A clique-based exact method for optimal winner determination in combinatorial auctions

Qinghua Wu, Jin-Kao Hao

A clique-based exact method for optimal winner determination in combinatorial auctions

Qinghua Wu [a], Jin-Kao Hao [b,c,*]

[a] School of Management, Huazhong University of Science and Technology, No. 1037, Luoyu Road, Wuhan, China [b] LERIA, Université d'Angers, 2 bd Lavoisier, 49045 Angers, France
[c] Institut Universitaire de France, Paris, France

Given a set of items to sell and a set of combinatorial bids, the Winner Determination Problem (WDP) in combinatorial auctions is to determine an allocation of items to bidders such that the auctioneer's revenue is maximized while each item is allocated to at most one bidder. WDP is at the core of numerous relevant applications in multi-agent systems, e-commerce and many others. We develop a clique-based branch-and-bound approach for WDP which relies on a transformation of WDP into the maximum weight clique problem. To ensure the efficiency of the proposed search algorithm, we introduce specific bounding and branching strategies using a dedicated vertex coloring procedure and a specific vertex sorting technique. We assess the performance of the proposed algorithm on a large collection of benchmark instances in comparison with the CPLEX 12.4 solver and other approaches. Computational results show that this clique-based method constitutes a valuable and complementary approach for WDP relative to the existing methods.

1. Introduction

Combinatorial auctions (CAs) allow bidders to buy entire bundles of goods (or items) in a single transaction [6]. One key issue in CAs is the winner determination problem (WDP) [18]. Given a set of combinatorial bids, each bid being defined by a subset of items with a price, two bids are conflicting if they share at least one item. WDP is to determine a conflict-free allocation of items to bidders (the auctioneer can keep some of the items) such that the auctioneer's revenue is maximized.

In terms of computational complexity, WDP is known to be NP-hard [26]. From the practical point of view, WDP is at the core of a number of relevant applications like cloud computing [27], electronic commerce [40], intelligent transportation systems [31,40], logistics services [40] and production management [25]. The computational challenge of WDP and its practical relevance have motivated the development of a variety of solution approaches in recent years, including both heuristic and exact methods. We provide a review of the main existing methods in the literature in Section 2.

In this paper, we are interested in solving WDP exactly using a clique-based approach. Indeed, it is known that WDP is equivalent to the weight set packing problem [40], and can be reduced to the maximum weight clique problem (MWCP). The first study on the clique-based approach for WDP was explored very recently in [37] where a heuristic is applied to approximate the transformed MWCP problem. In this work, we explore an exact approach with an effective branch-and-bound algorithm

* Corresponding author at: LERIA, Université d'Angers, 2 bd Lavoisier, 49045 Angers, France . Tel.: +33241735076; fax: +33241735073.
E-mail addresses: huawuqing1005@163.com (Q. Wu), hao@info.univ-angers.fr, jin-kao.hao@univ-angers.fr (J.-K. Hao).

(called *MaxWClique*). To get tight upper bounds on the maximum weight clique, we devise a dedicated vertex coloring heuristic which groups vertices of the largest possible weight into a same color class. In vertex coloring, vertices in a graph are assigned a color such that pairwise adjacent vertices are colored differently. The sum of the weights of the color classes produced in the process is an upper bound to the maximum weight clique in the graph. In addition, to prune the search tree effectively, the algorithm employs a global branching rule by presenting the vertices to the coloring procedure in a non-increasing weight order to obtain tight bounds.

The rest of this paper is organized as follows. In Section 2, we provide a literature review of the most representative approaches for WDP as well as MWCP and summarize the main contributions of our work. In Section 3, we establish the connections between the winner determination problem and the maximum weight clique problem. In Section 4, we present the clique-based branch-and-bound algorithm for MWCP (and WDP). In Section 5, we provide computational results of extensive experiments on three sets of WDP benchmark instances in the literature. In Section 6, we provide some insights on the performance of the proposed approach and discuss the classes of WDP instances most suitable for our clique-based approach. Perspectives and concluding remarks are provided in Sections 7 and 8 respectively.

2. Literature review and main contributions

In this section, we provide a literature review on the most representative approaches for WDP and MWCP, followed by a summary of the main contributions of our work.

2.1. Literature review on algorithms for the winner determination problem

The computational challenge of WDP and its wide practical applications have motivated a variety of solution approaches in the literature, including both heuristic and exact methods.

Heuristic methods are designed to find approximate solutions within acceptable computing time limits, but without provable optimal guarantee of the attained solutions. These methods are often applied when an optimal solution cannot be achieved or is not required. Some representative heuristic algorithms for WDP include a stochastic local search method (Casanova) [15], a hybrid algorithm combining simulated annealing with branch-and-bound (SAGII) [8], a hybrid genetic algorithm [3], a crossover-based tabu search algorithm [33] and a multi-neighborhood tabu search algorithm [37] which explores the cliquebased approach from a heuristic perspective.

On the other hand, considerable effort has been devoted to developing various exact methods for WDP. Attempts to apply exact methods to solve WDP (under the name of set packing) can be found as early as in the beginning of 1970s [23]. Many other solution methods have appeared in the literature ever since. Most exact algorithms are based on the general branch-and-bound (B&B) framework and branch on bids to find optimal allocations. Representative examples include the combinatorial auction structural search (CASS) [10], the Combinatorial Auction Multi-Unit Search (CAMUS) [19], the BOB algorithm [29], the CABOB algorithm [30], and the linear programming based B&B algorithm [21]. These B&B methods differ from each other mainly by (1) specific techniques to determine the lower and upper bounds, (2) their branching strategies and (3) some other techniques like preprocessing, decomposition of the bid graph, and identifying and solving tractable special cases. Especially, the upper-bounding methods play a key role to the performance of these B&B algorithms, and a typical upper-bounding method uses linear programming relaxations of the set packing formulation [21,30]. In addition, other mathematical formulations for WDP have also been studied within a branch-and-cut algorithm [7], a branch-and-price algorithm [9] and a dynamic programming algorithm [26]. However, these last methods do not seem to perform better than the integer linear programming CPLEX solver using a natural formulation of the problem, which indeed shows an excellent performance in many cases [1,8,30].

2.2. Literature review on algorithms for the maximum weight clique problem

Though various exact algorithms have been proposed for the unweighted case of the maximum clique problem (see e.g., [38]), MWCP is somewhat less studied in the literature. Yet, several exact algorithms have been proposed to solve this problem.

The B&B algorithm proposed by Östergård [22] (called *Cliquer*) is among the most popular and influential MWCP algorithms. *Cliquer* relies on an iterative deepening strategy similar to dynamic programming for bounding. Given an undirected graph $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$. The algorithm starts with the smallest subgraph containing only the last vertex in V and then iteratively finds a maximum weight clique for subgraphs $V_n = \{v_n\}$, $V_{n-1} = \{v_{n-1}, v_n\}$, $V_{n-2} = \{v_{n-2}, v_{n-1}, v_n\}, \ldots$. This process ends up with the last subgraph V_1 which is the original graph to be solved and returns the maximum weight clique found. During the backtrack search of *Cliquer*, the information obtained in previously computed smaller graphs is used for better upper bounds for larger graphs. The performance of *Cliquer* greatly depends on the initial ordering of V. In *Cliquer*, vertices are sorted in descending order of weights, and vertices with the same weights are sorted by descending order of the sum of weights of adjacent vertices.

In [11], Kumlander proposed an exact algorithm based on a heuristic vertex coloring and a backtrack search for MWCP. The first step of this algorithm is to obtain a vertex coloring $c = \{C_1, C_2, \ldots, C_k\}$ of the graph $G = \{V, E\}$ and reorder the vertices first by color classes and then by weights inside each color class in ascending order. Then during the search process of the algorithm, this vertex coloring is frequently used to prune branches of the maximum weight clique search tree, since the vertex coloring upper bound computed as $\sum_{i=1}^{k} max\{w(u)|u \in C_i \cap S\}\}$ can be served as a more precise estimation on the bound of the subproblem S. A

backtrack search similar to *Cliquer* is also used to prune the search tree. With these two pruning strategies, this algorithm is able to prune subproblems more effectively than Östergård's algorithm.

Like Östergård's *Cliquer* algorithm, the performance of Kumlander's algorithm greatly depends on the initial ordering of the vertices. In [12], a new sorting and coloring strategy was proposed. In [34], some further improvements were introduced, including some new ordering methods for greedy coloring, a strategy to limit color class sizes and a new implementation technique for the computation of coloring upper bounds. Finally, an edge orienting based exact algorithm is presented in [39].

2.3. Main contribution of our work

In this paper, we develop a new B&B algorithm for WDP which relies on a transformation of WDP into the maximum weight clique problem. Especially, we devise a coloring based upper-bounding method which leads to a faster completion of the search algorithm than using the traditional linear programming upper-bounding method in many cases. In addition, the coloring based method is also employed by the branching strategy to guide the choice of bids (vertices) during the tree search process. Experiments show that our clique-based approach is particularly effective for the class of WDP instances with many items per bid. The main contributions of this work can be summarized as follows.

First, this is the first study using an *exact* MWCP algorithm to solve WDP. Even though the relation between WDP and MWCP is known in the literature, the clique-based approach for WDP was explored only very recently in [37] by applying a heuristic approach to approximate the WDP problem. In this work, we further explore the clique-based approach and solve the WDP problem exactly with a B&B algorithm. To ensure its effectiveness, the proposed *MaxWClique* algorithm integrates some original features to update its lower and upper bounds. The proposed exact method not only has the theoretical advantage of guaranteeing the optimality of the solution found, and sometimes is even much faster than the clique-based heuristic approach.

Second, we report extensive computational results on three test suites of popular WDP benchmark instances with very different characteristics. We compare our results with the powerful CPLEX 12.4 solver which is known to be a highly effective tool for WDP in many cases. This study discloses that the clique-based approach and the IP solvers like CPLEX constitute two complementary solution methods and can be advantageously used in a joint manner to exactly solve different classes of WDP instances.

Third, from the perspective of solving MWCP, we explore new bounding and branching strategies based on vertex coloring within our B&B algorithm. Though vertex coloring has been frequently applied to exactly solve the unweighted maximum clique problem (see [38] for more details on this issue), for the weighted case (i.e., MWCP), this idea has only been formally explored in [11] where the initial graph is colored (once for all) before the B&B routine starts and the resulting coloring is used on the permanent base throughout the search. This strategy has the main advantage of running the coloring algorithm only once. However, since the clique algorithm manipulates many and *different* subgraphs of the initial graph G, the coloring for G is not necessarily appropriate for bound estimation of these reduced subgraphs.

In our work, we propose a new vertex coloring based algorithm which applies repeatedly a (fast) coloring algorithm to different subgraphs at different nodes of the search tree. Our method makes it possible to obtain tighter bounds of clique weight of the subgraphs, though coloring multiple graphs may be somewhat time consuming. Furthermore, as observed in [22,34], the search tree is pruned more effectively when the vertices of the initial graph are sorted in descending order of vertex weights. Moreover, it is known that the upper bound of the maximum weight of the clique in the subgraph will decrease faster when the vertex is always picked from the color class with the smallest color number [38]. As a consequence, we introduce a branching strategy which first sorts the vertices by color numbers in increasing order, and inside a color class by vertex weights in decreasing order and then always takes the vertices to join the clique in the sorted order. As we show in Section 6.2, equipped with our vertex coloring based bounding and branching strategies, our algorithm competes very favorably with the reference MWCP algorithms, confirming the value of our adopted bounding and branching strategies.

3. Winner determination and maximum weight clique

Our approach exploits the strong connection between the winner determination problem and the maximum weight clique problem to develop an exact approach for WDP. We first define the winner determination problem and then show its transformation to the maximum weight clique problem.

