 2013), combinatorial auctions (Wu & Hao, 2015b), computer vision (Zhang, Javed, & Shah, 2014) and genomics (Butenko & Wilhelm, 2006). Thus, a lot of effort has been devoted to develop both exact and heuristic algorithms for the MCP and the MWCP. See Wu and Hao (2015a) for a review on MCP algorithms, and see Cai and Lin (2016), Fang, Li, and Xu (2016), Jiang, Li, and Manyà (2016), Jiang, Li, and Manyà (2017), Li, Jiang, and Manyà (2017), San Segundo, Lopez, and Pardalos (2016), Wang, Cai, and Yin (2016), Zhou, Hao, and Goëffon (2017) for recent MCP and MWCP algorithms.

The best performing exact algorithms for the MCP and MWCP are usually based on the branch-and-bound (BnB) scheme and incorporate of a tight upper bound. Such upper bounds are based on the notion of *independent set* (IS) (Balas & Xue, 1991; Held, Cook, & Sewell, 2012; Konc & Janezic, 2007; Li et al., 2017; Li, Jiang, & Xu, 2015; Li & Quan, 2010; San Segundo, Matía, Rodríguez-Losada, & Hernando, 2013; Tomita, Sutani, Higashi, Takahashi, & Wakatsuki, 2010; Warren & Hicks, 2006), where an IS in a graph is a subset of vertices in which no two vertices are adjacent. In fact, all these algorithms exploit the property that any clique contains at most one vertex from an IS. Nevertheless, it is much harder to design an upper bound for the MWCP than for the MCP because it must consider the different vertex weights in the graph.

We identify two representative upper bounds for the MWCP in the literature. One is based on the notion of weighted IS cover and uses a set of weighted ISs to cover the weight of each vertex (Balas & Xue, 1991; Held et al., 2012; Warren & Hicks, 2006). It was also used to solve the MinSAT problem in Li, Zhu, Manyà, and Simon (2012). The other is based on MaxSAT reasoning (Li et al., 2017;

* Corresponding author.
E-mail addresses: chu-min.li@u-picardie.fr (C.-M. Li), yanlil2008@163.com (Y. Liu), jh_hgt@163.com (H. Jiang), felip@iiia.csic.es (F. Manyà), yu.li@u-picardie.fr (Y. Li).

https://doi.org/10.1016/j.ejor.2018.03.020
0377-2217/

Li et al., 2015; Li & Quan, 2010) and making it effective is very complex because of the NP-hardness of MaxSAT.

In this paper, we propose a new upper bound of $\omega(G^w)$ called UB_{WC} that is based on a novel notion called *weight cover*. A weight cover is defined over a set $\{D_1, D_2, \ldots, D_r\}$ of ISs of $G^w = (V, E, w)$ and associates a weight function w_i to each D_i. The function w_i assigns a non-negative weight $w_i(v)$ to each vertex v of G^w in such a way that $w(v) = \sum_{i=1}^{r} w_i(v)$. So, the weight $w(v)$ that the weight function of G^w assigns to each vertex v is covered by the weight functions associated to the ISs of a weight cover of G^w.

The effectiveness of UB_{WC} comes from the clever definition of the weight functions associated with the ISs of the weight cover. On the one hand, the notion of weight cover is different from the notion of weighted IS cover of Balas and Xue (1991) in that the ISs of a weighted IS cover do not have any associated weight function that assigns weights to individual vertices. On the other hand, making UB_{WC} effective is much easier than making an upper bound based on MaxSAT reasoning effective.

We incorporate the upper bound UB_{WC} into a generic BnB MWCP algorithm and use it to minimize the number of branches of the search space that must be traversed. As a result, we obtain a simple and efficient exact algorithm called WC-MWC. Experiments on a representative set of graphs show that UB_{WC} allows WC-MWC to reach, or even exceed, the performance of some of the best performing exact and heuristic MWCP algorithms on both small/medium graphs and real-world massive graphs. The performance of WC-MWC refutes two prevailing hypotheses in the field: (1) exact MWCP algorithms, despite proving optimality, are less adequate for large graphs than heuristic algorithms; and (2) algorithms designed for massive graphs are expected to perform worse on small graphs.

The paper is organized as follows. Section 2 introduces the preliminaries. Section 3 reviews existing MWCP upper bounds. Section 4 defines the weight cover notion and UB_{WC}. Section 5 describes the algorithm WC-MWC, explains how UB_{WC} minimizes the number of branches in WC-MWC and compares weight and weighted IS covers. Section 6 presents the empirical results. Section 7 concludes the paper.

2. Preliminaries

Let $G^w = (V, E, w)$ be a vertex-weighted undirected graph, where V is a set of n vertices $\{v_1, v_2, \ldots, v_n\}$, E is a set of m edges, and w is a weight function. The density of G^w is $2m/(n(n-1))$. Two vertices v_i and v_j of V are adjacent, or neighbours, if edge $(v_i, v_j) \in E$. The set of neighbours of a vertex v in G^w is denoted by $\Gamma(v) = \{v' | (v, v') \in E\}$. The cardinality of $\Gamma(v)$ is the degree of v. The subgraph of G^w induced by a subset V' of V, denoted by $G^w[V']$, is defined as $G^w[V'] = (V', E', w)$, where $E' = \{(v_i, v_j) \in E | v_i, v_j \in V'\}$. The maximum weight in V', $\max_{v \in V'}(w(v))$, is denoted by $w^*(V')$, and $w^*(\emptyset)$ is defined to be 0. For simplicity, we use the notation $v^{w(v)}$ to say that vertex v has weight $w(v)$.

A clique in $G^w = (V, E, w)$ is a subset C of V such that every two vertices in C are adjacent. The weight of C is $w(C) = \sum_{v \in C} w(v)$. An independent set (IS) of G^w is a subset D of V in which no two vertices are adjacent. An IS partition of G^w is a partition of V into ISs such that each vertex belongs to exactly one IS.

3. Previous upper bounds for the MWCP

A weighted IS cover of $G^w = (V, E, w)$ is a set $\{D_1, D_2, \ldots, D_r\}$ of ISs, with a weight W_i for each D_i, such that $V = D_1 \cup D_2 \cup \cdots \cup D_r$ and $\sum_{i: v \in D_i} W_i \geq w(v)$ for each $v \in V$. Since any clique of G^w contains at most one vertex from each D_i, we have that $UB_{WISC} = \sum_{i=1}^{r} W_i$ is an upper bound of $\omega(G^w)$. Note that each IS is assigned a weight but there are no weights assigned to the vertices in the ISs.

The quality of UB_{WISC} depends heavily on how the weighted IS cover is generated. In Held et al. (2012), Warren and Hicks (2006) and Li et al. (2012), it is generated by iteratively creating ISs until G^w becomes empty with the following procedure: Select a vertex v_{i_1} of minimum weight and construct a maximal IS $(v_{i_1}, v_{i_2}, \ldots, v_{i_q})$ without considering the vertex weights; assign the weight $w(v_{i_1}) = \min_{j=1}^{q}(w(v_{i_j}))$ to the IS; decrease the weight of each vertex in the IS by $\min_{j=1}^{q}(w(v_{i_j}))$; and remove the vertices with weight 0 from G^w before computing another IS. In this approach, the weight of an IS usually covers a small part of the weight that the weight function of G^w assigns to each vertex. Thus, the vertices of G^w usually need several ISs to cover their weight.

One can partition the graph G^w into a set $\Pi = \{D_1, D_2, \ldots, D_r\}$ of ISs and then use $UB_{IS} = \sum_{i=1}^{r} w^*(D_i)$ as an upper bound of $\omega(G^w)$. However, the upper bound UB_{IS} is very conservative for two reasons: (i) a maximum weight clique may not contain a vertex from each IS; and (ii) a maximum weight clique may not contain the most weighted vertex of an IS. In either case, UB_{IS} is not tight.

A subset of k ISs that cannot form a clique of size k is called a *conflicting set of ISs* (Li et al., 2017; Li et al., 2015; Li & Quan, 2010). Fang et al. (2016) improved UB_{IS} by repeatedly identifying disjoint subsets of conflicting ISs using MaxSAT reasoning. Let $\{S_1, S_2, \ldots, S_k\}$ be a subset of conflicting ISs in the set Π of ISs partitioning G^w and let $\delta = \min(w^*(S_1), \ldots, w^*(S_k))$. Fang et al. split each $S_i = \{u_1^{w(u_1)}, u_2^{w(u_2)}, \ldots, u_{|S_i|}^{w(u_{|S_i|})}\}$, where $1 \leq i \leq k$, into two ISs S_i' and S_i''. Without loss of generality, assume that $w(u_1) \geq \cdots \geq w(u_j) > \delta \geq w(u_{j+1}) \geq \cdots \geq w(u_{|S_i|})$. Then, $S_i' = \{u_1^{\delta}, \ldots, u_j^{\delta}, u_{j+1}^{w(u_{j+1})}, \ldots, u_{|S_i|}^{w(u_{|S_i|})}\}$ and $S_i'' = \{u_1^{w(u_1)-\delta}, \ldots, u_j^{w(u_j)-\delta}\}$, where the vertices with weight 0 are removed. Note that the maximum weight in S_i' is δ, and each weight $w(u_j)$ greater than δ is split into δ and $w(u_j) - \delta$ in this operation. After splitting, Fang et al. update $\Pi = (\Pi \setminus \{S_1, \ldots, S_k\}) \cup \{S_1'', \ldots, S_k''\}$ and improve UB_{IS} to $UB_{IS} - \delta$, because there is at least one ISs in $\{S_1', \ldots, S_k'\}$ that has no vertex in a clique. The next subset of conflicting ISs is identified in the updated Π to further improve UB_{IS}. This procedure is repeated until no conflicting ISs can be identified in Π. The resulting upper bound is denoted by UB_{MaxSAT}.

The main difficulty and complexity of UB_{MaxSAT} are in the identification of conflicting ISs. A complete identification is equivalent to computing $\omega(G^w)$, and thus is NP-hard. The approach in Fang et al. (2016) and Jiang et al. (2017) uses the incomplete method known as *unit clause propagation*, or *unit IS propagation*, to identify conflicting ISs.

The essential role of an upper bound in BnB algorithms for the MCP and MWCP is to reduce the number of branches of the search space that must be traversed. To this end, a BnB MCP (MWCP) algorithm usually partitions the vertices of G (G^w) into two sets, A and B, such that the upper bound of $\omega(G[A])$ ($\omega(G^w[A])$) is not greater than $|C_{max}|$ ($w(C_{max})$), where C_{max} is the best clique found so far, and the algorithm only needs to branch on the vertices of $B = V \setminus A$ to search for a clique better than C_{max}. State-of-the-art MCP algorithms, as the ones described in Konc and Janezic (2007), San Segundo and Tapia (2014) and Tomita et al. (2010), usually compute $|C_{max}|$ maximal ISs $\{D_1, D_2, \ldots, D_{|C_{max}|}\}$ and obtain $A = D_1 \cup D_2 \cup \cdots \cup D_{|C_{max}|}$ and $B = V \setminus A$, and then only need to branch on the vertices of B to search for a clique better than C_{max}. An improvement is to use incremental MaxSAT reasoning or simplified MaxSAT reasoning to insert vertices of B into A as in Jiang et al. (2016), Li et al. (2017), Li et al. (2015) and San Segundo, Nikolaev, and Batsyn (2015).

