Journal of Artificial Intelligence Research 72 (2021) 39-67 Submitted 08/2020; published 09/2021

A Semi-Exact Algorithm for Quickly Computing a Maximum Weight Clique in Large Sparse Graphs

Shaowei Cai SHAOWEICAI.CS@GMAIL.COM
Jinkun Lin JKUNLIN@GMAIL.COM
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing, China

Yiyuan Wang YIYUANWANGJLU@126.COM
School of Computer Science and Information Technology
Northeast Normal University, China

Darren Strash DSTRASH@HAMILTON.EDU
Department of Computer Science
Hamilton College, Clinton, NY, USA

Abstract

This paper explores techniques to quickly solve the maximum weight clique problem (MWCP) in very large scale sparse graphs. Due to their size, and the hardness of MWCP, it is infeasible to solve many of these graphs with exact algorithms. Although recent heuristic algorithms make progress in solving MWCP in large graphs, they still need considerable time to get a high-quality solution. In this work, we focus on solving MWCP for large sparse graphs within a short time limit. We propose a new method for MWCP which interleaves clique finding with data reduction rules. We propose novel ideas to make this process efficient, and develop an algorithm called FastWClq. Experiments on a broad range of large sparse graphs show that FastWClq finds better solutions than state-of-the-art algorithms while the running time of FastWClq is much shorter than the competitors for most instances. Further, FastWClq proves the optimality of its solutions for roughly half of the graphs, all with at least 10^5 vertices, with an average time of 21 seconds.

1. Introduction

The proliferation of large data sets brings with it a series of special computational challenges. Many data sets can be modeled as graphs, and the research of large real-world graphs has grown enormously in last decade. A *clique* of a graph is a subset of the vertices that are all pairwise adjacent. Cliques are an important graph-theoretic concept, and are often used to represent dense clusters. The *maximum clique problem* (MCP) is a long-standing problem in graph theory, for which the task is to find a clique with the maximum number of vertices in the given graph. An important generalization of MCP is the maximum weight clique problem (MWCP), in which each vertex is associated with a positive integer weight, and the goal is to find a clique with the largest weight. MWCP has valuable applications in many fields (Ballard & Brown, 1982; Balasundaram & Butenko, 2006; Gomez Ravetti & Moscato, 2008).

The decision version of MCP (and thus MWCP) is one of Karp's prominent 21 NP-complete problems (Karp, 1972), and is complete for the class W[1], the parameterized analog of NP (Fellows & Downey, 1998). Moreover, MCP (and thus MWCP) is not approximable within $n^{1-\varepsilon}$ for any $\varepsilon > 0$ unless P = NP (Zuckerman, 2007). Nevertheless, these negative theoretical results have been established for a "worst case" that is not often observed in practice. We therefore still have hope of solving MWCP problems which arise in specific problem domains.

1.1 Related Work

Given their theoretical importance and practical relevance, considerable effort has been devoted to the development of various methods for MCP and MWCP, mainly including exact algorithms and heuristic algorithms. Exact algorithms are dedicated to find an exact solution and prove its optimality, while heuristic algorithms have been devised with the purpose of providing (potentially) sub-optimal solutions within an acceptable time.

Almost all existing exact algorithms for MCP are branch-and-bound (BnB) algorithms, and they differ from each other mainly by their techniques to determine the upper bounds and their branching strategies. A large family of BnB algorithms use coloring to compute upper bounds (Tomita & Seki, 2003; Tomita & Kameda, 2007; Konc & Janezic, 2007; Tomita et al., 2010; San Segundo et al., 2013). Another paradigm encodes MCP into a MaxSAT instance and then applies MaxSAT reasoning to improve the upper bound (Li & Quan, 2010; Li, Fang, & Xu, 2013; Li, Jiang, & Manyà, 2017).

There are also numerous works on heuristic algorithms for MCP, most of which are local search algorithms (Singh & Gupta, 2006b; Pullan & Hoos, 2006; Pullan, 2006; Guturu & Dantu, 2008; Benlic & Hao, 2013). A milestone local search algorithm for MCP is the Dynamic Local Search (DLS) algorithm due to Pullan and Hoos (2006), which was later improved to the Phased Local Search (PLS) algorithm by Pullan (2006). Pullan (2009) later adapted PLS to weighted problems.