3.1. The winner determination problem (WDP)

Let $M = \{1, 2 \ldots m\}$ be a set of m items to be auctioned and $B = \{B_1, B_2 \ldots B_n\}$ a set of n bids. A bid B_j is a pair (S_j, P_j) where $S_j \subset M$ is a set of items, and P_j is the price of B_j ($P_j > 0$). Let a_{mn} be a matrix with m rows and n columns where $a_{ij} = 1$ if item $i \in S_j$, $a_{ij} = 0$ otherwise. Furthermore, define a decision variable for each bid B_j such that $x_j = 1$ if bid B_j is accepted (a winning bid), and $x_j = 0$ otherwise (a losing bid). Then, WDP, which concerns finding an allocation of items to bidders to maximize the auctioneer's revenue under the constraint that each item is allocated to at most one bid (some items may remain unassigned), can be modeled as the following integer program:

$$\text{Maximize} \sum_{j=1}^{n} P_j x_j \tag{1}$$

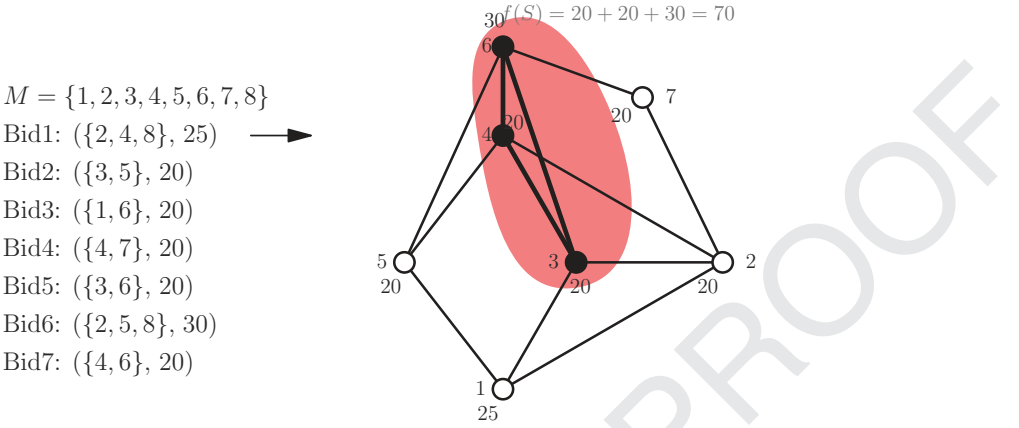

Fig. 1. The original MDP instance (left) and the transformed maximum weight clique instance $G = (V, E, W)$ (right) [37]. The maximum weight clique {3, 4, 6} on the graph leads to the three winning bids S = {Bid3, Bid4, Bid6} with a maximum revenue $f(S) = 70$.

$$\text{s.t.} \sum_{j=1}^{n} a_{ij} x_j \leq 1, i \in \{1..m\} \tag{2}$$

$$x_j \in \{0, 1\} \tag{3}$$

The above integer program model corresponds to a set packing problem [40], which can be reduced to the maximum weight clique problem.

3.2. From WDP to the maximum weight clique problem

Let $G = (V, E, W)$ be an undirected weighted graph where V denotes the set of vertices, E the set of edges and W the vertex weighting function that assigns a positive real number (weight) w_i to each vertex $i \in V$. A subset $C \subseteq V$ is a clique if every two vertices in C are connected by an edge. The maximum weight clique problem is to find a clique C with a maximum weight, which is defined as the sum of the weights of all the vertices in C, i.e., $W(C) = \sum_{i \in C} w_i$.

Given a WDP instance defined by a collection of bids $B = \{B_1, B_2 \ldots B_n\}$, each bid B_j being specified by its set of items S_j and its associated price P_j, we can transform the WDP instance into a MWCP instance $G = (V, E, W)$ using the following method. Each vertex $j \in V$ in the graph corresponds to a bid $B_j \in B$, its weight w_j is given by the price P_j of bid B_j. For any two vertices i and j in $G = (V, E, W)$, they are connected by an edge if and only if the corresponding item sets S_i and S_j share no common item, i.e., $S_i \cap S_j = \emptyset$, implying that the two corresponding bids B_i and B_j can be accepted together as winning bids. Now it is easy to observe that $C = \{i_1, \ldots, i_r\}$ is a maximum weight clique in the graph $G = (V, E, W)$, if and only if $\{B_{i_1}, \ldots, B_{i_r}\}$ are the r optimal winning bids with a maximum revenue for the corresponding WDP instance (see example of Fig. 1 from [37]).

Thus, in order to determine the winning bids for a given WDP instance, we only need to find a maximum weight clique in the corresponding graph $G = (V, E, W)$. For this purpose, we design a branch-and-bound algorithm for the maximum weight clique problem which is presented in the next section.

4. *MaxWClique*: a branch-and-bound algorithm for MWCP

4.1. The basic procedure

Branch-and-bound is one of the most successful paradigms for designing exact algorithms for MWCP (as well as its unweighted case, i.e., the maximum clique problem where the vertex weight is equal to 1) [11,12,22,38]. The success of a B&B algorithm mainly relies on the use of refined techniques for determining lower and upper bounds on the weight (or size) of the clique, and the proper branching strategies. Especially, vertex coloring techniques are frequently employed and proven to be effective for these purposes [11,12,38]. Our coloring based B&B algorithm *MaxWClique* for the maximum weight clique problem is based on and generalizes the procedure in [4] which was designed for the classical *unweighted* maximum clique problem. Moreover, our algorithm examines the graph relying on the standard B&B framework while the algorithms in [11,22] employs a backtracking search technique which examines the graph in the opposite order of a standard B&B algorithm.

The presentation of our *MaxWClique* algorithm (see Algorithm 1) follows the general recursive backtracking framework adopted by many other exact B&B algorithms for the *unweighted* maximum clique problem such as BB-MaxClique [32], MCQ

Algorithm 1 The branch-and-bound algorithm for the maximum weight clique problem.

Require: A weighted graph $G = (V, E, W)$
Ensure: The maximum weight clique C^* and its weight $W(C^*)$
/* C and C^* are two global variables designating respectively the currently growing clique and the largest weight clique found so far */
1: **Function Main**
2: $C \leftarrow \emptyset$ /* the clique currently under construction */
3: $C^* \leftarrow \emptyset$ /* the maximum weight clique found so far */
4: *ColorSort*$(V, ColV)$ /* color and resort vertices in V (Section 4.2) */
5: *MaxWClique*$(V, ColV)$
6: *return* C^* and $W(C^*)$
7: **End function** to provide a review of different solution approaches approaches approaches proposed in
8: **Function MaxWClique(set** P**, set** $ColP$**)**
/* P is the candidate set containing the vertices that can be added to C, vertices in P are sortedby non-decreasing order with respect to their color numbers, and inside color class sorted in non-increasing weights, $ColP$ is an array containing the color number of each vertex in P */
9: **if** ($P = \emptyset$ and $W(C) > W(C^*)$) **then**
10: $C^* \leftarrow C$ /* update the maximum weight clique found so far */
11: **End if**
12: **while** $P \neq \emptyset$ **do**
13: Compute the upper bound for the subgraph induced by P as $UB(P) = \sum_{i=ColP[1]}^{ColP[|P|]} W(I_i)$ where $W(I_i) = max\{W(v) : v \in I_i\}$ (Section 4.2)
14: **if** ($W(C) + UB(P) > W(C^*)$) **then**
15: Select the first vertex p in P / * branching rule (Section 4.3) */
16: Save set P and $ColP$
17: $C \leftarrow C \bigcup \{p\}$ /* expand C by adding p */
18: $P' \leftarrow P \bigcap N(p)$ /* remove the vertices not connected to p from P */
19: *ColorSort*$(P', ColP')$ /* resort vertices in the new candidate set P' */
20: *MaxWClique*$(P', ColP')$ /* go to the next level of recursion */
21: Restore P and $ColP$ /*step back from the precedent level of recursion*/
22: $C \leftarrow C \setminus \{p\}$ /* remove p from C, then try the next vertex in P */
23: $P \leftarrow P \setminus \{p\}$ /* continue to examine the left vertices in P (see line 12) */
24: $ColP \leftarrow ColP \setminus \{ColP[1]\}$ /* remove the color of p from $ColP$ */
25: **else**
26: return
27: **End if**
28: **End while**
29: **End function**

[35], MCS [36] and MaxCliqueDyn [13]. The proposed *MaxWClique* algorithm replies on two key vertex sets: the current clique C (also called solution) and the candidate vertex set P. C is a global set and designates the clique currently under construction while P is a subset of $V \backslash C$ such that $v \in P$ if and only if $\forall u \in C, \{u, v\} \in E$. In other words, each vertex of P must be connected to *all* the vertices of the current clique C. Let $N(v)$ be the set of the vertices adjacent to vertex v, then P can equivalently be defined by $P = \cap_{v \in C} N(v)$. Given the property of P, it is clear that any vertex v of P can be added to C to obtain a larger clique $C' = C \cup \{v\}$. This property constitutes one of the key foundations of Algorithm 1.

Starting with an empty clique $C = \emptyset$ and $P = V$ (see Algorithm 1, lines 2 and 5), the algorithm operates by recursively calling the function *MaxWClique* and uses a global variable C^* to maintain the largest weight clique found so far ($W(C^*)$ is thus the current *lower bound* of the maximum weight clique of $G = (V, E, W)$). At each recursion of the function *MaxWClique*, a vertex v is selected among the vertices in P to expand the current clique C. On backtracking, v is removed from C and P, and a new vertex is selected from P to expand C by calling again *MaxWClique* (see Algorithm 1, lines 21–24 and line 12).

Precisely, given the current clique C and its corresponding candidate set P, we use a coloring based method (see Section 4.2) to compute an *upper bound* for the subgraph induced by P (denoted by $UB(P)$). If $W(C) + UB(P) \leq W(C^*)$, C cannot lead to a clique with a weight larger than the weight of C^*, and thus the associated subtree can be safely pruned. Otherwise, the subtree rooted at the clique C needs to be further explored. In this case, a branching strategy is employed to determine the next vertex $v \in P$ to be selected to expand the current clique C (Algorithm 1, line 15). After the vertices in P are sorted according to its coloring result by the ColorSort procedure, we select the vertices in P in that order. After each branching step, P is updated by $P = P \cap N(v)$ (Alg. 1, line 19) to make sure that the required property of the set P is always verified.

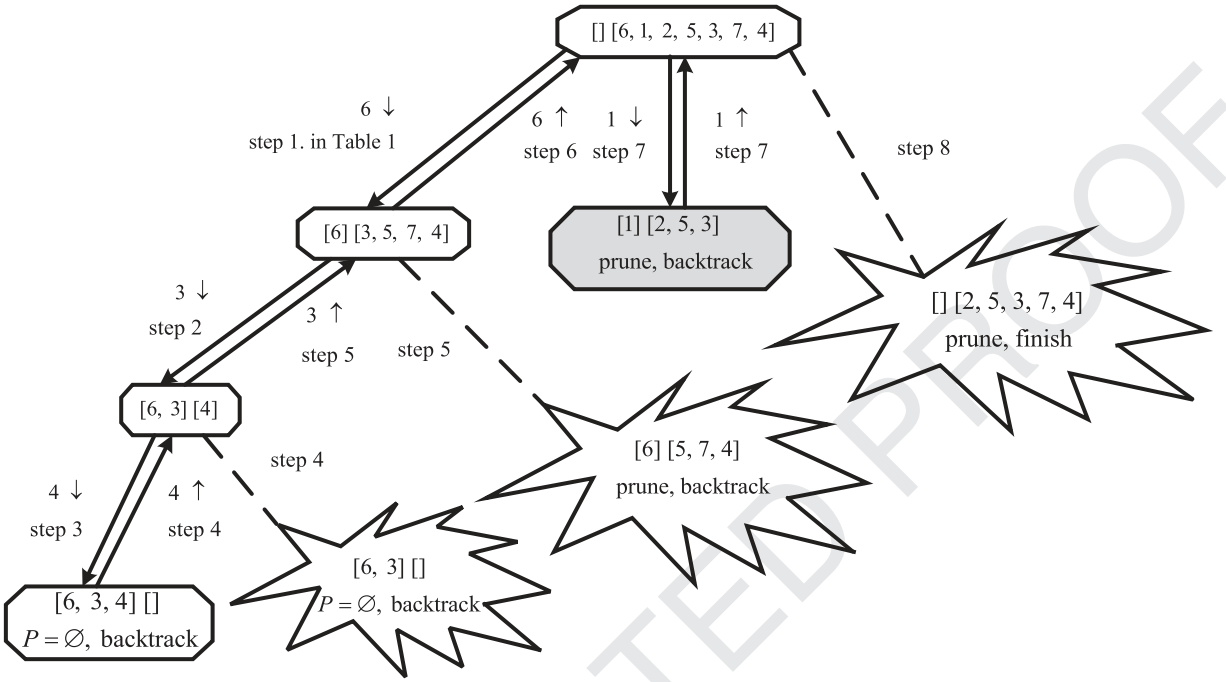

Fig. 2. The search tree of the *MaxWClique* algorithm applied to the example of Fig. 1. More details about steps 1 to 8 can be found in Table 1.