When solving the MWCP of a vertex-weighted graph G^w, it is harder to partition the vertices of G^w into A and B because of the vertex weights. As we will explain in Section 5, the approach of partitioning G^w into A and B in Held et al. (2012) and Warren and Hicks (2006), which is based on weighted IS covers, is less natural and inefficient. Fang et al. (2016) do not use UB_{MaxSAT} to partition G^w into A and B. If UB_{MaxSAT} is not sufficient for pruning, the effort spent to compute UB_{MaxSAT} is lost. To overcome this drawback, Jiang et al. (2017) do not compute UB_{MaxSAT} for the whole graph G^w, but for some selected subgraphs of G^w, by making MaxSAT reasoning incremental. This approach can partition G^w into A and B but is really sophisticated and complex.

4. UB_{WC}: a new upper bound for the MWCP

We first define and illustrate the notion of *weight cover* and then define the upper bound UB_{WC} of $\omega(G^w)$ for a vertex-weighted graph G^w. As we will see, UB_{WC} provides an easy and natural way to reduce the number of branches.

4.1. Weight cover

Definition 1 (Weight cover)**.** A weight cover of a vertex-weighted graph $G^w = (V, E, w)$ is a set $\Pi = \{(D_1, w_1), (D_2, w_2), \ldots, (D_r, w_r)\}$ such that

1. Each D_i is an IS of G^w.
2. Each w_i is a weight function that, for each $v \in V$, satisfies $w_i(v) > 0$ if $v \in D_i$ and $w_i(v) = 0$ if $v \notin D_i$.
3. The weight $w(v)$ of each $v \in V$ is preserved by the weight functions w_i, i.e., for each $v \in V$, it holds that $w(v) = \sum_{i=1}^{r} w_i(v)$.

In the sequel, each (D_i, w_i) of Π, where $D_i = \{v_1, v_2, \ldots, v_q\}$, is also represented by $D_i = \{v_1^{w_i(v_1)}, v_2^{w_i(v_2)}, \ldots, v_q^{w_i(v_q)}\}$, by assuming that $w_i(v) = 0$ for any $v \notin D_i$.

Example 1. Let $D_1 = \{v_1^3, v_3^2, v_5^3\}$, $D_2 = \{v_2^1, v_4^1, v_5^1\}$ and $D_3 = \{v_2^2, v_6^2\}$. Then, $\Pi = \{(D_1, w_1), (D_2, w_2), (D_3, w_3)\}$ is a weight cover of the graph of Fig. 1.

Proposition 1 (UB_{WC})**.** *If $\Pi = \{(D_1, w_1), (D_2, w_2), \ldots, (D_r, w_r)\}$ is a weight cover of the vertex-weighted graph $G^w = (V, E, w)$, then $UB_{WC}(G^w) = \sum_{i=1}^{r} w_i^*(D_i)$ is an upper bound of $\omega(G^w)$.*

Proof. Let C be a maximum weight clique of G^w. For each $v \in C$ and for each i, $1 \le i \le r$, it holds that $w_i(v) = 0$ if $v \notin D_i$, and $C \cap D_i$ contains at most one vertex because D_i is an IS. So, it holds that $\sum_{v \in C} w_i(v) = w_i(C \cap D_i) \le w_i^*(D_i)$. Furthermore, for any $v \in V$, it holds that $w(v) = \sum_{i=1}^{r} w_i(v)$ according to Condition 3 of Definition 1. Hence, $\omega(G^w) = \sum_{v \in C} w(v) = \sum_{v \in C} \sum_{i=1}^{r} w_i(v) = \sum_{i=1}^{r} \sum_{v \in C} w_i(v) \le \sum_{i=1}^{r} w_i^*(D_i)$. □

Example 2. Example 1 shows that $\Pi = \{D_1, D_2, D_3\} = \{\{v_1^3, v_3^2, v_5^3\}, \{v_2^1, v_4^1, v_5^1\}, \{v_2^2, v_6^2\}\}$ is a weight cover of the graph G^w in Fig. 1. By Proposition 1, $UB_{WC}(G^w) = \sum_{i=1}^{3} w_i^*(D_i) = 3 + 1 + 2 = 6$, which is the tightest upper bound of $\omega(G^w)$. However, the best upper bound from an IS partition Π' of G^w is $UB_{IS} = 8$. This happens, for example, when $\Pi' = \{\{v_1^3, v_3^2, v_5^4\}, \{v_2^3, v_6^2\}, \{v_4^1\}\}$.

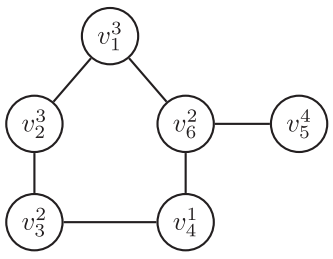

Fig. 1. A graph with $\omega(G^w) = 6$.

The main difference between a weighted IS cover of Balas and Xue (1991) and a weight cover is that no weight function is defined for the vertices of an IS in a weighted IS cover. The consequence of this difference can be stated in the following proposition.

Proposition 2 (Weight cover vs. weighted IS cover)**.** *(1) Any weighted IS cover preserving the weight of each vertex can directly be regarded as a weight cover inducing the same upper bound; (2) a weight cover cannot necessarily be regarded as a weighted IS cover preserving the weight of each vertex.*

Proof. (1) Let $\{D_1, D_2, \ldots, D_r\}$ with a weight W_i for each D_i be a weighted IS cover such that $\sum_{v \in D_i} W_i = w(v)$ for each $v \in V$. We associate a weight function w_i with each D_i such that $w_i(v) = W_i$ if $v \in D_i$ and $w_i(v) = 0$ if $v \notin D_i$. Clearly, $\{(D_1, w_1), (D_2, w_2), \ldots, (D_r, w_r)\}$ is a weight cover preserving the weight $w(v)$ of each vertex v in V and inducing the same upper bound. (2) The weight cover $\{D_1, D_2, D_3\} = \{\{v_1^3, v_3^2, v_5^3\}, \{v_2^1, v_4^1, v_5^1\}, \{v_2^2, v_6^2\}\}$ of the graph in Fig. 1 cannot be regarded as a weighted IS cover preserving the weight $w(v)$ of each vertex v. In fact, to regard the set of ISs as a weighted IS cover, one needs to assign a weight to each IS. The weight assigned to D_1 must be 3 to preserve the weight of v_1. However, the weight of v_3 is not preserved in this case because $\sum_{v_3 \in D_i} W_i = 3 > w(v_3)$. □

As Example 3 illustrates, any weighted IS cover can possibly be transformed into a weight cover with fewer ISs and/or inducing a better upper bound.

Example 3. Consider the weighted IS cover of the graph in Fig. 1: $\{D_1, D_2, D_3, D_4\} = \{\{v_1, v_3, v_5\}, \{v_2, v_5\}, \{v_2, v_6\}, \{v_1, v_4, v_5\}\}$ with $W_1 = 2$, $W_2 = 1$, $W_3 = 2$ and $W_4 = 1$, the IS D_4 can be split so that v_1 and v_5 can be inserted into D_1 and v_4 into D_2, resulting in the weight cover $\{\{v_1^3, v_3^2, v_5^3\}, \{v_2^1, v_4^1, v_5^1\}, \{v_2^2, v_6^2\}\}$ that contains fewer ISs and induces the same upper bound.

Example 3 suggests a way to obtain a weight cover from a weighted IS cover: repeatedly split an IS D and insert its vertices into other ISs without increasing the maximum weight of these ISs or such that the total increase does not exceed the weight of D. The obtained weight cover contains fewer ISs and induces the same or a possibly better upper bound. Note that fewer ISs allow to enlarge a weight cover more effectively when new vertices are added to the graph, which is particularly important for an incremental construction of the weight cover.

In the supplementary materials of this paper, we describe an algorithm for incrementally constructing a weight cover of a vertex-weighted graph that also allows us to compute UB_{WC} for that graph. We compare UB_{WC} with UB_{WISC}, based on weighted IS covers, and UB_{MaxSAT}, based on MaxSAT reasoning. The empirical results show that UB_{WC} is substantially tighter than UB_{WISC} and UB_{MaxSAT}.

The main role of an upper bound in BnB algorithms for the MCP and MWCP is to reduce the search space. In the next section, we describe how to use UB_{WC} to minimize the number of branches that must be traversed in a MWCP algorithm.

5. Minimizing the number of branches in a BnB algorithm with UB_{WC}

We present an efficient BnB MWCP algorithm called *WC-MWC* that uses UB_{WC} to reduce the search space. We first present the basic BnB search procedure that WC-MWC implements and then

describe how to use UB_{WC} to minimize the number of branches in the BnB search procedure. Finally, we define WC-MWC, which combines an efficient preprocessing and the BnB search procedure.

5.1. The basic BnB search procedure

Algorithm 1 presents the basic BnB search procedure that WC-MWC implements. Given a vertex-weighted graph $G^w = (V, E, w)$, a vertex ordering O over V, a growing clique C, a candidate set of vertices P to grow C in which every vertex is adjacent to every vertex in C, and the clique C^* of the greatest weight found so far in G^w, the algorithm calls the *Partition* function to partition P into A and B in such a way that $\omega(G^w[A]) \leq w(C^*) - w(C)$ and $B = V \setminus A = \{b_1, b_2, \ldots, b_{|B|}\}$ is the set of branching vertices. If B is empty, the current search is pruned. In this case, the clique C cannot grow to a clique of weight greater than $w(C^*)$ with the vertices of P. Otherwise, assuming that $b_1 < b_2 < \cdots < b_{|B|}$ w.r.t. O, the algorithm recursively branches on b_i, for $i = |B|, |B|-1, \ldots, 1$, to search for a clique containing b_i of weight greater than $w(C^*)$ in $G^w[\Gamma(b_i) \cap (\{b_{i+1}, \ldots, b_{|B|}\} \cup A)]$. If the initial call to the algorithm is SearchMWC$(G^w, V, O, \emptyset, \emptyset)$, it returns a maximum weight clique of G^w after exploring the whole search space.

Algorithm 1: SearchMWC(G^w, P, O, C, C^*), a generic MWCP algorithm.