Recently, there have been some dedicated algorithms for solving MCP in large graphs. These MCP algorithms (Rossi et al., 2014; Verma et al., 2015; San Segundo et al., 2016) heavily depend on the concept of the k-core (Seidman, 1983), which is a subgraph where all vertices have degree at least k, which can be computed in $O(m)$ (m is the number of edges) using bin sorting (Batagelj & Zaveršnik, 2003). We are not aware of any work using the k-core concept to develop MWCP algorithms, except implicitly by using an initial ordering of the vertices (Jiang, Li, & Manyà, 2017). Moreover, an analogous concept in vertex-weighted graphs requires prohibitive space ($O(\overline{w} \cdot m)$, where $\overline{w}$ is the average weight of vertices) for bin sorting, and does not allow fast computation.

MWCP is more complicated than MCP and some powerful techniques for MCP are not applicable or ineffective for solving MWCP due to the vertex weights. This partly explains the fact that there are relatively fewer algorithms for MWCP. Some exact algorithms for MWCP come from and generalize previous BnB methods designed for MCP (Östergård, 1999; Kumlander, 2004). The MaxSAT-based method was also generalized to MWCP by Fang et al. (2014), resulting in an exact MWCP algorithm named MaxWClq. Jiang et al. (2017) proposed an exact BnB algorithm for MWCP called WLMC, which is especially designed for large scale graphs. WLMC incorporates two important techniques. The first is a preprocessing to derive an initial vertex ordering and to reduce the size of the graph, and the other is the incremental vertex-weight splitting to reduce the number of branches in the search space. In parallel with WLMC, Li et al. (2018) proposed a new upper bound for MWCP which is based on the notion of a weight cover. The idea of a weight cover

is to compute a set of independent sets of the graph and define a weight function for each independent set so that the weight of each vertex of the graph is covered by such weight functions. This upper bound is used to develop a BnB-based exact algorithm named WC-MWC (Li et al., 2018). These two exact algorithms WLMC and WC-MWC achieve good results on large sparse graphs, and their performance is similar according to their experimental evaluation (Li et al., 2018). More recently, Jiang et al. (2018) proposed a BnB algorithm that combines a novel two-stage MaxSAT reasoning approach with effective BnB techniques for large graphs, and the resulting algorithm is called TSM-MWC, which significantly outperforms WLMC on a broad range of large real-world benchmarks. Therefore, TSM-MWC is the state of the art in this direction.

For solving MWCP, many researchers are devoted to designing effective heuristic algorithms, aiming to find a good quality solution in short time. Massaro, Pelillo, and Bomze (2002) proposed a complementary pivoting algorithm based on the corresponding linear complementarity problem. Busygin (2006) presented a heuristic method using a nonlinear programming formulation for MWCP. A hybrid evolutionary approach was offered by Singh and Gupta (2006a). An efficient local search algorithm for MCP called Phased Local Search (PLS) was extended to MWCP (Pullan, 2008), which interleaves different modes of local search. A local search algorithm called MN/NT integrates a combined neighborhood and a dedicated tabu mechanism, and shows better performance than previous heuristic algorithms on a broad range of benchmarks (Wu, Hao, & Glover, 2012). Afterwards, Wang, Cai, and Yin (2016) developed an improved local search algorithm called LSCC based on the configuration checking strategy and further improves MN/NT on a wide range of benchmarks. Based on LSCC, efficient local search MWCP algorithms for large graphs were developed. The LSCC+BMS algorithm improved LSCC by including a probabilistic heuristic called Best from Multiple Selection (BMS) (Cai, 2015), leading to much better performance on large graphs from the Network Data Repository. Fan et al. (2017) introduced the RRWL algorithm, another improved algorithm from LSCC, which incorporates restart, random walk and hash techniques, and shows better performance than LSCC+BMS. Besides the line of configuration-checking-based local search, Zhou, Hao, and Goëffon (2017) presented the generalized push operator for MWCP and then used this operator to develop two restarting tabu search algorithms ReTS1 and ReTS2. Nevertheless, ReTS1 and ReTS2 were mainly evaluated on medium- and small-sized graphs. Very recently, the authors of LSCC+BMS proposed the walk perturbation technique and several heuristics to diversify the search, resulting in an improved version named SCCWalk4L (Wang et al., 2020) for solving MWCP on large graphs. SCCWalk4L showed obviously better performance than previous heuristic algorithms on a broad range of large graphs (Wang et al., 2020), and thus represents the latest state of the art in heuristic MWCP algorithms for large graphs.