MaxWClique implicitly enumerates the cliques of the graph $G = (V, E, W)$ according to some predefined order (by color classes). Initially, after resorting the vertices by color classes, *MaxWClique* finds the largest clique C_1 that contains the vertex v_1. Then it finds C_2, the largest clique in $G - \{v_1\}$ that contains v_2 and so on. Without the pruning strategy in line 14, the algorithm will go through every maximal clique in the graph (A maximal clique is a clique that is not contained in a larger clique, which corresponds to a leaf node in the search tree). By applying the pruning techniques in line 14, we only safely prune some branches of the search tree which cannot lead to an optimal solution (this is guaranteed by the bounding condition $W(C) + UB(P) \leq W(C^*)$). Given the current clique C and its candidate vertex $P = \cap_{v \in C} N(v)$, we use a heuristic branching strategy to decide the order of adding the vertices of P to C. This is done by first sorting the vertices in P according to their color classes and then adding vertices in P in the sorted order.

Crucial to the understanding of our exact algorithm is the notion of search depth and recursive calling of function *MaxWClique*. Given the current clique C and its candidate set $P = \cap_{v \in C} N(v)$, function *MaxWClique(P, ColP)* aims at finding the maximum weight clique containing all vertices of C by adding the vertices in P. To achieve this, *MaxWClique(P, ColP)* examines the vertices in P one by one. Each time a vertex v in P is added to the current clique C, we continue to search the maximum weight clique containing all vertices of $C = C \cup \{v\}$ by adding the vertices in $P' = P \bigcap N(v)$. For this purpose, we recursively call *MaxWClique(P′, ColP′)* and move to the next level of search depth. Before going to the next level of search depth, we record the vertices of P in a $|V| \times K$ matrix MP where MP_i is used to store the vertices of P in the $(i+1)$-*th* ($i = |C|$) level of recursion (see also the space complexity analysis in Section 4.4). Then the vertices of P' will be further examined one by one by each new call to function *MaxWClique*. When a vertex is examined, it is removed from P'. Backtracking is invoked when $P' = \emptyset$ or the pruning condition is satisfied (see Algorithm 1, lines 12 and 14 as well as Fig. 2 for an illustrative example). After the return of *MaxWClique(P′, ColP′)*, we recover the vertices in P, backtrack to this level of search depth (see Algorithm 1, line 21), and continue to examine the remaining vertices in P. For this, vertex v is deleted from the depth and the next vertex of the depth becomes active and will be expanded (see Algorithm 1, lines 22–24 and line 12). This procedure continues until $P = \emptyset$ or the pruning condition $W(C) + UB(P) \leq W(C^*)$ is satisfied.

Note that the heuristic procedure *ColorSort*(*P*, *ColP*) only changes the order of the vertices in P (sort the vertices in P according to vertex coloring results), it does not remove any vertex from P nor add any new vertex to P. After each call to function *ColorSort*(*P*, *ColP*), the order of vertices in P is fixed. Then the vertices in P are examined in the sorted order. In this way, we can guarantee the unexamined vertices are always kept in P.

Using the sorting heuristic for P and the pruning rule $W(C) + UB(P) \leq W(C^*)$, our *MaxWClique* algorithm accelerates the enumeration process of the whole search space without missing any possible candidate solution, ensuring the completeness of the search procedure.

Two key components of our B&B algorithm are thus the strategy used to determine the upper bound on the maximum weight clique in the subgraph induced by P, and the branching strategy to determine the next vertex $v \in P$ to be added to the current clique C. In the following subsections, we formally describe these strategies.

4.2. A coloring based bounding strategy

For MWCP, the idea of using vertex coloring to estimate the upper bound for the maximum clique weight in a subgraph was first explored in [11] where the initial graph is colored once for all. Given an undirected graph $G = (V, E)$, a k-coloring of G is a partition of V into k independent sets (color classes). An independent set of G is a subset I of V such that no two vertices in I are connected by an edge. The graph coloring problem is to determine the smallest integer k (its chromatic number $\chi(G)$) such that there exists a k-coloring of G. Given a coloring $c = \{I_1, \ldots, I_k\}$ of G, we define, for each color class I_i of c, its weight as the maximum weight of the vertices in I_i, i.e., $W(I_i) = max\{W(v) : v \in I_i\}$.

For a given undirected weighted graph $G' = (V, E, W)$ and a given k-coloring $c = \{I_1, \ldots, I_k\}$ of G', since two vertices in a clique cannot belong to the same color class of c, at most one vertex in a color class can take part in the construction of a clique. Consequently, the maximum weight of the clique in G (denoted by $W(G')$) is bounded by the sum of the weights of the color classes induced by c, i.e.,

$$W(G') \leq \sum_{i=1}^{k} W(I_i) \tag{4}$$

From the above formulation, we observe that the quality of the upper bound depends on the k-coloring. In other words, to achieve tight upper bounds, it is better to use a coloring (for the subgraph induced by the current P) such that the sum of the weights of its color classes is as small as possible. On the other hand, since we must color the associated subgraph induced by P after each iteration of our algorithm, the coloring procedure needs to be fast enough. To fulfill these purposes, we develop in this paper a fast and effective greedy procedure to color the subgraph induced by P.

The basic idea of our greedy coloring procedure is to put the vertices with the largest possible weights into the same color class. This strategy could generally reduce the sum of the weights of the color classes such that a tighter upper bound can be obtained (see also [12]). The coloring procedure constructs sequentially the color classes one by one. At the start of the coloring procedure, all vertices in the subgraph are sorted in descending order with respect to their weights. To build the first color class I_1 which is initially an empty set, we first copy all the vertices of the subgraph into a vertex set U. Then at each step of building I_1, we take the first vertex $v \in U$ (the vertex with the largest weight in U), add it to I_1 and finally remove all vertices from U which are adjacent to v. The process continues until the vertex set U becomes empty. At this point, we finish the construction of the color class I_1. To build the class I_2, we remove from the initial subgraph all the vertices of I_1 and run the same procedure on the reduced graph. The coloring procedure ends when all vertices in the subgraph induced by P have been assigned to a particular color class. Algorithm 2 summarizes this greedy coloring procedure.

Algorithm 2 The coloring procedure: *ColorSort*(P, *ColP*).

Require: A weighted subgraph $G' = (V', E', W)$ induced by P
Ensure: The resorted vertex set P and its coloring result *ColP*
1: **Begin**
2: $k = 1$ {color number counter }
3: $l = 1$ {vertex counter}
4: Sort the vertices in V' in a descending order with respect to their weights
5: **while** $V' \neq \emptyset$ **do**
6: $U \leftarrow V'$
7: $I_k = \emptyset$
8: **while** $U \neq \emptyset$ **do**
9: Select the first vertex $v \in U$
10: $I_k = I_k \bigcup \{v\}$
11: $U = U \backslash \{v\}$
12: Remove all vertices which are adjacent to v from U
13: **end while**
14: $k = k + 1$
15: $V' = V' \backslash I_k$
16: **end while**
17: **for** $c = 1$ **to** k **do**
18: **for** $i = 1$ **to** $|I_c|$ **do**
19: $P[l] = I_c[i]$ /* resort the vertices in P in an increasing order with their color numbers */
20: $ColP[l] = c$ /* store the color number of each vertex $p \in P$ in *ColP* */
21: $l = l + 1$
22: **end for**
23: **end for**
24: **End**
25: Return the resorted vertex set P and its coloring result *ColP*

Finally, for an efficient implementation of the coloring procedure, we adopt a bit-parallel technique proposed in [32]. We first encode the adjacency matrix of the graph, as well as some vertex sets such as P and U into bit strings. Based on this binary encoding, we make full use of bitwise operations in parallel to accelerate a number of computations such as sorting vertices in P by weights (line 4, Algorithm 2) and computing graph transitions.

4.3. Branching

In step 15 of Algorithm 1, we need a branching rule to select the next vertex from the candidate set P to expand the current clique. To make this choice, we resort to the vertices in P based on their coloring and weight information.

Precisely, after a coloring $c = \{I_1, \ldots, I_k\}$ of P is obtained using the greedy coloring procedure (see Section 4.2), all vertices are copied back to the input candidate set P as they appear in the color classes and in increasing order with respect to index k (lines 17–23, Algorithm 2). Thus, vertices in P are sorted by non-decreasing order with respect to their color numbers, and in non-increasing weights inside color class. Then we take vertices in P to join the clique C in the sorted order i.e., we always select the first vertex $v \in P$ (the vertex with the maximum weight in the color class with the smallest color number) to expand the clique under construction. On backtracking, the algorithm deletes v from P and picks a new vertex from P (the first vertex in P after the removal of v). This process is repeated until the pruning condition ($W(C) + UB(P) \leq W(C^*)$) is verified. The upper bound for the maximum weight of the clique in the remaining subgraph after the removal of v can be easily computed as $UB(P) = W_{v'} + \sum_{i=c(v)+1}^{k} W(I_i)$ where v' is the next vertex in the same color class as v and $c(v)$ is the color number of v.

This branching strategy has several advantages (see also the analysis of Section 6.4). First, the number of color classes of the coloring for the remaining subgraph to be searched tends to be reduced more quickly by always selecting the vertices from the first color class. Second, by preferring to add first the vertices with larger weights to the clique, good solutions are generated earlier on. This strategy leads to tighter lower bounds and thus reduces the size of the search tree. Third, the weight of the color class which is currently under consideration tends to decrease more quickly by choosing the vertex with the maximum weight in this color class. Thus, our pruning rule may lead to a fast decrease to the upper bound of the maximum weight of the clique in the remaining subgraph to be searched, thus reducing the search space on average. In Section 6.4, we will provide experimental evidences to confirm these advantages.

4.4. Complexity analysis

In this section, we undertake a (time and space) complexity analysis of our *MaxWClique* algorithm. To establish the time complexity, we first analyze each recursive search step of the algorithm (line 12–26, Algorithm 1) and then give the time complexity of the whole algorithm. Each recursive search step implies two main procedures for computing the upper bound for the subgraph induced by P (line 13, Algorithm 1) and the greedy coloring procedure (line 20, Algorithm 1). For the upper bound computation, since the vertices in P are already sorted in non-decreasing order with respect to their color numbers, and inside color class sorted in non-increasing weights, this procedure can be achieved in $O(|V|)$ by adding up the weights of the first vertex in each color class. For the coloring procedure, the time complexity is $O(|V|^2)$, since it can be achieved in $|P'|$ steps, each coloring step assigning a color to a vertex with a time complexity of $O(|V|)$. Other procedures like computing $P^{'} \leftarrow P \bigcap N(p)$ at each recursive search step can also be completed in $O(|V|)$. Thus, each recursive search step of the algorithm requires no more than $O(|V|^2)$ time. For the whole algorithm, since the maximum search depth of our algorithm is K, where K is the number of color classes required to color the initial graph G, the maximum number of the nodes in the search tree of *MaxWClique* is 2^K. Thus, the worst-case time complexity of our algorithm is $O(|V|^2 2^K)$. Obviously, like other exact MWCP and MCP algorithms [2,38], our algorithm suffers from an exponential time complexity.