Input: $G^w = (V, E, w)$, a candidate set P, an ordering O over P, a growing clique C, and the greatest weight clique C^* found so far in G^w.
Output: $C \cup C'$, where C' is a maximum weight clique of $G^w[P]$, if $w(C \cup C') > w(C^*)$; otherwise, C^*.

```
1  begin
2      if P = ∅ then
3          return C;
4      (A, B) ← Partition(G^w[P], w(C*) − w(C), O);
5      if B = ∅ then
6          return C*;
7      Let B = {b_1, b_2, ..., b_|B|} and b_1 < b_2 < ··· < b_|B| w.r.t. O;
8      for i := |B| downto 1 do
9          P' ← Γ(b_i) ∩ ({b_{i+1}, ..., b_|B|} ∪ A);
10         C_1 ← SearchMWC(G^w, P', O, C ∪ {b_i}, C*) ;
11         if w(C_1) > w(C*) then
12             C* ← C_1; C' ← C_1 \ C;
13     return C*;
```

The crucial component of Algorithm 1 is the *Partition* function. The smaller the cardinality of the set of branching vertices B returned by the function, the lower the number of branches that must be traversed to find an optimal solution. However, partitioning a vertex-weighted graph G^w for the MWCP is more complicated than partitioning an unweighted graph G for the MCP. For example, we could partition G^w into A and B by constructing a weighted IS cover $\{D_1, D_2, \ldots, D_p\}$ incrementally and stop the construction once $\sum_{i=1}^{p} W_i > w(C^*) - w(C)$, where W_i is the weight of D_i. However, we could not take $A = D_1 \cup D_2 \cup \cdots \cup D_{p-1}$ because the weights of some vertices in $D_1 \cup D_2 \cup \cdots \cup D_{p-1}$ are not completely covered. Indeed, we must take $A = (D_1 \cup D_2 \cup \cdots \cup D_{p-1}) \setminus \{v | w(v) > \sum_{v \in D_i} W_i, 1 \leq i \leq p-1\}$, and $B = V \setminus A$. In other words, A is the set of vertices whose weight is completely covered by $\{D_1, D_2, \ldots, D_{p-1}\}$ and B is the set of the remaining vertices. The set B must include the vertices in the weighted IS cover whose weight is not completely covered. This approach was employed in Held et al. (2012) to partition G^w.

In the BnB MWCP algorithm WLMC (Jiang et al., 2017), an algorithm called GetBranches uses incremental MaxSAT reasoning to partitions G^w into A and B by first identifying a subset A of V in such a way that $G^w[A]$ has an IS partition Π and its UB_{IS} is not greater than $w(C^*) - w(C)$. Let $B = V \setminus A = \{b_1, b_2, \ldots, b_{|B|}\}$. Then, for $i = |B|, |B|-1, \ldots, 1$, the algorithm adds b_i to Π successively as a unit IS $\{b_i\}$, and then applies MaxSAT reasoning to compute the upper bound UB_{MaxSAT} of $G^w[A \cup \{b_i\}]$. If the computed UB_{MaxSAT} is not greater than $w(C^*) - w(C)$, the vertex b_i is removed from B and added to A because a clique of weight greater than $w(C^*)$ cannot be found in $G^w[A \cup \{b_i\}]$. In this way, the number of vertices of B is greatly reduced by MaxSAT reasoning. However, the algorithm GetBranches is based on incremental MaxSAT reasoning, and is really sophisticated and complex.

In the next subsection, we present a novel approach to partition G^w with UB_{WC} that is based on the incremental construction of a weight cover for G^w.

5.2. Minimizing the number of branches with UB_{WC}

Given a vertex-weighted graph $G^w = (V, E, w)$, a lower bound t of $\omega(G^w)$ and a vertex ordering $O : v_1 < v_2 < \cdots < v_n$, the algorithm Partition$_{WC}(G^w, t, O)$, described in Algorithm 2, partitions V into A and B by maintaining a weight cover Π of $G^w[A]$ and ensuring that $UB_{WC}(G^w[A]) \leq t$. Initially, A, B and Π are empty, and $UB_{WC}(G^w[A]) = 0$. Then, for $i = n, n-1, \ldots, 1$ (in this ordering), it incrementally constructs a weight cover for $G^w[A \cup \{v_i\}]$. If $UB_{WC}(G^w[A \cup \{v_i\}]) \leq t$, then the algorithm inserts v_i into A; otherwise, it inserts v_i into B.

Let v_i be the current vertex to be inserted into A, and let $\Pi = \{(D_1, w_1), (D_2, w_2), \ldots, (D_k, w_k)\}$ be a weight cover of $G^w[A]$ with $A = D_1 \cup D_2 \cup \cdots \cup D_k$. To insert v_i into A, Algorithm 2 tries to identify or derive a subset of ISs $\Delta = \{D_{j_1}, D_{j_2}, \ldots, D_{j_p}\}$ in Π not containing any neighbor of v_i so that the weight of vertex v_i can be split into the ISs of Δ. Concretely, for the first $p-1$ ISs in Δ, which are existing ISs in Π found in line 8, the algorithm defines $w_{j_s}(v_i) = w^*_{j_s}(D_{j_s})$ and inserts v_i into D_{j_s} (line 26); i.e., each D_{j_s} covers a part $w^*_{j_s}(D_{j_s})$ of the weight $w(v_i)$. Then, the algorithm uses the last IS D_{j_p} to receive $v_i^{w_{j_p}(v_i)}$, where $w_{j_p}(v_i) = w(v_i) - \sum_{s=1}^{p-1} w_{j_s}(v_i)$, the residual weight of v_i (line 27). Note that $w(v_i) = \sum_{s=1}^{p} w_{j_s}(v_i)$. Inserting v_i with weight $w_{j_s}(v_i)$ into every D_{j_s} obviously gives a weight cover of $G^w[A \cup \{v_i\}]$. Furthermore, since $w_{j_s}(v_i)$ is defined to be $w^*_{j_s}(D_{j_s})$ in the first $p-1$ ISs, every $w^*_{j_s}(D_{j_s})$ of the first $p-1$ ISs is unchanged, and the upper bound $UB_{WC}(G^w[A \cup \{v_i\}])$ is increased by $\max(w_{j_p}(v_i) - w^*_{j_p}(D_{j_p}), 0)$; i.e., $UB_{WC}(G^w[A \cup \{v_i\}]) = UB_{WC}(G^w[A]) + \max(w_{j_p}(v_i) - w^*_{j_p}(D_{j_p}), 0)$. Algorithm 2 inserts v_i into A and Π if $UB_{WC}(G^w[A \cup \{v_i\}]) \leq t$; otherwise, it inserts v_i into B.

The last IS D_{j_p} used to receive the residual weight of v_i is derived from an existing IS D_x of Π as described below if D_x can be found in line 12; otherwise, D_{j_p} is the last existing IS found in line 8.

Let $D_x = \{v_{x_1}^{w_x(v_{x_1})}, \ldots, v_{x_a}^{w_x(v_{x_a})}, v_{x_{a+1}}^{w_x(v_{x_{a+1}})}, \ldots, v_{x_b}^{w_x(v_{x_b})}\}$ be an IS of Π found in line 12, in which v_{x_a} is the neighbour of v_i with the greatest weight in D_x, such that $\beta = w^*_x(D_x) - w_x(v_{x_a}) > 0$ is the greatest value among all ISs containing a neighbour of v_i considered so far. Assume that $w_x(v_{x_1}) \geq \cdots \geq w_x(v_{x_a}) \geq w_x(v_{x_{a+1}}) \geq \cdots \geq w_x(v_{x_b})$. Since $\beta > 0$, we can split D_x into $D_{x'} = \{v_{x_1}^{w_x(v_{x_a})}, v_{x_2}^{w_x(v_{x_a})}, \ldots, v_{x_a}^{w_x(v_{x_a})}, v_{x_{a+1}}^{w_x(v_{x_{a+1}})}, \ldots, v_{x_b}^{w_x(v_{x_b})}\}$ and $D_{x''} = \{v_{x_1}^{w_x(v_{x_1}) - w_x(v_{x_a})}, \ldots, v_{x_{a-1}}^{w_x(v_{x_{a-1}}) - w_x(v_{x_a})}\}$. Note that $\beta > 0$ is the condition to guarantee that $D_{x''}$ is non-empty after removing all the vertices with weight 0 from $D_{x''}$. Obviously, $(\Pi \setminus \{(D_x, w_x)\}) \cup$

Algorithm 2: Partition$_{WC}(G^w, t, O)$, an algorithm to partition the vertices of G^w into two subsets.

Input: $G^w = (V, E, w)$, an integer t and a vertex ordering O
Output: a partition of V into A and B such that $\mathrm{UB}_{WC}(G^w[A]) \le t$

1 **begin**
2 $\Pi \leftarrow \emptyset$; /* Π *will be a weight cover of the form* $\{(D_1, w_1), (D_2, w_2), \ldots\}$ */
3 Let $v_1 < v_2 < \cdots < v_{|V|}$ be the ordering of V w.r.t. O;
4 **for** $i = |V|$ *to* 1 **do**
5 $\beta \leftarrow 0$;
6 $\Delta \leftarrow \emptyset$; /* Δ will be a subset of ISs not containing any neighbour of v_i */
7 **for** $j = 1$ *to* $|\Pi|$ **do**
8 **if** $\Gamma(v_i) \cap D_j = \emptyset$ **then** $\Delta \leftarrow \Delta \cup \{D_j\}$;
9 **else**
10 Let u be the neighbor of v_i with the greatest weight in D_j;
11 **if** $w^*_j(D_j) - w_j(u) > \beta$ **then**
12 $x \leftarrow j$, $\beta \leftarrow w^*_j(D_j) - w_j(u)$;
13 **if** $\sum_{j=1}^{|\Pi|} w^*_j(D_j) + \max(w(v_i) - (\sum_{D_k \in \Delta} w^*_k(D_k) + \beta), 0) \le t$ **then**
14 **break**;
15 **if** $\sum_{j=1}^{|\Pi|} w^*_j(D_j) + \max(w(v_i) - (\sum_{D_k \in \Delta} w^*_k(D_k) + \beta), 0) \le t$ **then**
16 **if** $\sum_{D_k \in \Delta} w^*_k(D_k) < w(v_i)$ *and* $\beta > 0$ **then**
17 Let $D_x = \{v_{x_1}^{w_x(v_{x_1})}, \ldots, v_{x_a}^{w_x(v_{x_a})}, v_{x_{a+1}}^{w_x(v_{x_{a+1}})}, \ldots, v_{x_b}^{w_x(v_{x_b})}\}$ with $w_x(v_{x_1}) \ge \cdots \ge w_x(v_{x_a}) \ge w_x(v_{x_{a+1}}) \ge \cdots \ge w_x(v_{x_b})$;
18 Let v_{x_a} be the neighbor of v_i with the greatest weight in D_x;
19 Let $D_{x'} = \{v_{x_1}^{w_x(v_{x_a})}, \ldots, v_{x_a}^{w_x(v_{x_a})}, v_{x_{a+1}}^{w_x(v_{x_{a+1}})}, \ldots, v_{x_b}^{w_x(v_{x_b})}\}$;
20 Let $D_{x''} = \{v_{x_1}^{w_x(v_{x_1}) - w_x(v_{x_a})}, \ldots, v_{x_{a-1}}^{w_x(v_{x_{a-1}}) - w_x(v_{x_a})}\}$;
21 $\Pi \leftarrow (\Pi \setminus \{(D_x, w_x)\}) \cup \{(D_{x'}, w_{x'}), (D_{x''}, w_{x''})\}$; $\Delta \leftarrow \Delta \cup \{D_{x''}\}$;
22 **if** $\Delta \neq \emptyset$ **then**
23 Let $\Delta = \{D_{j_1}, D_{j_2}, \ldots, D_{j_p}\}$;
24 /* Distribute the weight $w(v_i)$ over $D_{j_1}, D_{j_2}, \ldots, D_{j_p}$ */
25 **for** $s = 1$ *to* $p - 1$ **do**
26 $w_{j_s}(v_i) \leftarrow w^*_{j_s}(D_{j_s})$; $D_{j_s} = D_{j_s} \cup \{v_i^{w_{j_s}(v_i)}\}$;
27 $w_{j_p}(v_i) = w(v_i) - \sum_{s=1}^{p-1} w_{j_s}(v_i)$;
$D_{j_p} = D_{j_p} \cup \{v_i^{w_{j_p}(v_i)}\}$;
28 **else**
29 **create** a new IS $D_{|\Pi|+1} = \{v_i^{w(v_i)}\}$;
30 $\Pi \leftarrow \Pi \cup \{(D_{|\Pi|+1}, w_{|\Pi|+1})\}$;
31 $A \leftarrow A \cup \{v_i\}$;
32 **else** $B \leftarrow B \cup \{v_i\}$;/* v_i *cannot be added to* A */;
33 **return** (A, B);

$\{(D_{x'}, w_{x'}), (D_{x''}, w_{x''})\}$ is a weight cover of $G^w[A]$ without changing $\mathrm{UB}_{WC}(G^w[A])$, because $w_x(v_{x_a})$ and $\beta = w_x(v_{x_1}) - w_x(v_{x_a})$ are the greatest weights in $D_{x'}$ and $D_{x''}$, respectively. So, $D_{x''}$ is the last IS D_{j_p} and the algorithm inserts v_i into $D_{x''}$.