Further algorithms exist for solving MWCP, however, they are not feasible for solving large sparse graphs. San Segundo, Furini, and Artieda (2019) recently introduced an exact BnB algorithm for MWCP that uses a bitboard representation of the adjacency matrix, together with more advanced lower and upper bounds. However, as they use an adjacency matrix representation, their algorithm is not feasible for graphs with more than thousands of vertices. Of note here is that maximum weight independent set and minimum weight vertex cover solvers, many of which use data reductions, can also be used to solve MWCP (Lamm et al., 2019; Li et al., 2020; Cai et al., 2019). However, these algorithms would need to be run on the *complement graph*. For the graphs we consider here, the complement is dense and again infeasible to store in memory.

1.2 Contributions and Paper Organization

In many applications, either the time limit for computation is very short or computational resources are limited; however, the input graphs may be very large. This motivates us to develop algorithms for solving MWCP for large graphs within a short time limit (e.g., 100 seconds). In this work, we focus on solving MWCP for large sparse graphs. For such graphs, we show that we can make use of the sparsity to develop efficient algorithms that can solve MWCP quickly.

As the main contribution, we propose a method that interleaves clique finding and graph reduction, which is very effective for solving large sparse graphs. In a graph reduction procedure, we reduce the size of the graph by removing vertices that are proved to be in no clique of maximum weight. For large sparse graphs, their size can be reduced significantly by using a clique of certain quality in hand as a lower bound together with effective upper bound functions. On the other hand, the remaining smaller graph presents smaller search space and the algorithm may find better cliques more easily, which can then be used to further reduce the graph. As far as we know, this is the first algorithm that interleaves clique finding and graph reduction. Some existing algorithms for MCP (Rossi et al., 2014; Verma et al., 2015; San Segundo et al., 2016) and MWCP (Jiang et al., 2017) reduce the graph in a preprocessing procedure, which is done just one time. Moreover, the proposed framework interleaving clique finding and graph reduction allows us to develop semi-exact algorithms — the graph shrinks as the algorithm proceeds, and if the graph becomes empty, the found clique is proved to be optimal.

Besides the semi-exact algorithmic framework, we propose some ideas to make the method more efficient. The first two ideas are used in the clique finding phase:

- We propose a construct-and-cut algorithm for clique finding, in which an evaluation function is proposed for estimating the benefit of adding a vertex. The construct-and-cut method is proposed for the first time, which uses pruning techniques but differs from the BnB method.
- We also propose a dynamic version of the BMS heuristic, which is used in choosing a solution vertex.

The other two ideas are used in the graph reduction phase, including two novel upper bounds. The reduction rules based on these upper bounds significantly improve the ability of proving the optimality.

- We propose a branching-based upper bound, which makes use of the maximum weight neighboring vertex.
- We also propose a weighted-coloring-based upper bound, which relies on an algorithm to color the vertex-weighted graph.

Based on these ideas, we develop an algorithm called FastWClq[1]. To evaluate our algorithm, we consider two benchmarks of large graphs, the Network Data Repository benchmark (Rossi & Ahmed, 2015) and the KONECT benchmark (Kunegis, 2013). Also, to study the performance on these graphs with different weight distributions, we not only test the instances with weights generated according to the 'mod200' method used in the literature (Pullan, 2009), but also test them

1. Note that although this name has been used to denote an early version of the algorithm in the conference paper, here FastWClq refers to the final version as presented in this work.

with weights generated according to a normal distribution. Finally, we obtain four benchmarks for our experiments.