On the other hand, *MaxWClique* is space efficient with a space complexity of $O(|V|^2)$. First, to store the graph G of a MWCP instance, we use a binary matrix M of size $|V| \times |V|$, where $M_{ij} = 1$ if vertices i and j are adjacent, $M_{ij} = 0$ otherwise. In addition, when *MaxWClique* goes from the current level of recursion (say recursion-level i) to the next level of recursion (say recursion-level $i+1$), all vertex sets P in recursion levels from 1 to $i+1$ need to be stored. This can be achieved efficiently with a $|V| \times K$ matrix MP, where K ($K \leq |V|$) is the maximum search depth of our algorithm (see also the time complexity analysis). Indeed, when *MaxWClique* returns from recursion-level $i+1$ to recursion-level i, the space for recursion-level $i+1$ is reused and the vertex set P at recursion-level i only needs to be updated by removing the first vertex from P. Thus, in order to store the sets P at different levels of recursions, a $|V| \times K$ matrix suffices. Similarly, another two-dimension array of size $|V| \times K$ is also required to store the coloring of P (i.e., *ColP*) at each level of recursion which is also reused. Therefore, the total space requirement of *MaxWClique* is bounded by $O(|V|^2)$.

In addition to the above worst case complexity analysis, a more useful study in practice is to investigate the empirical scaling behavior of run-time of the proposed algorithm [14]. Such an analysis will shed light on how the algorithm scales on instances of various types and sizes, and thus constitutes an interesting research topic in the future.

4.5. A working example

In this section, we show an example to illustrate how the proposed algorithm works using the graph G in Fig. 1. Initially, $C = \emptyset$ and $C^* = \emptyset$, and the main steps of the algorithm *MaxWClique* (see Algorithm 1) to attain the maximum weight clique {3, 4, 6} is summarized in Table 1.

Table 1
The main steps of the *MaxWClique* algorithm applied to the example of Fig. 1.

Step	Depth of recursion	C	P	Algorithm process
Initially	Depth 1	$\emptyset$	{6, 1, 2, 5, 3, 7, 4}	Vertices in P are colored and re-sorted by non-decreasing order with respect to their color numbers, and inside color class sorted by non-increasing weights
Step 1	Depth 2	{6}	{3, 5, 7, 4}	Search the largest weight clique that contains vertex 6 in G, choose vertex 6 as the branching vertex and add it to clique C, color and resort candidate set P
Step 2	Depth 3	{6, 3}	{4}	Choose vertex 3 as the branching vertex and add it to clique C, color and resort candidate set P
Step 3	Depth 4	{6, 3, 4}	$\emptyset$	Add 4 to clique C, since $P = \emptyset$ and $W(C) > W(C^*)$, C^* and $W(C^*)$ are updated as $C^* = \{3, 4, 6\}$ and $W(C^*) = 70$, respectively. $P = \emptyset$, return to the precedent level of recursion
Step 4	Depth 3	{6, 3}	$\emptyset$	$P = \emptyset$, return to the precedent level of recursion
Step 5	Depth 2	{6}	{5, 7, 4}	On backtracking, remove the already expanded vertex 3 from P, since $W(C) + UB(P) = 30 + 40 \leq W(C^*)$, prune and return to the precedent level of recursion
Step 6	Depth 1	$\emptyset$	{1, 2, 5, 3, 7, 4}	Since $W(C) + UB(P) = 0 + 85 > W(C^*)$, select vertex 1 as the branching vertex
Step 7	Depth 2	{1}	{2, 5, 3}	Search the largest weight clique in $G - \{6\}$ that contains vertex 1, add vertex 1 to clique C, color and resort P, since $W(C) + UB(P) = 25 + 40 \leq 70$, prune and return to the precedent level of recursion
Step 8	Depth 1	$\emptyset$	{2, 5, 3, 7, 4}	Search the largest weight clique in the remaining graph not containing vertices 1 and 6 ($P = \{2, 5, 3, 7, 4\}$), since $W(C) + UB(P) = 0 + 60 \leq W(C^*)$, prune and the whole procedure stops

Before its search, *MaxWClique* first calls the function *ColorSort* (line 4, Algorithm 1) to color the vertices in V and then resorts these vertices in non-decreasing order with respect to their color numbers, and inside color class by non-increasing weights. After these coloring and resorting steps, V becomes $V = \{6, 1, 2, 5, 3, 7, 4\}$ with the respective coloring $ColP = \{1, 1, 2, 2, 3, 3, 4\}$. Then at the first step of the algorithm, *MaxWClique* chooses vertex 6 as the branching vertex and adds this vertex to the current clique C, thus $C = \{6\}$ and $P = \{3, 4, 5, 7\}$. Before moving to the next level of recursion, vertices in P are colored and resorted by *ColorSort*, leading to $P = \{3, 5, 7, 4\}$ with the respective coloring $ColP = \{1, 1, 1, 2\}$. At the second step of the algorithm, vertex 3 is selected as the branching vertex to expand the current clique C, thus, $C = \{6, 3\}$ and $P = \{4\}$. Once again, before going to the next level recursion, *ColorSort* is called to color and resort P, thus $P = \{4\}$ with coloring $ColP = \{1\}$. At step three, the only vertex 4 in P is selected to join C and, C and P become {6, 3, 4} and $\emptyset$ respectively. Since $P = \emptyset$ and $W(C) > W(C^*)$, C^* and the lower bound $W(C^*)$ are updated as $C^* = \{3, 4, 6\}$ and $W(C^*) = 70$, respectively. As $P = \emptyset$, *MaxWClique* returns to the precedent level of recursion. For step four, once again $P = \emptyset$ when removing vertex 4, *MaxWClique* returns to the precedent level of recursion and examines the remaining vertices in P. For step five where $P = \{5, 7, 4\}$ with coloring $ColP = \{1, 1, 2\}$ when removing vertex 3, the upper bound $UB(P)$ is computed as $UB(P) = W(I_1) + W(I_2) = 40$. Since $W(C) + UB(P) = 30 + 40 \leq W(C^*)$, we can safely prune the search here and *MaxWClique* returns to the precedent level of recursion. At step six, *MaxWClique* returns to the first level of recursion and examines the left vertices in P excluding vertex 6 (i.e., $P = \{1, 2, 5, 3, 7, 4\}$ with the respective coloring $ColP = \{1, 2, 2, 3, 3, 4\}$). Since $W(C) + UB(P) = 0 + 85 > W(C^*)$, *MaxWClique* chooses vertex 1 as the branching vertex and goes to the next level of recursion. For step seven, at this level of recursion, $C = \{1\}$, vertices in P are colored and resorted as $P = \{2, 5, 3\}$ (with coloring $ColP = \{1, 1, 2\}$). Since $W(C) + UB(P) = 25 + 40 \leq W(C^*)$, we prune the search tree and once again return to the first level of recursion. For step eight, $C = \emptyset$ and $P = \{2, 5, 3, 7, 4\}$ (with $ColP = \{2, 2, 3, 3, 4\}$), since $W(C) + UB(P) = 0 + 60 \leq W(C^*)$, we prune the search tree and the whole procedure stops. Finally, *MaxWClique* returns the maximum weight clique $C^* = \{6, 3, 4\}$ with its weight $W(C^*) = 70$. To complement these explanations, Fig. 2 shows the search tree generated by *MaxWClique* when it is applied to the example of Fig. 1.

5. Experimental results

In this section, we evaluate our *MaxWClique* algorithm on a large number of WDP benchmark instances in the literature and compare our results with those obtained by the general-purpose integer programming CPLEX 12.4 solver. Indeed, previous studies have showed that the general integer programming approach based on CPLEX is highly effective to WDP in many cases. In [30], it was shown that CPLEX 8.0 is comparable with one of the best performing exact algorithms CABOB. Two early studies demonstrated that CPLEX 6.5 is faster than (or comparable to) the first-generation special-purpose exact search algorithms [10,28]. So when we compare our clique-based branch-and-bound algorithm against CPLEX 12.4, we are comparing our algorithm against one of the state-of-the-art methods. For our experiments, CPLEX 12.4 is run on the mathematical model defined in Eqs. (1)–(3).

Our *MaxWClique* algorithm was programmed in C (available from the authors by request), and compiled with GNU gcc on an Intel Xeon E5440 with 2.83 GHz CPU and 8GB RAM without compilation optimization flag. When solving the DIMACS machine benchmarks[1], the run time on our machine is 0.31, 1.93 and 7.35 s respectively for graphs r300.5, r400.5 and r500.5. For a fair comparison between *MaxWClique* and CPLEX 12.4, we ran both software on the same computing platform.

[1] ftp://dimacs.rutgers.edu/pub/dsj/clique/.

5.1. Benchmark instances

Three sets of benchmark instances were considered in this paper to evaluate the efficiency of our proposed *MaxWClique* approach. The first set of 500 instances is composed of pre-generated problem instances, while the second and third sets are random instances created by generators for combinatorial auctions according to several distributions. The characteristics of these instances are described in [17,20,28] and summarized as follows.

The first set of benchmarks was provided by Lau and Goh [17], and includes 500 instances with up to 1500 items and 1500 bids. These instances are divided into 5 different groups, each group having 100 instances labeled as REL-*m*-*n*, where *m* is the number of items and *n* is the number of bids. To generate these instances, several factors are incorporated such as a pricing factor, a bidder preference factor and a fairness factor in distributing items among bids. More details about how these instances are generated can be found in [17].

The second test set of instances were obtained with a generator for combinatorial auctions provided by Sandholm [28], which can be used to generate instances of different sizes and distributions. The following four auction distributions were used.

- *Random*(m, n): The n bids were generated using the following method. For each bid, pick the number of items randomly from {1,2,…,m}. Randomly choose that many items without replacement from {1,2,…,m}. Pick the price randomly from a uniform distribution on [0, 1].
- *Weighted Random*(m, n): As above, but pick the price randomly from a uniform distribution on 0 and the number of items in the bid.
- *Uniform*(m, n, λ): For each of the n bids, randomly choose λ items without replacement from {1,2,…,m}. Pick the price randomly from [0, 1].
- *Decay*(m, n, α): For each of the n bids, include a first item randomly selected from {1,2,…,m}. Then repeatedly add a new item randomly selected from {1,2,…,m} with a probability of α until an item is rejected or all m items are included in the bid. Pick the price randomly from a uniform distribution on 0 and the number of items in the bid. In our experiments, the parameter α was set equal to 0.75, since as indicated in [28], this setting leads to the hardest instances on average (at least for the algorithm in [28]).

The third set of problem instances was generated by the CATS generator (Combinatorial Auction Test Suite) introduced in [20][2]. Five different auction distributions are available in the CATS suite: paths, regions, matching, scheduling, and arbitrary. For each of these distributions, we used the default parameters provided by CATS.

5.2. Experimental results on the REL instances

In this section, we show the computational statistics obtained by our *MaxWClique* algorithm on the REL benchmark instances and compare our results with those attained by the CPLEX 12.4 solver which was run on the SPP model provided in [1]. As indicated in studies like [3,8], the REL benchmarks are difficult for CPLEX and some other exact WDP algorithms. We are unaware of any exact WDP algorithm reporting results on the REL benchmarks to the best of our knowledge. To obtain the results, the time limit for both *MaxWClique* and CPLEX 12.4 was set to 3600 s. If an approach fails to solve an instance to optimality within the given time limit, we report the best results (the lower bound) obtained by the approach and denote the computational time as 3600 s.