If there is no IS in Π in which v_i can be inserted (i.e., $\Delta = \emptyset$ and $\beta = 0$) and $\sum_{j=1}^{|\Pi|} w^*_j(D_j) + w(v_i) \le t$, Algorithm 2 creates a new empty IS to inserts v_i into A (lines 28–30).

Concretely, Algorithm 2 inserts v_i into A in the following four cases:

1. It finds a set of ISs $\Delta = \{D_{j_1}, D_{j_2}, \ldots, D_{j_p}\}$ not containing any neighbour of v_i such that $\sum_{j=1}^{|\Pi|} w^*_j(D_j) + \max(w(v_i) - \sum_{D_k \in \Delta} w^*_k(D_k), 0) \le t$.
   In this case, each D_{j_s} of the first $p-1$ ISs covers a part $w_{j_s}(v_i) = w^*_{j_s}(D_{j_s})$ of the weight $w(v_i)$, and the last IS D_{j_p} receives v_i with weight $w_{j_p}(v_i) = w(v_i) - \sum_{s=1}^{p-1} w_{j_s}(v_i)$. So, $\mathrm{UB}_{WC}(G^w[A \cup \{v_i\}])$ is increased by $\max(w_{j_p}(v_i) - w^*_{j_p}(D_{j_p}), 0)$ and the number of ISs in Π is not increased.
2. It finds a set of ISs $\Delta = \{D_{j_1}, D_{j_2}, \ldots, D_{j_{p-1}}\}$ not containing any neighbour of v_i such that $\sum_{j=1}^{|\Pi|} w^*_j(D_j) + w(v_i) - \sum_{s=1}^{p-1} w^*_{j_s}(D_{j_s}) > t$, and finds an IS D_x containing some neighbours of v_i in which the most weighted vertex u is not a neighbour of v_i, such that $\sum_{j=1}^{|\Pi|} w^*_j(D_j) + \max(w(v_i) - (\sum_{s=1}^{p-1} w^*_{j_s}(D_{j_s}) + \beta), 0) \le t$, where $\beta = w^*_x(D_x) - w_x(u)$.
   In this case, each D_{j_s} of the $p-1$ ISs in Δ covers a part $w^*_{j_s}(D_{j_s})$ of the weight $w(v_i)$, and D_x is split into $D_{x'}$ and $D_{x''}$ so that $D_{x''}$ receives v_i with weight $w_{x''}(v_i) = w(v_i) - \sum_{s=1}^{p-1} w_{j_s}(v_i)$, increasing $\mathrm{UB}_{WC}(G^w[A \cup \{v_i\}])$ by $\max(w_{x''}(v_i) - w^*_{x''}(D_{x''}), 0)$.
3. It does not find any IS not containing any neighbour of v_i but finds an IS D_x containing some neighbours of v_i in which the most weighted vertex u is not neighbour of v_i and $\sum_{j=1}^{|\Pi|} w^*_j(D_j) + \max(w(v_i) - \beta, 0) \le t$, where $\beta = w^*_x(D_x) - w_x(u)$.
   In this case, D_x is split into $D_{x'}$ and $D_{x''}$, v_i is inserted into $D_{x''}$ with weight $w(v_i)$, increasing $\mathrm{UB}_{WC}(G^w[A \cup \{v_i\}])$ by $\max(w(v_i) - w^*_{x''}(D_{x''}), 0)$.
4. It finds that the most weighted vertex in all ISs is a neighbour of v_i, but $\sum_{j=1}^{|\Pi|} w^*_j(D_j) + w(v_i)$ is not greater than t.
   In this case, $\Delta = \emptyset$ and $\beta = 0$. A new IS is created to receive v_i with weight $w(v_i)$, increasing $\mathrm{UB}_{WC}(G^w[A \cup \{v_i\}])$ by $w(v_i)$.

In summary, the number of ISs is not increased by inserting v_i into the ISs of Π in the first case, and is increased by one in the last three cases. In the first three cases, the upper bound $\mathrm{UB}_{WC}(G^w[A \cup \{v_i\}])$ is increased by a value smaller than $w(v_i)$, and is increased by $w(v_i)$ in case 4. Finally, an existing IS D_x is split in case 2 and case 3, and a new IS is created in case 4. Note that $w_j(v_i)$ is well defined for each IS D_j in which v_i is inserted.

Example 4. Consider the graph in Fig. 1. Assume that the vertex ordering O is $v_6 < v_5 < \cdots < v_1$ and the lower bound is $t = 6$. Initially, $A = B = \Pi = \emptyset$. Algorithm 2 partitions the vertex set as follows: It first creates a new IS $D_1 = \{v_1^3\}$ in Π for v_1^3 and inserts v_1 into A, because $\mathrm{UB}_{WC}(G^w[A]) = 3 < t$ after inserting v_1 (case 4). Then, it creates a new IS $D_2 = \{v_2^3\}$ for v_2^3, because v_2^3 is adjacent to v_1^3, the most weighted vertex in D_1 (case 4: $\beta = 0$ and D_1 cannot be split to receive v_2). Now, $A = \{v_1^3, v_2^3\}$ and $\mathrm{UB}_{WC}(G^w[A]) = 6 \le t$. Next, v_3^2 can be inserted into D_1 (case 1) and v_4^1 into D_2 (case 1) so that $A = \{v_1^3, v_2^2, v_3^2, v_4^1\}$ and $\mathrm{UB}_{WC}(G^w[A]) = \sum_{k=1}^{2} w^*_k(D_k) = 6 \le t$, because $\Pi = \{(D_1, w_1), (D_2, w_2)\} = \{\{v_1^3, v_3^2\}, \{v_2^3, v_4^1\}\}$. Since v_5^4 is not adjacent to any vertex in D_1 and D_2, and $\sum_{k=1}^{2} w^*_k(D_k) + \max(4 - (w^*_1(D_1) + w^*_2(D_2)), 0) = 6 + 0 \le 6$ (case 1), the algorithm inserts v_5^3 into D_1, which does not increase $w^*_1(D_1)$, and v_5^1 into

D_2. Now, the weight of v_5 has been totally distributed into D_1 and D_2, and $A = \{v_1^3, v_2^3, v_3^2, v_4^1, v_5^4\}$. Next, the algorithm considers v_6^2. Since v_6 is adjacent to vertex v_1^3, the most weighted vertex in D_1, we have that D_1 cannot be split to derive an IS to include v_6. Nevertheless, $D_2 = \{v_2^3, v_4^1, v_5^1\}$ can be split into $D_{2'} = \{v_2^1, v_4^1, v_5^1\}$ and $D_{2''} = \{v_2^2\}$, because v_4^1 is the neighbor of v_6 with the greatest weight in D_2 and $\beta = w_2^*(D_2) - 1 = 2$ (case 3). Then, the algorithm inserts v_6^2 into $D_{2''}$, giving $D_{2''} = \{v_2^2, v_6^2\}$ and $\Pi = \{\{v_1^3, v_5^3, v_3^2\}, \{v_2^1, v_4^1, v_5^1\}, \{v_2^2, v_6^2\}\}$. Since $UB_{WC}(G^w[A \cup \{v_6^2\}]) = 3 + 1 + 2 = 6 \leq t$, vertex v_6^2 is inserted into A. Finally, $A = \{v_1^3, v_2^3, v_3^2, v_4^1, v_5^4, v_6^2\}$ and $B = \emptyset$.

We use an integer pair (v_{xj}, w_{xj}) to store the vertex v_{xj} and its weight w_{xj} in the IS D_x in decreasing weight ordering. Splitting an IS D_x into $D_{x'}$ and $D_{x''}$ can be done in $O(|V|)$. When a new vertex is inserted into D_x, it is inserted in a position so that the decreasing weight ordering is kept. The complexity of the insertion is $O(|V|)$. Note that inserting a vertex $v_i \in V$ into Π results in at most one more IS in Π. The total number of ISs in Π is thus $O(|V|)$. If θ is the greatest weight among the vertices in G^w, each vertex $v \in V$ has at most $w(v)$ occurrences in Π. So, $\Sigma_{D \in \Pi}|D| \leq \theta \times |V|$. The main cost to cover the total weight of a vertex v by Π is the identification (or derivation) of the ISs in which v has no neighbour, requiring to check if each vertex in an IS is a neighbour of v. So, the complexity of covering the total weight of a vertex by Π is $O(\theta \times |V|)$, and the complexity of Algorithm 2 is $O(\theta \times |V|^2)$.

Algorithm 2 distributes the residual weight of v_i into at most one IS containing some neighbours of v_i in which the most weighted vertex is not a neighbour of v_i, by splitting this IS into two ISs. Distributing the residual weight of v_i over k of such ISs requires to split these ISs into $2 \times k$ ISs. Too many ISs in Π could slow down the partition of G^w into A and B.

From the computational point of view, the weight cover defined here is very different from the weighted IS cover used in Held et al. (2012) and Warren and Hicks (2006). In Algorithm 2, the construction of a weight cover for $G^w[A]$ is driven by splitting the total weight of a vertex, while the construction of a weighted IS cover is driven by generating a weighted IS. An IS in a weight cover can be split to receive new vertices, while an IS in a weighted IS cover is not changed once created.

From a practical point of view, a weight cover under construction covers the whole weight of every vertex occurring in the cover at any moment, while a weighted IS cover under construction, as we explained in Section 5.1, generally does not cover the whole weight of each vertex occurring in it before G^w becomes empty. Consequently, the weight cover defined in this paper performs more naturally in minimizing the number of branches in a BnB MWCP algorithm.

Algorithm 2 is also very different from the GetBranches algorithm used in Jiang et al. (2017) to partition G^w into A and B in that it is much simpler because it does not need to apply MaxSAT reasoning, which is very complex and time consuming. In fact, the complexity of detecting a subset of conflicting ISs in GetBranches is $O(|V|^2)$ (see Li et al., 2017 for a detailed complexity analysis of MaxSAT reasoning). Several sets of conflicting ISs have to be detected to insert a vertex into A. So, inserting a vertex into A in GetBranches requires $O(h \times |V|^2)$ time, where h is the maximum number of sets of conflicting ISs needed to insert a vertex into A. Therefore, the complexity of the GetBranches algorithm in Jiang et al. (2017) is $O(h \times |V|^3)$, which is much higher than the complexity of Algorithm 2.