We compare our FastWClq algorithm with state-of-the-art algorithms for solving very large MWCP instances, including both exact and heuristic algorithms. Experiments show that FastWClq significantly outperforms previous algorithms in terms of the solution quality and run time. Further, as a semi-exact algorithm, FastWClq proves the optimality of its solutions for more than half of the challenging large graphs with more than 100 000 vertices, achieving similar (yet worse) performance with the state-of-the-art exact MWCP algorithm TSM-MWC in terms of the proving ability.

Note that an early version of FastWClq has been published in a conference paper (Cai & Lin, 2016). We describe the new contributions in this article below. We optimize the framework of FastWClq and propose two new ideas to improve FastWClq, including a clique-improving method during the clique finding phase and a weighted-coloring-based upper bound during the reduction phase. In addition, we add more experiments to evaluate performance of FastWClq, and compare with the latest state-of-the-art algorithms.

In the next section, we introduce some necessary background knowledge. Then, we describe the framework of the semi-exact method in Section 3. The details of the clique finding phase and the graph reduction phase are presented in Section 4 and 5 respectively. Experimental evaluations of our algorithm, FastWClq, are presented in Section 6. Finally, we give some concluding remarks and outline the future work in Section 7.

2. Preliminaries

Let $G = (V,E)$ be an undirected graph where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices and $E \subset \{\{u,v\} \mid u,v \in V\}$ is the set of edges in G. We denote their cardinalities by $n = |V|$ and $m = |E|$. We use $V(G)$ and $E(G)$ to denote the vertex set and the edge set of graph G. A vertex-weighted undirected graph is an undirected graph $G = (V,E)$ combined with a mapping $w : V \mapsto \mathbb{N}$ (where $\mathbb{N}$ is the set of natural numbers) so that each vertex $v \in V$ is associated with a natural number $w(v)$ as its weight. We use a triple to denote a vertex-weighted graph, i.e., $G = (V,E,w)$. For a subset $S \subseteq V$, we let $G[S]$ denote the subgraph induced by S, which is formed from S and all the edges $\{\{u,v\} \in E \mid u,v \in S\}$ connecting pairs of vertices in S. The weight of S is $w(S) = \sum_{v \in S} w(v)$. The (open) *neighborhood* of a vertex v is $N(v) = \{u \in V \mid \{u,v\} \in E\}$, and we denote the *closed neighborhood* by $N[v] = N(v) \cup \{v\}$. The *degree* of v is $d(v) = |N(v)|$.

Given a graph G, a *clique* $C \subseteq V$ is a set of pairwise adjacent vertices, while an *independent set* I is a set of pairwise nonadjacent vertices. A clique or independent set is *maximal* if it is not included in a larger clique or independent set. The *maximum clique problem* (MCP) is to find a clique of maximum cardinality in a graph, and the *maximum weight clique problem* (MWCP) is to find a clique of the maximum weight in a vertex-weighted graph. We say a vertex is "bad" if it is proved to be in no optimal solution.

For a graph, a *proper vertex coloring* is an assignment of colors to all vertices of the graph such that no two adjacent vertices share the same color. Equivalently, a proper vertex coloring of a graph G is a partition $\{A_1, A_2, ..., A_k\}$ of the vertex set $V(G)$ into independent sets. Under a proper vertex coloring, the set of vertices for a color forms an independent set. Therefore, any clique can contain at most one vertex from one color. This can be used to compute an upper bound for MCP and MWCP.

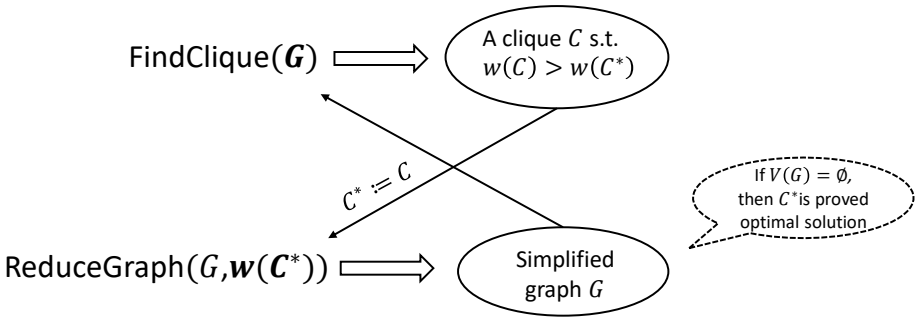

Figure 1. A semi-exact method for MWCP

The BMS (Best from Multiple Selection) heuristic by Cai (2015) is a sampling technique, which is used to choose a good-quality element from a large set. It randomly picks k elements and returns the best one with respect to some criterion. We use a dynamic BMS heuristic in our algorithm.