Tables 2–6 summarize the computational results obtained by our *MaxWClique* algorithm in comparison with those obtained by CPLEX 12.4 on the same set of 94 REL instances which were used in previous studies like [3,8,37] to assess the performance of heuristic approaches. Column 2 reports the density of the transformed graph (i.e., the number of edges of the graph divided by the number of edges of the complete graph of the same order). Columns 3 and 6 give respectively for *MaxWClique* and CPLEX 12.4 the optimal value if an optimal solution is found or the best lower bound achieved if no optimal solution is achieved within the time limit. Column 4 (denoted by Steps) indicates the number of branching steps required by *MaxWClique*. Each branching step corresponds to adding a vertex $v \in P$ to the current clique C. Note that the time reported for *MaxWClique* is for the computation of MWCP only. It does not include the pre-processing time to create the MWCP graph and the time to map the solution of MWCP back to WDP. Including these two steps slightly increases the computational time (less than 0.5 seconds).

From Tables 2–6, we observe that for all of these 94 selected REL instances, our *MaxWClique* algorithm is able to find the optimal solutions within the given time limit. Concerning the results obtained by CPLEX, we observe that only for the 30 REL-1000-500 instances, CPLEX is able to solve these instances to optimality within the given time limit. For 55 of the 94 selected instances, CPLEX fails to reach an optimal solution. For the other 9 instances, CPLEX is able to reach an optimal solution but fails to prove its optimality. From the table, it can also be seen that *MaxWClique* consistently outperforms CPLEX by achieving better results in much shorter times or being faster than CPLEX for every instance.

To further illustrate the effectiveness of the *MaxWClique* algorithm for the instances of this test set, we summarize in Table 7 the averaged results obtained by our *MaxWClique* algorithm in comparison with those obtained by CPLEX on all the 500 REL instances of the five groups of the first test set. In Table 7, column μ_W corresponds to the arithmetic average revenue obtained by the corresponding approach on the 100 instances of each group while column μ_{Time} reports the average time in second.

[2] http://www.cs.ubc.ca/~kevinlb/CATS.

Table 2
MaxWClique versus CPLEX 12.4 on some of the REL-500-1000 instances.

Instance	Density	*MaxWClique*			CPLEX	
		W	Steps	Time	*W*	Time
in101	0.31	**72724.61**	30711159	558.53	67101.94	3600
in102	0.29	**72518.22**	15407971	264.57	70292.58	3600
in103	0.30	**72129.50**	21705581	375.53	69703.05	3600
in104	0.30	**72709.64**	17867510	296.03	71579.58	3600
in105	0.29	**75646.12**	18855463	343.48	68431.12	3600
in106	0.29	**71258.61**	10649749	176.76	66621.12	3600
in107	0.30	**69713.40**	33379993	534.32	69182.50	3600
in108	0.31	**75813.20**	63070304	1089.81	74637.79	3600
in109	0.29	**69475.89**	12732376	219.96	65901.61	3600
in110	0.29	**68295.28**	19924045	336.73	67618.87	3600
in111	0.30	**75133.29**	26063679	458.04	72242.28	3600
in112	0.30	**71342.48**	19503263	329.40	70588.82	3600
in113	0.31	**73365.87**	39108287	718.52	70475.80	3600
in114	0.30	**69224.75**	24946718	668.85	66757.96	3600
in115	0.30	**70221.56**	16280138	267.73	66149.07	3600
in116	0.31	**70032.43**	21235344	381.35	69308.00	3600
in117	0.29	**69982.83**	16282639	289.80	69923.79	3600
in118	0.31	72160.98	24555810	672.10	72160.98	3600
in119	0.30	**67038.42**	30748116	541.61	64934.13	3600
in120	0.32	**75514.93**	36458776	1042.98	74658.12	3600
Average		**71715.10**	24974346	478.30	69413.45	3600

Table 3
MaxWClique versus CPLEX 12.4 on some REL-1000-1000 instances.

Instance	Density	*MaxWClique*			CPLEX	
		W	Steps	Time	*W*	Time
in201	0.15	**81557.74**	411331	3.07	79466.83	3600
in202	0.15	**90708.12**	573636	4.85	90537.28	3600
in203	0.16	86239.21	746917	6.06	86239.21	3600
in204	0.16	87075.42	876275	7.23	87075.42	3600
in205	0.15	**86515.95**	724793	5.75	84016.43	3600
in206	0.15	**91518.96**	449189	3.51	86888.23	3600
in207	0.16	**93129.24**	755874	6.13	89085.69	3600
in208	0.15	**94904.67**	419107	3.61	91782.04	3600
in209	0.15	**87268.96**	719742	5.08	83166.69	3600
in210	0.15	**89962.39**	493544	4.02	86940.49	3600
in211	0.15	**84913.68**	684138	4.96	84028.31	3600
in212	0.16	**90778.20**	850172	7.11	85390.73	3600
in213	0.16	**85369.18**	847181	6.61	83501.07	3600
in214	0.15	**85181.60**	700029	5.03	83554.16	3600
in215	0.17	**91531.69**	1560650	12.85	85965.20	3600
in216	0.16	**91580.93**	565825	4.79	85656.94	3600
in217	0.13	86962.92	215705	1.52	86962.92	3600
in218	0.16	**94965.19**	525335	4.46	88300.26	3600
in219	0.15	**93586.43**	524144	3.79	86006.20	3600
in220	0.17	**89792.90**	1181878	9.78	87883.45	3600
Average		**89177.16**	691273.25	5.51	86122.37	3600

The results reported in Table 7 further confirm that *MaxWClique* dominates the CPLEX 12.4 solver on the whole set of the REL instances. Indeed, on four of the five groups of instances, *MaxWClique* is able to achieve better results in much shorter times than CPLEX, while on the other remaining group (REL-1000-500) where both *MaxWClique* and CPLEX reach the same revenue, *MaxWClique* remains much faster.

5.3. Experimental results on the Sandholm benchmarks

In this section, we test our *MaxWClique* algorithm on the four Sandholm's distributions. To produce the test instances for each distribution, we fixed the number of bids (n) equal to 2000 and varied the number of items (m) from 100 to 500. For the uniform distribution, we fixed the number of items contained in each bid equal to 20. For each pair of fixed m and n, 100 problem instances were generated.

Table 8 summarizes the comparison results between *MaxWClique* and CPLEX. Each row in Table 8 corresponds to the average results of *MaxWClique* and CPLEX on the 100 instances of each pair of fixed m and n. From Table 8, we observe that the random

Table 4
MaxWClique versus CPLEX 12.4 on some of the REL-1000-500 instances.

Instance	Density	*MaxWClique*			CPLEX	
		W	Steps	Time	*W*	Time
in401	0.14	77417.48	9991	0.06	77417.48	24.26
in402	0.14	76273.33	11167	0.06	76273.33	30.35
in403	0.15	74843.95	9870	0.05	74843.95	35.43
in404	0.16	78761.69	16068	0.09	78761.69	33.76
in405	0.16	75915.90	24609	0.12	75915.90	67.49
in406	0.14	72863.32	12441	0.06	72863.32	55.51
in407	0.17	76365.71	20623	0.09	76365.71	77.82
in408	0.15	77018.83	14625	0.07	77018.83	124.24
in409	0.13	73188.62	9462	0.06	73188.62	51.24
in410	0.16	73791.65	20860	0.12	73791.65	50.81
in411	0.15	73935.40	15787	0.06	73935.40	27.10
in412	0.16	75292.63	13065	0.07	75292.63	69.12
in413	0.16	74434.99	21079	0.10	74434.99	62.49
in414	0.17	77146.37	24569	0.11	77146.37	74.75
in415	0.14	73519.12	11299	0.06	73519.12	68.57
in416	0.16	73487.01	20287	0.09	73487.01	56.36
in417	0.15	74981.35	13442	0.07	74981.35	49.13
in418	0.14	71404.84	10536	0.05	71404.84	28.75
in419	0.15	72505.21	13692	0.08	72505.21	40.36
in420	0.15	75510.68	12007	0.07	75510.68	54.17
in421	0.16	75694.94	17334	0.09	75694.94	27.50
in422	0.15	77443.90	14312	0.05	77443.90	28.30
in423	0.13	68134.35	7015	0.06	68134.35	35.17
in424	0.17	77352.75	20772	0.11	77352.75	49.28
in425	0.17	77333.91	20102	0.11	77333.91	49.21
in426	0.17	76430.18	21417	0.11	76430.18	211.26
in427	0.15	76387.56	16086	0.11	76387.56	57.56
in428	0.15	77384.94	13432	0.06	77384.94	52.00
in429	0.15	75540.96	16565	0.06	75540.96	61.19
in430	0.16	79038.75	17985	0.09	79038.75	66.95
Average		75313.34	15683.3	0.08	75313.34	57.33

Table 5
MaxWClique versus CPLEX 12.4 on some of the REL-1500-1500 instances.

Instance	Density	*MaxWClique*			CPLEX	
		W	Steps	Time	*W*	Time
in601	0.09	**108800.44**	841358	8.01	105286.85	3600
in602	0.08	**105611.47**	514680	4.86	99254.88	3600
in603	0.08	**105121.02**	390253	3.77	101270.04	3600
in604	0.09	**107733.80**	1100930	10.00	105185.67	3600
in605	0.09	**109840.98**	723970	7.11	103694.50	3600
in606	0.09	107113.06	665305	6.31	107113.06	3600
in607	0.09	**113180.28**	718312	7.35	103095.66	3600
in608	0.09	**105266.10**	769076	6.84	99490.66	3600
in609	0.09	**109472.33**	574016	5.73	100895.86	3600
in610	0.10	113716.96	1293161	13.14	113716.96	3600
in611	0.09	106666.32	474365	4.53	106666.32	3600
in612	0.09	109796.70	614466	6.30	109796.70	3600
in613	0.09	**107980.15**	759740	7.29	99328.57	3600
in614	0.10	**108364.57**	932585	9.09	100513.13	3600
in615	0.08	**110508.81**	388152	3.62	104433.21	3600
in616	0.09	**109740.48**	710625	6.69	108139.54	3600
in617	0.09	**113302.43**	691033	6.59	105899.16	3600
in618	0.10	**111385.08**	1462985	15.45	105154.80	3600
in619	0.09	**107571.59**	763031	7.27	98035.64	3600
in620	0.09	**110937.97**	773302	7.63	101712.44	3600
Average		**109105.52**	758067.25	7.38	103934.18	3600

Table 6
MaxWClique versus CPLEX 12.4 on some of the REL-1000-1500 instances.

Instance	Density	*MaxWClique*			CPLEX	
		W	Steps	Time	*W*	Time
in501	0.08	88656.95	879603	9.28	88656.95	3600
in502	0.08	**86236.91**	449725	4.56	83757.54	3600
in503	0.07	**87812.37**	590872	6.21	86318.17	3600
in504	0.10	**85600.00**	555385	5.55	84220.22	3600
Average		**87076.55**	618896.25	6.40	85738.22	3600

Table 7
Comparison of *MaxWClique* and CPLEX on the five groups of 500 REL instances.

Instance	ins	*MaxWClique*		CPLEX	
		μ_W	μ_{Time}	μ_W	μ_{Time}
REL-500-1000	100	**71470.93**	436.86	69178.52	3600
REL-1000-500	100	75540.68	0.08	75540.68	57.82
REL-1000-1000	100	**89158.98**	5.56	86107.85	3600
REL-1000-1500	100	**89552.18**	6.39	88072.36	3600
REL-1500-1500	100	**108627.17**	7.29	103469.53	3600
Average		**86869.98**	90.46	84473.78	2891.51

Table 8
Comparison of *MaxWClique* and CPLEX on the Sandholm benchmarks. "-" denotes that CPLEX ran out of memory within a time limit of 3600 s.