Moreover, the detected conflicting ISs used to insert a vertex into A are disabled in Jiang et al. (2017) and cannot be used to cover the weight of other vertices, because the conflicting subsets of ISs should be disjoint to ensure the soundness of the approach of Jiang et al. (2017). However, in Algorithm 2, no IS is disabled and all the ISs used to insert a vertex v into A can be used to insert other vertices into A after including the weight of v. In this way, our new approach allows one to insert more vertices into A.

5.3. WC-MWC: a new exact MWCP algorithm

Preprocessing is crucial for the efficiency of MWCP and MCP algorithms, especially on massive graphs (Jiang et al., 2016; 2017; San Segundo et al., 2016). Thus, we combine the efficient preprocessing of Jiang et al. (2017) with Algorithm 1 to obtain a new exact algorithm called WC-MWC, as depicted in Algorithm 3.

Algorithm 3: WC-MWC(G^w), an exact algorithm for the MWCP.

Input: $G^w = (V, E, w)$
Output: a maximum weight clique C^* of G^w

```
1  begin
2      (C_0, O_0, G_1^w) ← Initialize(G^w, 0);
3      C* ← C_0, V_1 ← the vertex set of G_1^w;
4      Order V_1 w.r.t. the initial ordering O_0;
5      for i := |V_1| to 1 do
6          C ← {v_i}, P ← Γ(v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{|V_1|}};
7          if w(P) + w(C) > w(C*) then
8              (C'_0, O'_0, G_2^w) ← Initialize(G^w[P], w(C*) − w(C));
9              if w(C'_0) + w(C) > w(C*) then
10                 C* ← C'_0 ∪ C;
11             V_2 ← the vertex set of G_2^w;
12             C_2 ← SearchMWC(G_2^w, V_2, O'_0, C, C*);
13             if w(C_2) > w(C*) then
14                 C* ← C_2;
15     return C*;
```

Given $G^w = (V, E, w)$ and a lower bound lb of $\omega(G^w)$, the preprocessing procedure *Initialize*(G^w, lb) of Jiang et al. (2017) performs three tasks: derives a vertex ordering O_0, seeks an initial clique C_0 and reduces the input graph G^w to a simpler graph G_1^w. Initialize(G^w, lb) computes the ordering $O_0 : v_1 < v_2 < \cdots < v_n$ as follows: Given a copy H of G^w (without the weight function), it removes the vertex with the smallest degree in H and names it v_1; then, it removes the vertex with the smallest degree in $H[V \setminus \{v_1\}]$ and names it v_2, and so on. After removing the vertices $v_1, v_2, \ldots, v_i$ from H, if the smallest degree in $H[V \setminus \{v_1, v_2, \ldots, v_i\}]$ is $|V| - i - 1$, then $V \setminus \{v_1, v_2, \ldots, v_i\}$ becomes the initial clique C_0. If $w(C_0) > lb$, then $lb = w(C_0)$. Initialize(G^w, lb) returns $(C_0, O_0, G^w[V_1])$, where $V_1 = \{v \mid w(\{v\} \cup \Gamma(v)) > lb\}$. In other words, any vertex u such that the total weight of u and its neighbors is not greater than lb is eliminated from G^w, because u cannot be in any clique of weight greater than lb. Experiments in Jiang et al. (2017) show that this preprocessing can significantly speed up the search in real-world massive graphs.

WC-MWC calls Initialize(G^w, lb) to preprocess the original graph G^w (line 2) and all the first level subgraphs $G^w[P]$ (line 8), and then calls the BnB procedure SearchMWC (Algorithm 1) to search for a clique of weight greater than $w(C^*)$ in the reduced subgraph G_2^w (line 12). The impact of the preprocessing in WC-MWC is analyzed in detail in the supplementary materials of this paper.

BnB algorithms for the MCP and MWCP usually use distinct vertex orderings for branching and upper bounding. A feature of WC-MWC, as well as of the MCP algorithms in Li et al. (2017), is that the same ordering O_0 is used for branching (i.e. for sorting the branching candidates in Algorithm 1) and upper bounding (i.e., for sorting the vertices in the incremental construction of the weight

cover to minimize the number of branches in Algorithm 2). The ordering O_0 computed by the Initialize(G^w, lb) procedure is based on vertex degrees and is independent of vertex weights. MWCP algorithms can also sort the vertices according to their weight.

We empirically compared the ordering O_0 with two orderings based on vertex weights. The results, presented in the supplementary materials, indicate that the ordering based on vertex degrees is more effective for the incremental construction of weight covers and for branching in WC-MWC than the other two orderings.

Algorithm 3 solves massive and medium graphs in the same way. So, one does not need to judge if a graph is massive or medium before using Algorithm 3, because the algorithm makes no distinction between massive and medium graphs.

In the implementation of WC-MWC, vertices are represented by positive integers and are used as array indices. Once an ordering $v_1 \prec v_2 \prec \cdots \prec v_n$ is generated, it is kept during the search using an integer array $order[v_i] = i$. The vertex weight is kept in the same way. The input graph G^w, as well as the reduced graph G_1^w obtained at the root (line 2 of Algorithm 3) and each reduced graph G_2^w obtained at the first level of the search tree (line 8 of Algorithm 3), is stored in memory by associating a list of neighbours with each vertex. For every graph G_2^w given to the BnB search procedure SearchMWC, an adjacency matrix is built and stored in a bitmap to speed up the weight cover construction in SearchMWC.

6. Empirical investigation

We empirically evaluated the new upper bound UB_{WC} and the algorithm WC-MWC. WC-MWC was implemented in C and compiled using GNU gcc -O3. Experiments were performed on Intel Xeon CPUs E5-2680 v4@2.40 gigahertz under Linux with 128 gigabytes of memory. We tested two types of graphs in the experiments:

Medium and small graphs: They include DIMACS,[1] BHOSLIB[2] and random graphs and contain up to 15,000 vertices. Their density ranges from 0.1 to 0.99.

Real-world massive sparse graphs: They are from the Network Data Repository[3] (Rossi and Ahmed, 2015) and were used to evaluate MWCP algorithms in Cai and Lin (2016), Jiang et al. (2017) and Wang et al. (2016). The graphs contain up to 66M vertices and 1800M edges.

The weights of the vertices in DIMACS, BHOSLIB, random and massive graphs are assigned as in Cai and Lin (2016), Fang et al. (2016), Jiang et al. (2017) and Wang et al. (2016): each vertex v is represented by a positive integer and its weight is defined to be $w(v) = (v \bmod 200) + 1$.

6.1. The efficiency of UB_{WC} in reducing the number of branches in WC-MWC

We implemented the next three *Partition* functions for Algorithm 1 to evaluate the efficiency of UB_{WC} in reducing the number of branches in WC-MWC.

$Partition_{IS}$: It is the *Partition* function implemented with UB_{IS}. Initially, A, B and Π are empty. Let $\Pi = \{D_1, D_2, \ldots, D_r\}$ be the IS partition of $G^w[A]$. The vertex v is inserted into A if v can be inserted into an existing IS D_i of Π so that $\sum_{i=1}^{r} w^*(D_i) \le t$ or into a newly created IS D_{r+1} so that $\sum_{i=1}^{r+1} w^*(D_i) \le t$; otherwise, v is inserted into B.

$Partition_{MaxSAT}$: It is the *Partition* function implemented with UB_{MaxSAT}. Let $B = \{b_1, b_2, \ldots, b_{|B|}\}$ and $A = V \setminus B$ be the initial partition of the vertices of G^w given by $Partition_{IS}$. $Partition_{MaxSAT}$ reduces the cardinality of B as in WLMC Jiang et al. (2017): for $i = |B|, |B|-1, \ldots, 1$, removes every b_i from B and adds it to A if the computed UB_{MaxSAT} for $(G^w[A \cup \{b_i\}])$ is not greater than t.

$Partition_{WISC}$: It is the *Partition* function implemented with UB_{WISC}. It partitions V by computing the ISs $\{D_1, D_2, \ldots, D_p\}$ incrementally: Let $O : v_1 \prec v_2 \prec \cdots \prec v_n$ be the initial vertex ordering of G^w; from n to 1, select a vertex v_{i_1} of minimum (residual) weight in G^w and construct a maximal IS $(v_{i_1}, v_{i_2}, \ldots, v_{i_q})$ without considering the vertex weights; assign the weight $w(v_{i_1})$ to the IS; decrease the weight of each vertex of G^w that is in the IS by $w(v_{i_1})$; and remove the vertices with weight 0 from G^w before computing another IS. The procedure stops once $\sum_{i=1}^{p} W_i > t$, where W_i is the weight of D_i. Then, it takes $A = \{v | w(v) = \sum_{v \in D_i} W_i, 1 \le i \le p-1\}$ and $B = V \setminus A$. MWSS (Held et al., 2012) uses an implementation similar to $Partition_{WISC}$ to minimize the number of branches.

Table 1
Comparison of the mean number of branching vertices returned by $Partition_{IS}$, $Partition_{MaxSAT}$, $Partition_{WISC}$ and $Partition_{WC}$ on DIMACS graphs and 5 BHOSLIB graphs, excluding the instances that were solved by WC-MWC within 1 second and the instances that were not solved by WC-MWC within 5 hours.