3. A Semi-Exact Method for MWCP: The Framework

In this section, we introduce the framework of our semi-exact method for solving MWCP, which interleaves clique finding and graph reduction. The algorithmic framework is illustrated in Figure 1.

This semi-exact method is composed of two sub-algorithms, a clique finding algorithm and a graph reduction algorithm. The clique finding algorithm aims to find a larger-weight clique than the best found clique C^*; once such a clique is found, C^* is updated to this new clique. As $w(C^*)$ is a lower bound on the optimal clique weight, when $w(C^*)$ is updated to a larger value, a tighter lower bound is obtained, and then the graph reduction algorithm is called to further simplify the graph. The graph reduction algorithm aims to simplify the graph by detecting "bad" vertices (i.e. those vertices that cannot be in any clique of maximum weight) and removing as many of them as possible. Additionally, if the graph becomes empty after reduction, then the best found solution C^* is proved to be optimal. Hence, our method is a semi-exact method.

Based on this framework, we develop an algorithm called FastWClq. The pseudocode of FastWClq is shown in Algorithm 1. After the initialization, the algorithm executes a main loop until a limited time is reached, or an exact solution is found and proved. In the main loop, FastWClq works by interleaving the two sub-algorithms, as shown in Figure 1. The algorithm for clique finding in FastWClq adopts a construct-and-cut technique. In each iteration (line 4), it constructs a maximal clique step by step. Also, the construction procedure may be cut before reaching a maximal clique, using pruning techniques by computing upper bounds. We introduce the details of the two sub-algorithms in the next two sections.

4. Clique Finding

In this section, we describe our algorithm for clique finding. We propose a novel method called construct-and-cut for quickly finding a high quality clique. Also, when a clique of higher weight (compared to the best found one) is located, we further improve the clique by trying possible local modifications.

Algorithm 1: FastWClq(G, *cutoff*)

Input: vertex-weighted graph $G = (V, E, w)$, the *cutoff* time
Output: A clique of G

```
1 initialization;
2 while elapsed time < cutoff do
3     while w(C) ≤ w(C*) do
4         C := FindClique(G);
5     C* := C;
6     G := ReduceGraph(G, w(C*));
7     if G becomes empty then
8         return C*;      //exact solution
9 return C*;
```

Before going to the details of the construct-and-cut algorithm, let us first introduce some notation and definitions.

- C: the current clique under construction.
- *StartSet*: the set containing candidate vertices which can serve as a starting vertex to construct a clique. In the beginning of the algorithm, *StartSet* is initialized as $V(G)$.
- $CandSet = \bigcap_{v \in C} N(v)$, i.e., each vertex in *CandSet* is adjacent to all vertices in C. This is the set of candidate vertices that can be added to extend the current clique.
- The *effective neighborhood* of vertex v: is defined as $N(v) \cap CandSet$. The concept is very important, as $w(N(v) \cap CandSet)$ is used in both pruning a construction procedure and evaluating the quality of candidate vertices.

In our clique construction procedure (Algorithm 2), the algorithm first pops a random vertex from *StartSet* to serve as the starting vertex from which a clique will be extended, if *StartSet* is not empty (line 4). If *StartSet* becomes empty, which means all vertices have been used as the starting vertex, then another round of clique construction begins by resetting *StartSet* to $V(G)$, and we adjust our strategy parameter (lines 1-3). After the starting vertex u is chosen, the clique is initialized with the vertex, and *CandSet* is initialized as $N(u)$ (lines 5-6). Then the clique is extended iteratively by each time adding a vertex $v \in CandSet$, until *CandSet* becomes empty (lines 7-12). Also, we use a cost-effective upper bound to prune the procedure (line 9) — the construction procedure is cut before the normal termination. Obviously, $w(C) + w(v) + w(N(v) \cap CandSet)$ is an upper bound on weight of any clique extended from C by adding v and more vertices. We would like to note that, the construct-and-cut method has recently been used to generate initial assignments for the Boolean Satisfiability (SAT) problem (Cai, Luo, Zhang, & Zhang, 2021), which is parallel to this work.