Instance	ins	Density	*MaxWClique*		CPLEX	
			μ_W	μ_{Time}	μ_W	μ_{Time}
Random2000_100	100	0.03	19.92	2.69	19.92	0.89
Random2000_200	100	0.02	17.01	0.95	17.01	2.36
Random2000_300	100	0.02	16.83	0.15	16.83	4.15
Random2000_400	100	0.01	15.07	0.10	15.07	5.26
Random2000_500	100	0.01	13.86	0.05	13.86	7.23
Wrandom2000_100	100	0.04	45.16	1.67	45.16	0.70
Wrandom2000_200	100	0.02	43.05	0.53	43.05	3.13
Wrandom2000_300	100	0.02	42.35	0.23	42.35	4.19
Wrandom2000_400	100	0.01	40.23	0.13	40.23	5.46
Wrandom2000_500	100	0.01	39.57	0.06	39.57	9.01
Uniform2000_100_20	100	0.01	2.71	0.05	2.71	110.82
Uniform2000_200_20	100	0.10	4.29	0.25	4.29	189.21
Uniform2000_300_20	100	0.24	6.19	5.73	6.19	1423.57
Uniform2000_400_20	100	0.35	8.28	112.36	7.85	–
Uniform2000_500_20	100	0.45	9.26	1206.35	8.93	–
Decay2000_100	100	0.78	68.78	3600.00	85.19	0.05
Decay2000_200	100	0.89	122.79	3600.00	166.26	0.11
Decay2000_300	100	0.93	184.07	3600.00	223.12	2.33
Decay2000_400	100	0.95	230.26	3600.00	277.90	2.45
Decay2000_500	100	0.96	269.07	3600.00	321.59	2.31

distribution and the weighted random distribution are easy for both algorithms. It is also interesting to notice that the algorithms achieve their performance very differently. The performance of *MaxWClique* increases with the decrease of the number of items due to the decrease of the density of the transformed graphs, while the reverse is true for the performance of CPLEX. We also notice that on the random and weighted random distributions, the speeds are comparable, but *MaxWClique* is slightly faster than CPLEX. The uniform distribution is much harder than the random distribution and the weighted random distribution for both algorithms. The difficulty of the instances increases dramatically with the number of items for both algorithms. *MaxWClique* performs significantly better than the CPLEX on the uniform distribution, for two sets of benchmark instances (Uniform200_400_20 and Uniform200_500_20), CPLEX fails to report a solution since it runs out of memory. The decay distribution is significantly harder for the *MaxWClique* algorithm, for all of the five sets of decay distribution instances, *MaxWClique* fails to find the optimal solution due to the high density of the transformed graphs. However, the decay distribution seems to be easy for CPLEX which dominates *MaxWClique* by achieving better results in much shorter times on this distribution.

Table 9
Comparison of *MaxWClique* and CPLEX on the CATS distributions.

Instance	ins	Density	*MaxWClique*		CPLEX	
			μ_W	μ_{Time}	μ_W	μ_{Time}
Arbitrary2000_20	100	0.08	2401.65	2.83	2401.65	0.11
Arbitrary2000_40	100	0.18	4348.33	3600.00	4348.33	0.20
Arbitrary2000_60	100	0.25	4839.06	3600.00	5010.08	0.39
Arbitrary2000_80	100	0.34	6431.46	3600.00	6815.30	0.53
Arbitrary2000_100	100	0.50	7722.34	3600.00	8295.74	9.98
Matching2000_20	100	0.73	83.42	37.08	83.42	0.01
Matching2000_40	100	0.86	111.29	3600.00	111.29	0.02
Matching2000_60	100	0.92	252.88	3600.00	254.56	0.01
Matching2000_80	100	0.95	303.11	3600.00	311.98	0.03
Matching2000_100	100	0.96	424.73	3600.00	464.67	0.03
Paths2000_20	100	0.82	14.57	3600.00	14.57	0.02
Paths2000_40	100	0.84	19.17	3600.00	20.91	0.03
Paths2000_60	100	0.81	21.45	3600.00	24.95	0.05
Paths2000_80	100	0.81	29.52	3600.00	33.43	0.05
Paths2000_100	100	0.81	31.27	3600.00	36.15	0.06
Regions2000_20	100	0.03	2702.50	3.16	2702.50	0.06
Regions2000_40	100	0.22	3427.92	3600.00	3427.92	0.11
Regions2000_60	100	0.36	4915.37	3600.00	5003.04	0.13
Regions2000_80	100	0.41	6759.59	3600.00	7126.73	0.11
Regions2000_100	100	0.55	7115.06	3600.00	7747.56	0.31
Scheduling2000_20	100	0.52	45.03	3365.69	45.03	0.02
Scheduling2000_40	100	0.74	79.63	3600.00	81.31	0.03
Scheduling2000_60	100	0.82	122.76	3600.00	124.62	0.05
Scheduling2000_80	100	0.87	166.68	3600.00	166.68	0.05
Scheduling2000_100	100	0.89	211.39	3600.00	216.11	0.05

5.4. Experimental results on the CATS distributions

We turn now our attention to the performance of *MaxWClique* on the five CATS distributions: paths, regions, matching, scheduling and arbitrary. For each of the distributions, we used the default parameters in the CATS instance generators, fixed the number of bids to 2000 (n) and varied the number of items (m) from 20 to 100 (We also found that the instances with more than 100 items are significantly difficult for our *MaxWClique* algorithm). For each pair of fixed m and n, 100 problem instances were generated. Table 9 summarizes the comparison results between *MaxWClique* and CPLEX.

From Table 9, we observe that on all of the five CATS distributions, CPLEX performs much better and is significantly faster than our *MaxWClique* algorithm. Indeed, our *MaxWClique* algorithm performs poorly on the CATS test suite since it is able to find the optimal solution only for some instances with 20 items. This may be explained by the fact that (see also the analysis in the next section), the instances from the CATS distributions usually contain only small numbers of items per bid, which leads to dense graphs which are much harder for *MaxWClique* to find the maximum weight clique. Inversely, as indicated in studies like [30], approaches based on the ILP model such as CPLEX seem more appropriate to handle these cases with short bids.

6. Analysis of the performance of *MaxWClique*

6.1. Performance of MaxWClique *on WDP*

Our *MaxWClique* approach seeks the maximum weight clique in the transformed graph to solve the WDP problem. In this section, we provide some insights into the performance of *MaxWClique* and try to identify the classes of problem instances which are the most suitable and most difficult for this clique-based approach. As we observed from experimental results presented in Tables 2–9, it seems that the density of the transformed graph impacts on the behavior of the *MaxWClique* algorithm and there is a clear correlation between the instance difficulty and the density of the transformed graph. Obviously, graphs with a low density is much easier for *MaxWClique*, since for sparse graphs, the number of the vertices in the candidate set P decreases more quickly as the current clique C expands (see Section 4.1). Thus, from the perspective of the search tree, the path from the root node to the leaf node is much shorter for sparse graphs, leading to a considerably smaller search tree for the *MaxWClique* algorithm. On the other hand, the situation is different for graphs with a high density where each vertex has more adjacent vertices. Indeed, since the vertices adjacent to a specific vertex are kept in the candidate set P for further recursive examination, the depth of the search tree will increase and more computational time will be required. However, the case of a complete graph is an exception for *MaxWClique*. Since in this case, the maximum weight clique can be immediately reached by adding all the vertices in the graph to the clique. Then at each level of the recursion, since the pruning condition $W(C) + UB(P) \leq W(C^*)$ ($W(C^*) = \sum_{i=1}^{n} w_i$) always holds, *MaxWClique* can prune very effectively the search tree, leading to fast completion of the search procedure.

Table 10
Comparison of four MWCP algorithms on the set of 15 MWCP instances. The best result for each instance is marked in bold.

N	Density	*MaxWClique*	*Cliquer*	*DK*	*ÖK*
1000	0.40	7.27	**2.12**	5.54	15.18
1000	0.50	88.40	**36.13**	108.39	415.46
900	0.50	48.31	**18.02**	67.29	188.35
700	0.60	156.72	**90.12**	429.75	1038.35
500	0.60	14.18	**13.75**	24.13	72.73
500	0.70	**233.12**	324.23	1082.56	7431.88
300	0.70	**3.22**	4.27	12.55	32.78
300	0.80	**78.12**	280.32	712.25	9918.46
200	0.80	**2.13**	6.02	12.38	38.35
200	0.90	**29.53**	1640.02	1080.49	>10800
150	0.90	**1.72**	36.23	30.56	968.00
150	0.95	**4.05**	1846.12	232.30	>10800
150	0.98	**0.01**	1942.13	297.67	>10800
100	0.95	**0.02**	1.45	0.86	57.81
100	0.98	**0.01**	0.65	0.03	115.75

Thus, it can be expected that our *MaxWClique* algorithm is especially effective for WDP instances with a large numbers of items per bid. Indeed, for an instance with many items per bid, two bids have a higher chance of being conflicting by sharing a common item, thus leading to a sparser transformed graph. This can explain why our *MaxWClique* algorithm shows excellent performances on most of the tested REL benchmark instances and on the instances from the random, weighted random and uniform distributions of the Sandholm test suite. We will provide additional computational evidence in Section 6.3 to support this expectation. Reversely, the clique-based approach may run into trouble on instances with small numbers of items per bid (like the decay distribution of the Sandholm test suite and the CATS distributions) since they lead to much denser graphs. On the other hand, approaches based on the ILP model like CPLEX tend to handle such instances well thanks to the multiple and dedicated technologies (pre-processing…) used. Thus, our clique-based approach can be considered as a complementary method with respect to other exact WDP methods.

6.2. *Performance of* MaxWClique *on MWCP*

Our *MaxWClique* algorithm is a clique-based approach, it is interesting to investigate whether *MaxWClique* remains competitive on the original maximum weight clique problem. To answer this question, we make a comparison with two well-known and fast exact algorithms in the literature for MWCP: the *Cliquer* algorithm proposed by Östergård [22] and the *DK* algorithm proposed by Kumlander [11].

For the *Cliquer* algorithm, we used its last version released in 2008[3] and ran it with its default parameters. For the *DK* algorithm, we downloaded its source code which was implemented in VB[4]. Given that *MaxWClique* and *Cliquer* were written in C which is much faster than VB, we faithfully translated *DK*'s VB code in C[5]. In addition, we also included in our comparison another version of *Cliquer* implemented (in VB) by Kumlander [12][6] that was used in [12] to compare with the *DK* algorithm. Again, we faithfully translated the VB code into a faster C code and denote this *Cliquer* implementation by *ÖK*. Note that in *ÖK*, the initial vertex ordering was given by a greedy vertex coloring, whereas in the original *Cliquer* algorithm of [22], the vertices were sorted by vertex weights and the sum of weights of adjacent vertices. As observed in [34], these two ordering strategies degrade the performance of *ÖK* relative to the original *Cliquer*.

For this experiment, we used gcc to compile (with no optimization option) the four compared algorithms (*MaxWClique*, *Cliquer*, *DK*, and *ÖK*). Our comparison was based on a set of 15 random graphs with 100 to 1000 vertices, where the weights of vertices were randomly assigned from 1 to 10. Table 10 summarizes the run times of *Cliquer*, *DK* and *MaxWClique* to solve these instances. Columns 1 and 2 respectively indicate the number of vertices and the densities of the graphs. Table 10 discloses that *MaxWClique* competes favorably with *Cliquer*. Moreover, these two methods perform quite differently on sparse and dense graphs and complement each other. *Cliquer* is faster than *MaxWClique* for the relatively easy sparse graphs (with density < 0.7). However, *MaxWClique* is much faster than *Cliquer* for graphs of density ≥ 0.7, and the speed-up also grows with the density of the graph. With respect to *DK*, we observe again that *MaxWClique* competes very favorably. Indeed, *MaxWClique* is faster than *DK* over all instances except the first instance where *MaxWClique* is slightly slower. When comparing *DK* and *Cliquer*, we note that *DK* is faster for graphs with density > 0.8 while the reverse is true for graphs with density ≤ 0.8. Finally, the results in Table 10 also reveal that *DK* is faster than *ÖK* for all tested instances, confirming the contribution of sorting the initial vertices by weights to the overall performance of the *Cliquer* algorithm, as already observed in [34].