Instance	$\|V\|$	D	$\|B_{IS}\|$	$\|B_{MaxSAT}\|$	$\|B_{WISC}\|$	$\|B_{WC}\|$
brock400_1	400	0.75	9.27	1.16	1.28	**1.00**
brock400_2	400	0.75	8.69	1.06	1.15	**1.00**
brock400_3	400	0.75	9.32	1.13	1.26	**1.00**
brock400_4	400	0.75	8.76	1.08	1.21	**1.00**
brock800_1	800	0.65	9.63	1.87	1.06	**1.00**
brock800_2	800	0.65	9.33	1.78	1.08	**1.00**
brock800_3	800	0.65	9.33	1.80	1.08	**1.00**
brock800_4	800	0.65	9.03	1.75	1.04	**1.00**
C125.9	125	0.90	6.06	**0.35**	1.39	0.99
C250.9	250	0.90	14.92	**0.37**	1.82	1.00
DSJC1000_5	1000	0.50	7.10	2.28	1.03	**1.00**
DSJC500_5	500	0.50	6.43	2.13	1.03	**1.00**
gen200_p0.9_44	200	0.90	9.02	**0.29**	1.84	1.00
gen200_p0.9_55	200	0.90	9.86	**0.30**	2.27	1.00
gen400_p0.9_75	400	0.90	15.19	**0.13**	2.19	1.00
hamming8-4	256	0.64	4.69	1.57	**0.96**	0.99
johnson16-2-4	120	0.76	2.01	1.18	**0.94**	1.00
keller4	171	0.65	3.06	1.27	**0.81**	0.98
MANN_a27	378	0.99	1.11	**0.11**	1.02	1.00
MANN_a9	45	0.93	0.96	**0.58**	1.27	0.92
p_hat1000-1	1000	0.24	4.94	1.77	1.04	**0.97**
p_hat1000-2	1000	0.49	18.29	1.66	3.31	**1.00**
p_hat1500-1	1500	0.25	6.82	2.85	1.20	**0.99**
p_hat1500-2	1500	0.51	26.01	1.80	3.98	**1.00**
p_hat300-1	300	0.24	3.15	0.97	0.68	**0.44**
p_hat300-2	300	0.49	5.67	1.22	1.36	**0.89**
p_hat300-3	300	0.74	10.64	1.06	1.77	**0.99**
p_hat500-1	500	0.25	3.92	2.02	0.98	**0.85**
p_hat500-2	500	0.50	10.84	1.56	2.02	**0.96**
p_hat500-3	500	0.75	17.48	1.15	3.03	**1.00**
p_hat700-1	700	0.25	4.05	2.16	0.98	**0.90**
p_hat700-2	700	0.50	15.65	1.50	3.21	**0.98**
p_hat700-3	700	0.75	28.91	1.80	4.95	**1.00**
san1000	1000	0.50	3.14	0.89	**0.58**	0.98
san200_0.9_1	200	0.90	3.00	**0.84**	1.28	0.97
san200_0.9_2	200	0.90	8.41	**0.25**	1.53	1.00
san200_0.9_3	200	0.90	8.59	**0.22**	1.47	1.00
san400_0.5_1	400	0.50	2.41	0.84	**0.63**	0.79
san400_0.7_1	400	0.70	5.32	0.58	**0.56**	1.00
san400_0.7_2	400	0.70	4.70	0.72	**0.53**	1.00
san400_0.7_3	400	0.70	4.97	1.02	**0.80**	1.00
san400_0.9_1	400	0.90	14.23	**0.04**	0.42	1.00
sanr200_0.7	200	0.70	5.55	1.04	1.12	**0.99**
sanr200_0.9	200	0.90	10.89	**0.25**	1.96	1.00
sanr400_0.5	400	0.50	5.55	1.99	1.05	**0.99**
sanr400_0.7	400	0.70	8.10	1.30	1.16	**1.00**
frb30-15-1	450	0.82	6.56	1.34	**0.58**	1.00
frb30-15-2	450	0.82	6.61	1.54	**0.47**	1.00
frb30-15-3	450	0.82	6.45	1.53	**0.44**	1.00
frb30-15-4	450	0.82	6.77	1.51	**0.30**	1.00
frb30-15-5	450	0.82	6.62	1.45	**0.29**	1.00

[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://networkrepository.com.

Table 2
Comparison of the runtimes in seconds of MWSS, MWCLQ, WLMC and WC-MWC, and the search tree sizes in 10^5 of MWCLQ, WLMC and WC-MWC on DIMACS graphs and 5 BHOSLIB graphs, excluding easy graphs that were solved by all the solvers within 1 second and hard graphs that were not solved by any solver within the cutoff time of 5 hours.

Instance	$\omega(G^w)$	MWSS	MWCLQ		WLMC		WC-MWC	
		Time	Tree	Time	Tree	Time	Tree	Time
brock200_1	2821	1.32	0.55	0.26	**0.26**	0.74	0.30	**0.21**
brock400_1	3422	811.1	141.6	**108.6**	**128.4**	484.0	142.0	132.5
brock400_2	3350	858.3	**132.4**	**121.4**	162.4	598.2	192.3	162.7
brock400_3	3471	860.0	110.4	**89.39**	**93.95**	340.4	105.1	104.7
brock400_4	3626	411.3	**84.39**	**76.89**	150.5	523.2	196.0	157.7
brock800_1	3121	11,304	**1202**	**1387**	2309	7124	1772	2338
brock800_2	3043	16,666	**1834**	**1933**	3086	9221	2634	2916
brock800_3	3076	14,396	**1335**	**1496**	2646	7801	2120	2564
brock800_4	2971	16,391	**1864**	**1726**	3218	9246	2720	3027
C2000.5	2466	–	**9189**	**11964**	–	–	11410	15,849
C250.9	5092	510.3	27.38	**35.68**	**14.23**	137.9	35.98	42.04
DSJC1000_5	2186	–	90.26	**81.55**	138.8	250.5	**76.90**	90.18
DSJC500_5	1725	–	1.42	**0.81**	2.29	3.20	**1.29**	1.16
gen200_p0.9_44	5043	44.38	6.16	5.45	**0.34**	2.88	1.10	**1.15**
gen200_p0.9_55	5416	26.26	2.12	2.44	**0.47**	3.66	1.25	**1.12**
gen400_p0.9_75	8006	–	–	–	**147.1**	3137	1218	**2890**
hamming10-2	50,512	**0.34**	12.34	839.3	**0.01**	3.60	–	–
johnson16-2-4	548.0	1.05	3.03	**0.10**	2.30	0.76	**2.20**	0.22
keller5	3317	–	8433	17487	–	–	**8007**	**15,584**
MANN_a27	12,283	–	–	–	**0.16**	**3.06**	0.54	5.48
MANN_a45	34,265	–	–	–	**5.31**	**1934**	207.6	6630
p_hat1000-1	1514	5.61	0.79	0.56	0.46	0.76	**0.31**	**0.43**
p_hat1000-2	5777	10,804	958.2	2178	10.06	98.12	**8.19**	**18.75**
p_hat1500-1	1619	46.10	5.21	3.84	3.93	6.08	**1.94**	**2.84**
p_hat1500-2	7360	–	–	–	469.2	8346	**391.2**	**1373**
p_hat300-3	3774	14.33	2.06	2.09	0.29	1.68	**0.26**	**0.42**
p_hat500-2	3920	17.24	1.57	2.01	0.12	0.70	**0.08**	**0.38**
p_hat500-3	5375	5508	438.7	757.1	10.73	118.2	**9.65**	**18.11**
p_hat700-1	1441	1.18	0.17	**0.13**	0.14	0.22	**0.07**	0.14
p_hat700-2	5290	439.6	22.99	40.40	0.30	2.95	**0.29**	**1.33**
p_hat700-3	7565	–	3243	9345	17.52	329.3	**12.65**	**45.79**
san1000	1716	8.15	122.1	173.6	**0.45**	**1.50**	0.52	8.18
san200_0.9_2	6082	11.57	1.26	1.40	**0.08**	**0.40**	0.38	0.45
san200_0.9_3	4748	82.49	13.89	14.03	**1.22**	9.64	6.16	**4.82**
san400_0.7_1	3941	6.22	2.42	3.23	**0.48**	2.41	1.30	**1.91**
san400_0.7_2	3110	14.28	4.51	**4.67**	**2.56**	9.16	5.45	5.38
san400_0.7_3	2771	26.52	6.70	6.10	**2.44**	8.10	3.34	**3.52**
san400_0.9_1	9776	2898	367.2	1096	**16.26**	**426.3**	969.2	2863
sanr200_0.9	5126	51.66	5.19	5.63	**0.99**	8.26	3.31	**3.22**
sanr400_0.5	1835	2.31	0.51	**0.30**	0.49	0.80	**0.32**	**0.30**
sanr400_0.7	2992	133.2	31.58	**24.10**	**30.57**	92.69	31.02	27.43
frb30-15-1	2990	9588	**173.1**	**229.1**	1665	7816	4457	4331
frb30-15-2	3006	2043	**16.37**	**28.45**	827.0	3960	1493	1932
frb30-15-3	2995	3730	**93.47**	**125.8**	673.2	3116	1395	1520
frb30-15-4	3032	4329	**99.73**	**180.0**	1104	5029	1274	1284
frb30-15-5	3011	1518	**34.91**	**53.21**	707.7	3573	1664	2116

We solved the 80 DIMACS instances and 5 BHOSLIB graphs (*frb30**) with WC-MWC and compared the mean number of branching vertices obtained with the four *Partition* functions. At each search tree node of WC-MWC, we computed the number of branching vertices returned by each *Partition* function. Then, WC-MWC effectively branched on every vertex in the set B returned by Partition$_{WC}$ if B was not empty. Finally, the mean number of branching vertices of every *Partition* function was computed by dividing the total number of branching vertices in the search tree by the search tree size.

Table 1 shows the results of the comparison, excluding the easy instances that were solved by WC-MWC within 1 second and the hard instances that were not solved within 5 hours. WC-MWC is slower in Table 1 than in other tables because here it computes four *Partition* functions at every node. The mean number of branching vertices of Partition$_{IS}$, Partition$_{MaxSAT}$, Partition$_{WISC}$ and Partition$_{WC}$ are denoted by $|B_{IS}|$, $|B_{MaxSAT}|$, $|B_{WISC}|$ and $|B_{WC}|$, respectively. Some means are smaller than 1 because the set of branching vertices returned by the corresponding *Partition* functions is empty at some nodes. Partition$_{WC}$ produces the smallest mean number of branching vertices on 26 out of a total of 51 instances. Partition$_{MaxSAT}$ produces the smallest mean number of branching vertices on 12 instances, which are highly dense graphs such as *gen** and *MANN**. Partition$_{WISC}$ produces the smallest mean number of branching vertices on 13 instances. In general, Partition$_{WC}$ produces substantially smaller sets of branching vertices thanks to UB$_{WC}$.

6.2. Comparison of WC-MWC with other algorithms

The following algorithms (also called *solvers*) are compared with WC-MWC:

MWSS: It is an exact BnB solver for the Maximum Weight Stable Sets (MWSS) (Held et al., 2012). To solve the MWCP for $G^w = (V, E, w)$, we run MWSS[4] on the complement graph of G^w. At

[4] The source code is available at https://code.google.com/archive/p/exactcolors/source.

Table 3
Comparison of the mean runtimes in seconds and mean tree sizes in 10^5 for random graphs with different number of vertices (N) and densities (D). There are 50 graphs to solve at each point (N, D). The symbol '–" means that the corresponding solver did not solve any graph within 5000 seconds at the point. When a solver solved less than 50 graphs at a point, the number of graphs solved by each solver (#solved) is shown in the next row.