4.1 Choosing a Solution Vertex

An important component in FindClique is the procedure ChooseSolutionVertex (Algorithm 3), which selects a vertex from *CandSet* to extend the current clique. We call such a vertex a *solu-*

Algorithm 2: FindClique(G)

```
if StartSet = ∅ then
    StartSet := V(G);
    Adjust BMS parameter t;
u := pop a random vertex from StartSet;
C := {u};
CandSet := N(u);
while CandSet ≠ ∅ do
    v := ChooseSolutionVertex(CandSet, t);
    if w(C) + w(v) + w(N(v) ∩ CandSet) ≤ w(C*) then break;
    C := C ∪ {v};
    CandSet := CandSet \ {v};
    CandSet := CandSet ∩ N(v);
if w(C) ≥ w(C*) then
    C := ImproveClique(C);
return C;
```

Algorithm 3: ChooseSolutionVertex($CandSet$, t)

```
if |CandSet| < t then
    return a vertex v ∈ CandSet with the greatest b̂ value;
v* := a random vertex in CandSet;
for iteration := 1 to t − 1 do
    v := a random vertex in CandSet;
    if b̂(v) > b̂(v*) then v* := v;
return v*;
```

tion vertex. To this end, we propose a novel function to estimate the benefit of vertices, and a dynamic BMS heuristic to choose a solution vertex.

Benefit Estimation Function. For choosing a vertex to add into the current clique, we need to estimate the benefit of adding a vertex, and then we choose the one with the best estimated benefit. We define the benefit of adding a vertex v as $benefit(v) = w(C_f) - w(C)$, where C_f is the final clique grown from $C \cup \{v\}$ by the construction procedure.

An ideal strategy is to pick the vertex with the best benefit at each iteration to extend the clique. However, we cannot know the true benefit value of a vertex until we finish the construction procedure. In order to compare candidate vertices at the current iteration, we propose a function to estimate the benefit of adding a vertex. The function is based on two considerations:

1. If a candidate vertex v is added into the clique C, the weight of C is increased by $w(v)$, which is a trivial lower bound of $benefit(v)$.

2. Suppose a candidate vertex v is selected to be added into the clique C. The best possible weighted clique grown from $C \cup \{v\}$ is $C \cup \{v\} \cup (N(v) \cap CandSet)$, for which the weight is

$w(C)+w(v)+w(N(v)\cap CandSet)$. Thus, an upper bound of $benefit(v)$ is $w(v)+w(N(v)\cap CandSet)$.

An estimation function should take into account both the lower bound and upper bound of $benefit(v)$. A simple and intuitive function which embodies this principle is to take the average over these two bounds:

$$\begin{aligned}\hat{b}(v) &= \frac{w(v)+w(v)+w(N(v)\cap CandSet)}{2}\\ &= w(v)+w(N(v)\cap CandSet)/2.\end{aligned}$$

Dynamic BMS Heuristic. We choose the solution vertex based on the $\hat{b}$ values of candidates, according to a dynamic BMS heuristic. The original BMS heuristic is a probabilistic strategy which returns the best element from multiple samples. Cai (2015) theoretically showed that BMS can approximate the best selection strategy very well in $O(1)$ time. The probability that the BMS heuristic chooses a vertex whose $\hat{b}(v)$ value is better than ρ of the vertices in *CandSet* is $Pr(E) > 1-\rho^t$. Another advantage of the BMS heuristic is that we can control the greediness of the algorithm by adjusting the parameter t. However, this has not been exploited previously, and previous algorithms with BMS adopt a static parameter, that is, the number of samplings t stays the same (Cai, 2015; Wang et al., 2016).