[3] Available at: http://users.tkk.fi/pat/cliquer.html
[4] Available at: http://www.kumlander.eu/graph/Weighted/clsVColorBTw.txt
[5] The C code is available at: www.info.univ-angers.fr/pub/hao/MaxWClique.html
[6] http://www.kumlander.eu/graph/Weighted/clsPatricWeight.txt

Table 11
MaxWClique versus MN/TS on 25 REL and Sandholm instances.

Instance	Density	*MaxWClique*			MN/TS	
		W	T_s	T_{hit}	W	T_{hit}
in101	0.31	72724.61	558.53	65.29	72724.61	**5.46**
in102	0.29	72518.22	264.57	29.19	72518.22	**19.91**
in103	0.30	72129.50	375.53	183.23	72129.50	**18.52**
in104	0.30	72709.64	296.03	125.56	72709.64	**7.33**
in201	0.15	81557.74	3.07	**0.22**	81557.74	9.45
in202	0.15	90708.12	4.85	**0.58**	90708.12	2.47
in203	0.16	86239.21	6.06	**0.46**	86239.21	3.88
in204	0.16	87075.42	7.23	**0.39**	87075.42	2.67
in401	0.14	77417.48	0.06	**0.01**	76273.33	0.16
in402	0.14	76273.33	0.06	**0.02**	76273.33	0.38
in403	0.15	74843.95	0.05	**0.03**	74843.95	3.02
in404	0.16	78761.69	0.09	**0.03**	78761.69	0.87
in501	0.08	88656.95	9.28	**1.02**	88656.95	1.47
in502	0.08	86236.91	4.56	**0.80**	86236.91	1.76
in503	0.07	87812.37	6.21	**1.86**	87812.37	19.63
in504	0.10	85600.00	5.55	**1.63**	85600.00	4.62
in601	0.09	108800.44	8.01	**0.83**	108800.44	9.12
in602	0.08	105611.47	4.86	**0.89**	105611.47	1.72
in603	0.08	105121.02	3.77	**1.03**	105121.02	1.21
in604	0.09	107733.80	10.00	**3.15**	107733.80	16.62
Random2000_100	0.03	18.16	1.89	1.05	18.16	**0.17**
Wrandom2000_100	0.03	43.52	2.09	**1.08**	43.52	7.02
Uniform2000_100_10	0.33	6.85	80.14	19.66	6.85	**19.17**
Decay2000_100	0.78	68.13	3600.00	3313.15	**86.37**	**217.96**
Decay2000_200	0.89	125.88	3600.00	3215.32	**159.18**	**220.01**

6.3. Clique approach for WDP: exact algorithm vs heuristic algorithm

In [37], the authors explored the clique-based approach for solving WDP by applying a clique heuristic called MN/TS. Based on a large computational study on various WDP benchmark instances, they showed that MN/TS competes very favorably with several heuristic algorithms specially designed for WDP. In this section, we carry out an additional study to contrast *MaxWClique* of this paper and MN/TS of [37]. Since we are comparing an *exact* algorithm (*MaxWClique*) which guarantees the optimality of its solutions and a heuristic algorithm (MN/TS) which only provides lower bounds, some cautions must be taken. In fact, as a heuristic, MN/TS just tries to reach a solution as good as possible. Unlike MN/TS (and any other heuristics), the exact *MaxWClique* algorithm not only attains the optimal solution C^*, but also proves there does not exist any solution better than C^*. In many cases, even if the best (optimal) solution can be found at the early stage of the search process, the algorithm needs additional time to prove the optimality of the found solution. As a consequence, it is not meaningful to directly compare the computing times required by an exact method and a heuristic. Yet, it is interesting to contrast these two different solution approaches (exact and heuristic) via the clique-based approach for WDP.

For this purpose, we applied *MaxWClique* to solve exactly 25 REL and Sandholm benchmark instances used in [37] and reported our results in Table 11 along with the results of MN/TS extracted from [37]. Note that both algorithms were programmed in C and run on the same computing platform. In Table 11, the time of MN/TS (T_{hit}) is the time for MN/TS to hit for the first time its best results (lower bounds) and each row in Table 11 corresponds to a single instance. To make a fair comparison, for the exact *MaxWClique* algorithm, we reported in Table 11 the time for *MaxWClique* to hit the optimal results (T_{hit}) as well as the time for *MaxWClique* to complete its search (i.e., prove the optimality of the solution found) (T_s). In some sense, one can compare the two T_{hit} columns of MN/TS and *MaxWClique*. From Table 11, we observe that *MaxWClique* hits its best solutions quickly (T_{hit}), especially for the instances with low density (also see the analysis of Section 6.1), though for some instances with high density (such as the two 'decay' instances), *MaxWClique* performs much worse than MN/TS. Naturally, *MaxWClique* requires in general much more time (T_s) to prove the optimality of it solutions.

To further highlight the advantage of our *MaxWClique* algorithm over MN/TS for solving large sparse graphs (see also Section 6.1), we tested both algorithms on 24 groups (10 graphs per group) of randomly generated large sparse graphs with 20,000–50,000 vertices and a density ranging from 0.02 to 0.10. For each given N (vertices) and *Density*, we generated 10 random graphs, where the vertices were assigned a random weight from 1 to 1000. We used both *MaxWClique* and MN/TS to solve each of these 240 instances with a time limit of 300 s (the same time limit as used in [37] for MN/TS). We summarized in Table 12 the comparative results of *MaxWClique* and MN/TS (averaged over the 10 instances of each group).

The results of Table 12 show a clear dominance of *MaxWClique* over MN/TS on these graphs. For each of the 24 groups of instances, the *MaxWClique* algorithm attains a much larger average weight when compared to MN/TS. Particularly, for most of the tested instances (those marked with an asterisk in Table 12), our *MaxWClique* algorithm is able to complete its search (i.e., prove the optimality of the solution found) within the given timeout limit while MN/TS only finds much worse sub-optimal

Table 12
Comparison of *MaxWClique* and MN/TS on 24 families of 240 randomly generated large sparse graphs under a time limit of 300 s (the same timeout limit as in [37]).

N	ins	Density	*MaxWClique*			MN/TS	
			μ_W	μ_{T_s}	$\mu_{T_{hit}}$	μ_W	$\mu_{T_{hit}}$
20000	10	0.02	**4054.6***	18.79	6.25	3899.0	189.10
20000	10	0.04	**4933.3***	29.95	15.69	4780.0	236.02
20000	10	0.06	**5573.5***	64.26	28.78	5333.3	135.96
20000	10	0.08	**5917.2***	143.01	98.47	5763.6	226.18
20000	10	0.10	**6351.7***	286.36	115.32	6253.5	223.23
25000	10	0.02	**4294.9***	21.25	12.10	3922.4	170.56
25000	10	0.04	**4945.1***	52.74	21.15	4796.9	153.65
25000	10	0.06	**5623.0***	123.39	75.28	5436.8	139.18
25000	10	0.08	**6220.8***	271.85	118.69	5768.3	146.75
25000	10	0.10	**6521.9**	>300.00	287.87	6296.2	191.28
30000	10	0.02	**4424.4***	28.29	19.56	4165.0	102.62
30000	10	0.04	**5170.3***	83.95	39.28	4842.1	189.63
30000	10	0.06	**5719.9***	229.08	139.84	5561.2	142.29
30000	10	0.08	**6236.1**	>300.00	279.23	5995.6	251.62
35000	10	0.02	**4592.2***	40.66	16.68	4197.7	115.71
35000	10	0.04	**5184.5***	131.19	52.28	4880.9	165.23
35000	10	0.06	**5752.9***	285.27	172.56	5595.2	211.95
40000	10	0.02	**4645.7***	58.46	36.32	4209.9	233.31
40000	10	0.04	**5244.6***	190.58	81.21	4935.4	145.41
40000	10	0.06	**5946.8**	>300.00	268.02	5569.3	231.58
45000	10	0.02	**4667.3***	67.23	42.60	4223.8	145.92
45000	10	0.04	**5249.1***	223.69	136.14	4959.7	231.76
50000	10	0.02	**4685.5***	78.18	43.95	4290.9	145.42
50000	10	0.04	**5475.9***	269.69	168.20	5029.2	218.05

solutions. We also tested both algorithms under relaxed time conditions and observed that *MaxWClique* always dominates MN/TS on these large sparse graphs even if MN/TS finds improved solutions. The outcomes of this experiment are consistent with those of Section 6.1 and further confirm the effectiveness of the *MaxWClique* algorithm for solving large sparse graphs which remain difficult for MN/TS.

To summarize, with the clique-based approach, exact algorithms like *MaxWClique* and heuristic algorithms like MN/TS are complementary approaches and can be used to solve instances of different characteristics. In particular, *MaxWClique* is suitable for solving instances with low density while MN/TS is more effective for solving instances with high density. Considering these two solution approaches together, we conclude that these approaches enlarge the class of WDP instances that can be solved exactly or approximately with respect to the existing WDP approaches. Finally, from a more general perspective, these clique-based approaches could also be useful to handle large graphs in other settings like social network analysis [5].

6.4. Analysis of the sorting and branching strategy

As shown in [12], sorting and branching are very important since they can greatly affect the performance of a maximum weight clique algorithm. In our *MaxWClique* algorithm, we employ a coloring based vertex sorting technique, which first sorts vertices by color numbers in increasing order, and then inside color class by weights in decreasing order. Further more, before the coloring procedure is applied, all vertices presented to the greedy coloring procedure are sorted by weights in descending order. Thus, the color class with a smaller color number constructed by our greedy coloring procedure will include vertices of higher weights. Since the branching rule of *MaxWClique* selects these sorted vertices to join the clique in order, our *MaxWClique* algorithm favors the vertices with higher weights when branching. In Section 4.3, we put forward some expected advantages of our sorting and branching strategy. In this section, we provide experimental evidences to support these expectations. For this purpose, we compare our sorting and branching strategy (denoted by S_1) with two other strategies, S_2, sorting vertices by color numbers in decreasing order (such as $C_k, C_{k-1}, \ldots, 1$), and inside color class by weights in decreasing order, and S_3, sorting vertices by color numbers in decreasing order, and inside color class by weights in increasing order. Detailed experiments with these three sorting strategies were conducted on 6 selected WDP instances. For a fair comparison, we used the same greedy coloring procedure (Algorithm 2) and the same upper bounding strategy based on graph coloring.

The computational results are provided in Table 13 where we show the time required by the B&B algorithm with each different strategy to solve a given instance. As we can observe, the B&B algorithm with our sorting and branching strategy performs much better than the algorithms with the two other strategies, showing the merit of our adopted sorting and branching strategy. Finally, we mention that several similar sorting and branching strategies were developed and analyzed in [30], showing the interest of preferring to choose bids (vertices) with large profits (high weights) as a good branching technique.

Table 13
Comparison of three different sorting and branching strategies.

Instance	Density	Three sorting and branching strategies		
		S_1	S_2	S_3
in101	0.31	558.53	1172.91	1731.92
in201	0.15	3.07	4.51	5.31
in401	0.14	0.06	0.07	0.08
in501	0.08	9.28	15.36	19.58
in601	0.09	8.01	13.87	17.02
Random2000_100	0.03	2.69	3.02	3.58
Uniform200_400_20	0.35	112.36	280.90	393.26

7. Future research direction

As future work, one would like to investigate how other clique-based exact algorithms perform on the WDP problem. Since different clique-based algorithms are efficient for different classes of WDP instances. Such an investigation may enlarge the classes of WDP that can be effectively solved.