N	D	$\omega(G^w)_{avg}$	MWSS	MWCLQ		WLMC		WC-MWC	
			Time	Tree	Time	Tree	Time	Tree	Time
150	0.9	3406.08	2.31	0.66	0.47	**0.08**	0.42	0.20	**0.17**
150	0.95	4795.12	0.77	0.38	0.38	**0.03**	0.13	0.06	**0.09**
150	0.98	6826.42	**0.01**	**0.03**	0.03	0.04	0.10	**0.03**	0.05
200	0.9	5091.34	93.00	13.53	12.97	**1.49**	11.42	4.82	**4.29**
200	0.95	7456.96	78.26	16.42	22.30	**0.25**	3.17	1.40	**1.92**
200	0.98	10923.1	**0.12**	0.11	0.32	**0.08**	0.29	**0.08**	0.18
300	0.7	2470.16	5.52	1.35	**0.98**	1.75	4.57	**1.31**	1.28
300	0.8	3348.92	107.0	**14.23**	**12.85**	16.32	71.72	17.21	16.75
300	0.9	5339.10	–	592.4	835.2	**208.3**	2311	531.7	**788.3**
	#solved		0	50		47		50	
400	0.6	2271.20	11.39	3.14	**1.99**	2.92	5.91	**2.34**	2.20
400	0.7	2930.34	152.0	30.19	**22.99**	30.28	86.45	**29.74**	26.36
400	0.8	4078.04	4450	890.0	**870.3**	**703.1**	3551	1127	1138
	#solved		2	50		47		50	
500	0.5	1819.36	5.13	1.08	**0.75**	1.75	2.53	**0.93**	0.90
500	0.6	2283.50	36.49	**7.08**	**5.54**	11.41	24.70	7.17	7.74
500	0.7	2968.24	694.4	**91.48**	**85.05**	157.7	511.2	124.0	134.5
1000	0.4	1770.26	59.22	8.32	**5.84**	12.94	14.83	**6.56**	6.92
1000	0.5	2193.32	625.7	89.39	**77.74**	143.0	245.2	**79.93**	94.45
1000	0.6	2787.64	–	**1708**	**1861**	–	–	2338	2892
	#solved		0	50		0		50	
2000	0.3	1607.38	245.0	25.34	**18.87**	44.13	37.15	**17.93**	19.15
2000	0.4	2001.04	3664	339.2	**327.8**	564.2	829.1	**260.2**	375.4
2000	0.45	2237.84	–	**1515**	**1675**	3348	4732	1567	2248
	#solved		0	50		26		50	
5000	0.1	1098.80	177.6	3.68	5.71	5.05	4.69	**2.41**	**4.26**
5000	0.2	1459.76	1819	98.08	94.53	157.8	126.1	**46.23**	**68.18**
5000	0.3	1834.98	–	2250	2799	3683	4641	**1556**	**2684**
	#solved		0	50		46		50	
10,000	0.1	1191.90	2616	25.03	52.11	29.49	46.85	**18.70**	**43.46**
10,000	0.2	1605.60	–	1797	2720	3719	2642	**1590**	**2075**
	#solved		0	50		50		50	
15,000	0.1	1267.88	–	101.6	264.9	87.62	198.5	**53.73**	**180.4**
	#solved		0	50		50		50	

every search tree node, MWSS uses the UB_{WISC} based on a weighted IS cover to minimize the number of branches.

MWCLQ: It is one of the best exact MWCP solvers for medium and small graphs. It applies extended MaxSAT reasoning to compute tight upper bounds of $\omega(G^w)$ (Fang et al., 2016). The implementation of MWCLQ does not allow to solve massive graphs. So, we only use it to test medium and small graphs.

WLMC: It is a very recent exact MWCP solver[5] (Jiang et al., 2017). WLMC and WC-MWC share the same implementation and preprocessing, but WLMC uses UB_{MaxSAT} to minimize the number of branches and WC-MWC uses UB_{WC}.

FastWClq: It is a very recent heuristic MWCP solver that interleaves between clique construction and graph reduction. It outperforms other heuristic solvers like LSCC+BMS (Wang et al., 2016) on massive sparse graphs (Cai & Lin, 2016). We compare WC-MWC with FastWClq to refute the prevailing hypothesis that states that exact MWCP algorithms are less adequate than heuristic algorithms on massive graphs.

All reported times include both preprocessing and search times for each solver, but does not include the time for reading the input graphs into the memory.

We solved 80 DIMACS graphs and 5 BHOSLIB graphs (*frb30**) using a cutoff time of 5 hours. The exact solvers MWSS, MWCLQ, WLMC and WC-MWC solved 58, 63, 65 and 66 DIMACS instances, respectively, and all the BHOSLIB instances. Table 2 shows the runtimes for the four solvers and the search tree sizes for MWCLQ, WLMC and WC-MWC. It does not include the graphs that all the solvers solved within 1 second and the graphs that were not solved by any solver within 5 hours.

Table 4
The highest density D in which each solver is able to solve 50 graphs at the point (N, D) within 5000 seconds for a fixed number of vertices N.

N	MWSS	MWCLQ	WLMC	WC-MWC
300	0.87	0.91	0.89	**0.93**
400	0.78	**0.82**	0.79	**0.82**
500	0.74	**0.79**	0.75	0.78
1000	0.56	**0.62**	0.58	0.60
2000	0.40	**0.47**	0.44	0.46
5000	0.22	**0.31**	0.29	**0.31**
10,000	0.11	**0.21**	**0.21**	**0.21**
15,000	0.06	0.15	**0.16**	**0.16**

In Table 2, WLMC solves two graphs more than MWCLQ, because MaxSAT reasoning is incremental in WLMC but is not incremental in MWCLQ. WC-MWC also solves three graphs more than MWCLQ thanks to UB_{WC}. Interestingly, if we set the cutoff time to 5000 seconds, WLMC solves two graphs less than MWCLQ, suggesting that the MaxSAT reasoning in MWCLQ degenerates more easily than the incremental MaxSAT reasoning in WLMC.

WLMC outperforms WC-MWC on the *MANN* family and the *san1000* graph. *MANN* graphs are very dense, so that the ISs

[5] Its source code is available at http://home.mis.u-picardie.fr/~cli/EnglishPage.html.

contain very few vertices in general and MaxSAT reasoning identifies many disjoint conflicting subsets of ISs to minimize the number of branches in WLMC. In fact, the mean number of branches at a search tree node generated by $\text{Partition}_{MaxSAT}$ for a *MANN* graph is substantially smaller than the mean number of branches generated by Partition_{WC} (see Table 1), and the search tree of WLMC is also substantially smaller. As for the *san1000* graph, WLMC and WC-MWC produce search trees of similar size, but WC-MWC spends more time to compute the branching vertices at a node because it often has to split ISs when solving *san1000*. Nevertheless, WC-MWC solves one graph more than WLMC in Table 2, and is substantially faster than WLMC on *brock**, *p_hat** and *frb** graphs. MWSS has the worst performance on the solved graphs. Overall, WC-MWC is the most robust solver on the DIMACS and BHOSLIB graphs.

Table 3 shows the mean runtimes of MWSS, MWCLQ, WLMC and WC-MWC using a cutoff time of 5000 seconds for each solver and each graph, as well as the mean search tree sizes of MWCLQ, WLMC and WC-MWC, on random graphs with different numbers of vertices (N) and densities (D). At each point (N, D), 50 graphs of N vertices were randomly generated so that every two vertices are adjacent with probability D. The means of $\omega(G^w)$ for each point (N, D) are denoted by $\omega(G^w)_{avg}$. When there is at least one solver that does not solve 50 graphs at a point, the total number of graphs solved by each solver is shown in the next row. We do not display the results for the densities that are too easy for a given N. WC-MWC and MWCLQ solve the 50 graphs at every point in Table 3. MWSS cannot solve any graph at 6 points, and WLMC solves less than 50 graphs at 5 points. WC-MWC is faster than MWCLQ for very dense graphs (density is not smaller than 0.9) or very sparse

Table 5
Comparison of WC-MWC with WLMC and FastWClq on massive graphs. "–" means that the graph was not solved within the cutoff time; "↓" means that the best solution found by FastWClq is not optimal. Runtimes are in second and tree sizes are in 10^5.

Instance	$\|V\|$	D	$\omega(G^w)$	WC-MWC		WLMC		FastWClq	
				Tree	Time	Tree	Time	Best	Avgt
#cutoff time=1000 seconds									
aff-digg	872.6K	6E−05	3836	**49.54**	**186.8**	204.6	756.0	2967↓	948.1
aff-flickr-user-groups	396.0K	1E−04	1720	**1.36**	6.44	1.51	**6.02**	1720	622.8
aff-orkut-user2groups	8730K	9E−06	971	74.11	378.1	**34.58**	**375.5**	848↓	819.3
dbpedia-link	11,621K	1E−06	5062	21.68	**24.36**	**6.17**	26.67	4973↓	560.5
delaunay_n24	16,777K	4E−07	797	159.1	7.71	**26.39**	8.21	797	**5.16**
friendster	8658K	1E−06	5511	13.04	**5.28**	**1.63**	5.49	2885↓	92.44
hugebubbles-00020	21,198K	1E−07	400	98.62	5.68	**20.18**	6.49	400	**5.14**
inf-europe_osm	50,912K	4E−08	646	233.5	8.54	**0.1**	**8.25**	646	8.66
inf-road-usa	23,947K	1E−07	766	90.70	6.14	**0.1**	6.98	766	**5.84**
rec-dating	168.8K	1E−03	1699	**1.39**	17.08	1.80	**15.23**	1568↓	554.9
rec-libimseti-dir	221.0K	7E−04	1938	**1.74**	**13.20**	1.77	13.36	1938	468.5
rec-movielens	71.57K	4E−03	3777	**1.26**	**20.41**	4.13	35.80	3420↓	954.4
rgg_n_2_24_s0	16776K	8E−07	2514	48.67	11.33	**0.1**	12.25	2514	**9.33**
scc_twitter-copen	8.58K	1E−02	58,699	0.79	6.63	**0.39**	8.38	58,699	**0.12**
sc-TSOPF-RS-b2383-c1	38.12K	2E−02	960	**0.38**	**17.32**	4.71	43.28	960	230.9
soc-digg	770.8K	2E−05	5303	**1.17**	**4.17**	1.23	5.83	5303	82.44
socfb-A-anon	3097K	5E−06	2872	5.05	5.96	**3.09**	**5.01**	2872	30.25
socfb-konect	59,216K	5E−08	981	91.69	12.83	**0.46**	**11.07**	981	33.29
socfb-uci-uni	58,790K	5E−08	1045	27.92	**8.48**	**0.25**	8.72	1045	40.45
soc-flickr-und	1715K	1E−05	10,127	**4.22**	**42.34**	7.41	170.5	10,126↓	514.9
soc-livejournal-user-groups	7489K	4E−06	1054	30.94	60.39	**23.44**	**59.62**	991↓	608.8
soc-ljournal-2008	5363K	3E−06	40,432	0.74	**3.57**	**0.57**	6.36	40,258↓	20.32
soc-orkut-dir	3072K	2E−05	6147	26.66	55.50	**5.39**	**47.93**	6147	64.36
soc-orkut	2997K	2E−05	5452	22.76	43.02	**6.13**	**39.94**	5452	65.73
soc-sinaweibo	58,655K	1E−07	4759	73.93	**48.02**	**6.67**	52.31	4545↓	922.2
tech-ip	2250K	9E−06	668	2.65	9.02	**1.39**	**8.63**	587↓	857.8
web-wikipedia-growth	1870K	2E−05	4741	9.22	**10.83**	**3.02**	11.39	4741	72.62
web-wikipedia_link_it	2936K	2E−05	89,947	2.10	**30.94**	**2.03**	80.27	2500↓	4.15
wikipedia_link_en	27,154K	8E−08	4624	19.41	6.48	**2.00**	**6.25**	4624	96.01
#cutoff time=10,000 seonds									
bio-human-gene1	22.28K	5E−02	134,713	30.99	6635	**6.66**	**2637**	134,362↓	4571
bio-human-gene2	14.34K	9E−02	135,310	27.72	1801	**4.11**	**1474**	135,059↓	1097
bio-mouse-gene	45.10K	1E−02	59,952	139.9	**3697**	**50.86**	4024	59,855↓	1840
bn-···865_session_1-bg	1827K	1E−04	29,370	7.04	**1054**	**6.67**	1391	28,544↓	7467
bn-···867_session_1-bg	1827K	9E−05	29,425	5.04	779.2	**4.44**	**676.2**	29,208↓	5491
bn-···867_session_2-bg	1827K	9E−05	36,021	3.96	846.2	**3.57**	**711.9**	35,428↓	7571
bn-···868_session_1-bg	1827K	9E−05	31,940	**553.3**	7912	–	-	31,940	**249.7**
bn-···869_session_1-bg	1827K	8E−05	27,957	**5.56**	**925.0**	9.13	3075	27,453↓	3555
bn-···870_session_1-bg	1827K	9E−05	28,810	**22.84**	1055	–	-	28,810	**126.0**
bn-···871_session_1-bg	1827K	1E−04	37,828	16.33	1104	**15.58**	1357	37,828	**383.5**
bn-···873_session_2-bg	1827K	8E−05	32,445	8.36	**823.7**	**6.19**	1277	32,064↓	5330
bn-···874_session_2-bg	1827K	1E−04	30,885	13.08	1187	**11.62**	1779	30,885	**152.2**
bn-···876_session_1-bg	1827K	8E−05	50,355	**58.98**	1562	–	-	50,355	**584.6**
bn-···878_session_1-bg	1827K	8E−05	27,775	69.45	1219	**24.37**	4972	27,775	**135.7**
bn-···889_session_2	1827K	8E−05	24,771	18.93	831.3	**6.30**	**822.8**	24,497↓	7363
bn-···912_session_2	1827K	9E−05	35,063	10.58	885.9	**9.49**	3110	35,063	**33.03**
tech-p2p	5792K	9E−06	18897	**187.4**	**1684**	–	–	17,250↓	871.8
twitter_mpi	9862K	2E−06	13,524	46.54	**299.1**	**41.34**	1876	11,801↓	639.6

graphs (density is not greater than 0.3), but is slower than MWCLQ when the density is between 0.3 and 0.9.