In addition, it would be interesting to explore the possibilities of adapting the proposed algorithm to other WDP variants with other constraints and business rules. In particular, the following issues could be investigated. First, we may modify transformation rules from WDP to MWCP. For instance, in some settings, a participant may wish to submit two or more bids but require that at most one bid will be allocated [19]. To handle this additional constraint, the transformation rule from WDP to MWCP can be modified as follows: any two vertices in the transformed MWCP instance are connected by an edge if and only if the corresponding bids share no common item and are not submitted by the same participant, implying that the two corresponding bids can be accepted together as winning bids. Second, we may modify the objective function of the algorithm. For instance, in some situations, the allocation rule seeks to maximize the total socially efficient outcomes. In this case, we can adjust our objective function value by further including the costs of all participants. Third, we may use our algorithm as an independent component for more complex auction situations. For instance, the iterative combinatorial auctions [24] consists of multi-round auctions and can be decomposed into several single-round auctions. For each single round auction, our algorithm can be directly applied to determine an optimal allocation.

Finally, contrary to the maximum clique problem which is one of the most studied combinatorial problems for a long time, its vertex weight version (i.e., the MWCP problem) is much less studied and only few exact algorithms exist. Moreover, there are currently no well-defined benchmark instances for performance assessment of a MWCP algorithm. Usually, the MWCP instances reported in the published papers are not available. With this work, we have generated a large number of MWCP instances with quite different structures by transforming various WDP instances. Clearly, these instances can form the basis of a standard benchmark for the MWCP problem. As it is shown in Sections 5 and 6.2, the proposed *MaxWClique* algorithm performs particularly well on some classes of instances, providing some good indications about our algorithm for the maximum weight clique problem.

8. Conclusion

Combinatorial auctions find more and more applications in divers domains, but determining the winners in combinatorial auctions is a hard combinatorial problem. In this paper, we have investigated an approach which transforms the optimal winner determination problem into the maximum weight clique problem. To solve the later clique problem, we introduced *MaxWClique*, a branch-and-bound algorithm which integrates effective bounding and branching strategies using a dedicated vertex coloring procedure.

We have evaluated extensively the performance of the proposed algorithm via a large experimental assessment with three well-known test suites (REL, Sandholm, CATS) from the literature. We have shown that in many cases, this clique-based algorithm can achieve very competitive results compared to the powerful CPLEX 12.4 solver, which is known to be one of the current best performing exact solvers for WDP. In particular, this clique-based approach is able to successfully solve the whole set of the REL instances, which are difficult for both exact and heuristic approaches in the literature. In addition, the proposed algorithm runs in a linear space, while CPLEX has an exponential space complexity, and runs out of virtual memory in some cases. The experiments have also disclosed that the clique-based approach performs much worse than CPLEX for the CATS distributions. Often this corresponds to problem instances with a short list of items per bid (leading to dense graphs) where other approaches like CABOB [30] and CPLEX perform very well. To sum, since the proposed *MaxWClique* algorithm and existing approaches are suitable for different classes of problem instances, they all together cover a larger spectrum of cases that can be solved effectively. In this sense, *MaxWClique* is not really a competitor, instead, it constitutes an interesting alternative and complementary approach to the important winner determination problem.

Finally, *MaxWClique* enriches the family of available algorithms for maximum clique problems and can be advantageously employed to enlarge the class of MCP and MWCP instances that can be solved exactly. Moreover, the various types of WDP instances used in this paper can constitute the basis for a future standard benchmark for the MWCP problem.

Uncited Reference

[16].

Acknowledgments

We are grateful to the reviewers and Prof. G. Kochenberger for their helpful comments and suggestions which helped us to improve the paper. This work is partially supported by the RaDaPop (2009-2013, 24-Radapop) and LigeRO projects (2009-2013, 07-LigeRO) from the Region of Pays de la Loire (France), the PGMO (2014-0024H) project from the Jacques Hadamard Mathematical Foundation (Paris), and the National Natural Science Foundation Program of China (Grants 71401059, 71131004, 71531009).

References

[1] A. Andersson, M. Tenhunen, F. Ygge, Integer programming for combinatorial auction winner determination, in: Proceedings of the 4th International Conference on Multi-agent Systems, IEEE Computer Society Press, New York, 2000, pp. 39–46.
[2] I.M. Bomze, M. Budinich, P.M. Pardalos, M. Pelillo, The maximum clique problem, in: Handbook of Combinatorial Optimization, Springer, 1999, pp. 1–74.
[3] D. Boughaci, B. Benhamou, H. Drias, A memetic algorithm for the optimal winner determination problem, Soft Comput. 13 (8–9) (2009) 905–917.
[4] R. Carraghan, P.M. Pardalos, An exact algorithm for the maximum clique problem, Op. Res. Lett. 9 (6) (1990) 375–382.
[5] A. Clauset, M.E.J. Newman, C. Moore, Finding community structure in very large networks, Phys. Rev. E 70 (2004) 066111.
[6] P. Cramton, Y. Shoham, R. Steinberg, Combinatorial Auctions, MIT Press, 2006.
[7] L.F. Escudero, M. Landete, A. Marín, A branch-and-cut algorithm for the winner determination problem, Decis. Support Syst. 46 (3) (2009) 649–659.
[8] Y. Guo, A. Lim, B. Rodrigues, Y. Zhu, Heuristics for a bidding problem, Comput. Op. Res. 33 (8) (2006) 2179–2188.
[9] O. Günlük, L. Lászlo, S. de Vries, A branch-and-price algorithm and new test problems for spectrum auctions, Manag. Sci. 51 (3) (2005) 391–406.
[10] Y. Fujishima, K. Leyton-Brown, Y. Shoham, Taming the computational complexity of combinatorial auctions: optimal and approximate approaches, in: Proceedings of the 6th International Joint Conference on Artificial Intelligence, 1999, pp. 548–553.
[11] D. Kumlander, A new exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring and a backtrack search, in: Proceedings of The Forth International Conference on Engineering Computational Technology, Civil-Comp Press, 2004, pp. 202–208.
[12] D. Kumlander, On importance of a special sorting in the maximum weight clique algorithm based on colour classes, in: Proceedings of the 2nd International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences, 2008, pp. 165–174.
[13] J. Konc, D. Janežič, An improved branch and bound algorithm for the maximum clique problem, MATCH - Commun. Math. Comput. Chem. 58 (2007) 569–590.
[14] H.H. Hoos, T. Strüzle, On the empirical scaling of run-time for finding optimal solutions to the traveling salesman problem, Eur. J. Op. Res. 238 (1) (2014) 87–94.
[15] H.H. Hoos, C. Boutilier, Solving combinatorial auctions using stochastic local search, in: Proceedings of the 17th National Conference on Artificial Intelligence, 2000, pp. 22–29.
[16] A. Holland, B. O'sullivan, Towards Fast Vickrey pricing using constraint programming, Artif. Intell. Rev. 21 (3–4) (2004) 335–352.
[17] H. Lau, Y.G. Goh, An intelligent brokering system to support multi-agent web-based 4th-party logistics, in: Proceedings of the 14the International Conference on Tools with Artificial Intelligence, 2002, pp. 154–161.
[18] D. Lehmann, M. Rudolf, T. Sandholm, et al., The winner determination problem, in: Cramton, et al. (Eds.), Combinatorial Auctions, MIT Press, Cambridge, 2006.
[19] K. Leyton-Brown, Y. Shoham, M. Tennenholz, An algorithm for multi-unit combinatorial auctions, in: Proceedings of the 7th International Conference on Artificial intelligence, 2000a, pp. 56–61.
[20] K. Leyton-Brown, M. Pearson, Y. Shoham, Towards a universal test suite for combinatorial auction algorithms, in: Proceedings of the ACM Conference on Electronic Commerce, ACM Press, Minneapolis, 2000, pp. 66–76.
[21] N. Nisan, Bidding and allocation in combinatorial auctions, in: Proceedings of the ACM Conference on Electronic Commerce, ACM SIGecom, ACM Press, Minneapolis, 2000, pp. 1–12.
[22] P.R.J. Östergård, A new algorithm for the maximum-weight clique problem, Nordic J. Comput. 8 (4) (2001) 424–436.
[23] M.W. Padberg, On the facial structure of set packing polyhedra, Math. Program. 5 (1) (1973) 199–215.
[24] D.C. Parkes, L.H. Ungar, Iterative combinatorial auctions: theory and practice, in: Proceedings of the 17th National Conference on Artificial Intelligence, 2000, pp. 74–81.
[25] A.K. Ray, M. Jenamani, P.K.J. Mohapatra, Supplier behavior modeling and winner determination using parallel MDP, Expert Syst. Appl. 38 (5) (2011) 4689–4697.
[26] M.H. Rothkopf, A. Pekeč, R.M. Harstad, Computationally manageable combinatorial auctions, Manag. Sci. 44 (8) (1998) 1131–1147.
[27] P. Samimi, Y. Teimouri, M. Mukhtar, A combinatorial double auction resource allocation model in cloud computing, Inf. Sci., In Press, 2014.
[28] T. Sandholm, Algorithm for optimal winner determination in combinatorial auctions, Artif. Intell. 135 (1–2) (2002) 1–54.
[29] T. Sandholm, S. Suri, BOB: Improved winner determination in combinatorial auctions and generalizations, Artif. Intell. 145 (1–2) (2003) 33–58.
[30] T. Sandholm, S. Suri, A. Gilpin, D. Levine, CABOB: a fast optimal algorithm for winner determination in combinatorial auctions, Manag. Sci. 51 (3) (2005) 374–390.
[31] S. Satunin, E. Babkin, A multi-agent approach to intelligent transportation systems modeling with combinatorial auctions, Expert Syst. Appl. 41 (15) (2014) 6622–6633.
[32] P.S. Segundo, D. Rodríguez-Losada, A. Jiménez, An exact bit-parallel algorithm for the maximum clique problem, Comput. Op. Res. 38 (2) (2011) 571–581.
[33] I. Sghir, J.K. Hao, I.B. Jaafar, K. Ghédira, et al., A Recombination-based tabu search algorithm for the winner determination problem, in: P. Legrand, et al. (Eds.), AE 2013, Lecture Notes in Computer Science, 8752, 2014, pp. 157–169.
[34] S. Shimizu, K. Yamaguchi, T. Saitoh, S. Masuda, Some improvements on Kumlander's maximum weight clique extraction algorithm, Acad. Sci. Eng. Technol. 6 (2012) 12–20.
[35] E. Tomita, T. Seki, An efficient branch-and-bound algorithm for finding a maximum clique, in: Proceedings of the 4th international conference on Discrete Mathematics and Theoretical Computer Science, Lecture Notes in Computer Science, 2731, 2003, pp. 278–289.
[36] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, M. Wakatsuki, A simple and faster branch-andbound algorithm for finding a maximum clique, Lect. Notes in Comput. Sci. 5942 (2010) 191–203.
[37] Q. Wu, J.K. Hao, Solving the winner determination problem via a weighted maximum clique heuristic, Expert Syst. Appl. 42 (1) (2015) 355–365.
[38] Q. Wu, J.K. Hao, A review on algorithms for maximum clique problems, Eur. J. Op. Res. 242 (3) (2015) 693–709.
[39] K. Yamaguchi, S. Masuda, A new exact algorithm for the maximum weight clique problem, in: Proceedings of the 23rd International Technical Conference on Circuits/Systems, Computers and Communications, 2008, pp. 317–320.
[40] S. de Vries, R.V. Vohra, Combinatorial auctions: a survey, INFORMS J. Comput. 15 (3) (2003) 284–309.