Table 4 shows the highest density D in which each solver is able to solve 50 graphs at the point (N, D) within the cutoff time for a fixed number of vertices N. In the experiment, N ranges from 300 to 15,000.

Overall, WC-MWC and MWCLQ present a similar performance on random graphs and are significantly better than WLMC and MWSS on these graphs.

To evaluate WC-MWC on massive graphs, we considered 203 real-world graphs from the Network Data Repository, including the 52 and 90 graphs used to evaluate WLMC (Jiang et al., 2017) and FastWClq (Cai & Lin, 2016), respectively, and 13 graphs of brain networks. Since MWCLQ was not designed for massive graphs, we compared WC-MWC with WLMC and FastWClq, which can be regarded as the best exact and heuristic MWCP algorithms on massive graphs. For WC-MWC and WLMC, we report the time needed to solve each graph. FastWClq solved each graph 10 times with different seeds. We report the mean time (Avgt) to reach the best solution in each run, and the best solution found (Best) over the 10 runs. The cutoff time was set to 1000 seconds except for 18 hard graphs (*twitter_mpi, tech_p2p, bio** and the 13 graphs of *brain networks*), which used a cutoff time of 10,000 seconds.

Table 5 shows the results for 47 graphs, excluding the 156 graphs that both WC-MWC and WLMC solved within 5 seconds. The best times are in bold (FastWClq times are not in bold if the best solution found is not optimal). WC-MWC solved the 47 instances of the table, and is faster than WLMC and FastWClq on 19 instances. For example, WC-MWC is four times faster than WLMC for *aff-digg*, *soc-flickr-und* and *bn-human-BNU_1_0025878_session_1-bg*, and is more than six times faster than WLMC for *twitter_mpi*. WLMC did not solve 4 hard instances and FastWClq did not find the optimum of 23 instances (marked with '↓') within the cutoff time. WLMC is never more than two times faster than WC-MWC, except for the graph *bio-human-gene1*. Moreover, WC-MWC significantly outperforms WLMC on the graphs whose solving time is longer than 50 seconds. These results suggest that the incremental MaxSAT reasoning in WLMC degenerates more easily for massive graphs than the incremental weight cover construction in WC-MWC.

Note that the MaxSAT reasoning in UB_{MaxSAT} allows WLMC to develop a smaller search tree than WC-MWC on 34 graphs in Table 5. However, WLMC is slower than WC-MWC on 20 of these 34 graphs because MaxSAT reasoning is time-consuming. Overall, WC-MWC exhibits the best performance on the massive sparse graphs. This is due to the fact that UB_{WC} does not apply MaxSAT reasoning and computing UB_{WC} is usually easier and faster than computing UB_{MaxSAT}.

7. Conclusions

We proposed the new upper bound UB_{WC} for the MWCP that is based on the novel notion of weight cover. A weight cover is a set of independent sets (ISs) with a weight function for each IS that assigns a weight to each vertex occurring in the IS in such a way that the total weight of each vertex is preserved across the ISs in the weight cover. This feature of a weight cover allows its incremental construction and its application to minimize the number of branches that must be traversed in a BnB MWCP algorithm. The notion of weight cover is different from the notion of weighted IS cover in the literature. In a weighted IS cover, each IS is associated with a weight, but no weights are assigned to the vertices in the IS. Consequently, the new upper bound UB_{WC} is different from the upper bound UB_{WISC} based on weighted IS covers. It is also different from the upper bound UB_{MaxSAT} based on MaxSAT reasoning, because it does not need MaxSAT reasoning to be computed.

We developed the BnB MWCP algorithm WC-MWC that uses UB_{WC} to minimize the number of branches by incrementally constructing a weight cover for the subgraph at every search tree node. The experimental results show that UB_{WC} allows WC-MWC to reach, or even exceed, the performance of some of the best performing exact and heuristic MWCP algorithms on both small/medium graphs and real-world massive graphs. The performance of WC-MWC refutes two prevailing hypotheses in the field: (1) exact MWCP algorithms, despite proving optimality, are less adequate for large graphs than heuristic algorithms; and (2) algorithms designed for massive graphs are expected to perform worse on small graphs.

Acknowledgments

Work supported by the National Natural Science Foundation of China (grants 61272014, 61370183, 61472147 and 61370184), the platforms Matrics of University of Picardie Jules Verne and HPC of Jianghan Univeristy, the Spanish Ministry of Economy and Competitiveness–FEDER (grant TIN2015-71799-C2-1-P).

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.ejor.2018.03.020.

References

Balas, E., & Xue, J. (1991). Minimum weighted coloring of triangulated graphs, with application to maximum weight vertex packing and clique finding in arbitrary graphs. *SIAM Journal on Computing, 20*(2), 209–221.

Butenko, S., & Wilhelm, W. E. (2006). Clique-detection models in computational biochemistry and genomics. *The European Journal of Operational Research, 173*(1), 1–17.

Cai, S., & Lin, J. (2016). Fast solving maximum weight clique problem in massive graphs. In *Proceedings of the twenty-fifth international joint conference on artificial intelligence* (pp. 568–574).

Fang, Z., Li, C. M., & Xu, K. (2016). An exact algorithm based on MaxSAT reasoning for the maximum weight clique problem. *Journal of Artificial Intelligence Research, 55*, 799–833.

Garey, M. R., & Johnson, D. S. (1979). In W. H. Freeman (Ed.), *Computers and intractability: A guide to the theory of NP-completeness*.

Held, S., Cook, W. J., & Sewell, E. C. (2012). Maximum-weight stable sets and safe lower bounds for graph coloring. *Mathematical Programming Computation, 4*, 363–381.

Jiang, H., Li, C. M., & Manyà, F. (2016). Combining efficient preprocessing and incremental MaxSAT reasoning for MaxClique in large graphs. In *Proceedings of the twenty-second European conference on artificial intelligence* (pp. 939–947).

Jiang, H., Li, C. M., & Manyà, F. (2017). An exact algorithm for the maximum weight clique problem in large graphs. In *Proceedings of the thirty-first AAAI conference on artificial intelligence* (pp. 830–838).

Konc, J., & Janezic, D. (2007). An improved branch and bound algorithm for the maximum clique problem. *MATCH Communications in Mathematical and in Computer Chemistry, 58*(3), 569–590.

Li, C. M., Jiang, H., & Manyà, F. (2017). On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Computers & Operations Ressearch, 84*, 1–15.

Li, C. M., Jiang, H., & Xu, R. (2015). Incremental MaxSAT reasoning to reduce branches in a branch-and-bound algorithm for MaxClique. In *Proceedings of the ninth international conference on learning and intelligent optimization* (pp. 268–274).

Li, C. M., & Quan, Z. (2010). An efficient branch-and-bound algorithm based on MaxSAT for the maximum clique problem. In *Proceedings of the twenty-forth AAAI conference on artificial intelligence* (pp. 128–133).

Li, C. M., Zhu, Z., Manyà, F., & Simon, L. (2012). Optimizing with minimum satisfiability. *Artificial Intelligence, 190*, 32–44.

Mascia, F., Cilia, E., Brunato, M., & Passerini, A. (2010). Predicting structural and functional sites in proteins by searching for maximum-weight cliques. In *Proceedings of the twenty-forth AAAI conference on artificial intelligence* (pp. 1274–1279).

Rossi, R., & Ahmed, N. (2015). The network data repository with interactive graph analytics and visualization. In *Proceedings of the twenty-ninth AAAI conference on artificial intelligence* (pp. 4292–4293).

San Segundo, P., Lopez, A., & Pardalos, P. M. (2016). A new exact maximum clique algorithm for large and massive sparse graphs. *Computers & Operations Research, 66*, 81–94.

San Segundo, P., Matía, F., Rodríguez-Losada, D., & Hernando, M. (2013). An improved bit parallel exact maximum clique algorithm. *Optimization Letters, 7*(3), 467–479.

San Segundo, P., Nikolaev, A., & Batsyn, M. (2015). Infra-chromatic bound for exact maximum clique search. *Computers & Operations Research, 64*, 293–303.
San Segundo, P., & Tapia, C. (2014). Relaxed approximate coloring in exact maximum clique search. *Computers & Operations Research, 44*, 185–192.
Tomita, E., Sutani, Y., Higashi, T., Takahashi, S., & Wakatsuki, M. (2010). A simple and faster branch-and-bound algorithm for finding a maximum clique. In *Proceedings of the forth international workshop on algorithms and computation* (pp. 191–203).
Wang, Y., Cai, S., & Yin, M. (2016). Two efficient local search algorithms for maximum weight clique problem. In *Proceedings of the thirtieth AAAI conference on artificial intelligence* (pp. 805–811).
Warren, J. S., & Hicks, I. V. (2006). Combinatorial branch-and-bound for the maximum weight independent set problem. Master's thesis. Texas A&M University. Tech Rep.
Wu, Q., & Hao, J. (2015a). A review on algorithms for maximum clique problems. *The European Journal of Operational Research, 242*(3), 693–709.
Wu, Q., & Hao, J. (2015b). Solving the winner determination problem via a weighted maximum clique heuristic. *Expert Systems with Applications, 42*(1), 355–365.
Zhang, D., Javed, O., & Shah, M. (2014). Video object co-segmentation by regulated maximum weight cliques. In *Proceedings of the thirteenth European conference on computer vision* (pp. 551–566).
Zhian, H., Sabaei, M., Javan, N. T., & Tavallaie, O. (2013). Increasing coding opportunities using maximum-weight clique. In *Proceedings of the fifth computer science and electronic engineering conference* (pp. 168–173).
Zhou, Y., Hao, J., & Goëffon, A. (2017). PUSH: a generalized operator for the maximum vertex weight clique problem. *The European Journal of Operational Research, 257*(1), 41–54.