In general, a greater t value indicates a greater greediness and more computation time. Based on this observation, we propose a dynamic BMS heuristic. In our algorithm, we start from a small t value (denoted as t_0), so that the algorithm works fast. Whenever *StartSet* becomes empty, which means we do not find a better clique with this t value, we double t (i.e., $t := 2t$), with a upper bound limitation t_{max}, to make the algorithm construct cliques in a greedier way.

4.2 Improving the Clique

In the clique construction procedure, if a better clique C (compared to the best-found one) is obtained, we try to further improve it by examining the possible improvement with respect to each vertex in the clique. Specifically, for each vertex $v \in C$, we examine whether the following modification results in a larger-weight clique:

We remove v from C, and then *CandSet* is set to the common neighborhood of $C \setminus \{v\}$ accordingly. Then, find a maximum weight clique C'_v from *CandSet*, using a branch-and-bound method (the bound is given by weighted coloring, which will be introduced in Section 5.2 in detail). If $w(C'_v) > w(v)$, then a better clique $(C \setminus \{v\}) \cup C'_v$ is found and we update C accordingly. Otherwise, C remains unchanged.

Such examinations of possible improvements are executed for all vertices $v \in C$, and C is updated whenever a modification with improvement is identified. This procedure terminates when all vertices $v \in C$ are examined. As the size of *CandSet* is relatively small, this procedure is very fast.

5. Graph Reduction

By applying sound data reduction rules (which usually depend on a clique in hand), a graph can be reduced to a (sometimes significantly) smaller graph while keeping the optimal solution. This is desirable as algorithms can solve the original instance by solving a smaller and easier one. Many existing reduction rules for weighted problems rely on local structure (Lamm et al., 2019; Li et al., 2020), whereas our data reduction rules exploit lower bounds and upper bounds to reduce the graph

globally. While the lower bound is simply the weight of the current best clique, much effort is devoted to designing upper bound functions. In this section, we introduce a graph reduction algorithm, which relies on three upper bounds: a trivial bound, a branching-based bound and a coloring-based bound; the latter two bounds are proposed in this work.

Definition 1 (A Generic Upper Bound, UB)**.** *Given a vertex-weighted graph $G = (V,E,w)$ and a vertex $v \in V(G)$, we let $UB(v)$ denote an (integer) upper bound on the weight of any clique containing v. That is, $UB(v) \geq \max\{w(C) \mid C \text{ is a clique of } G, v \in C\}$.*

Now, we consider the below reduction rule, which is a general rule and should be used with the help of a lower bound and a upper bound.

General Reduction Rule. *Given a vertex-weighted graph $G = (V,E,w)$ and a clique C_0 in G, $\forall v \in V(G)$, if there is an upper bound $UB(v)$ such that $UB(v) \leq w(C_0)$, then delete v and its incident edges from G.*

The above rule indeed represents a family of reduction rules. In order to obtain an applicable concrete rule, we need to specify the upper bound function and the input clique. We use the notation $Rule(UB,C_0)$ to denote a concrete rule where UB is the upper bound function and C_0 is the input clique.

Proposition 1 (Soundness of the General Reduction Rule)**.** *Let G be a vertex-weighted graph, G' the resulting graph by applying $Rule(UB,C_0)$ on G, and let w^* be the weight of a maximum weight clique of graph G, and $C^*_{G'}$ the maximum weight clique of G'. Then, $w^* = \max\{w(C_0), w(C^*_{G'})\}$.*

Proof. If $w(C_0) = w^*$, then the proposition obviously holds. Now we consider the case $w(C_0) < w^*$. For graph G and a vertex $v \in V(G)$, let C^*_v be a clique with the maximum weight among cliques containing vertex v. If a function UB satisfies Definition 1, we have $w(C^*_v) \leq UB(v)$. On the other hand, any vertex deleted by $Rule(UB,C_0)$ satisfies $UB(v) \leq w(C_0)$, and thus $w(C^*_v) \leq UB(v) \leq w(C_0) < w^*$, meaning that v cannot be contained in any clique with weight w^*. Thus, any vertex that is in a clique with weight w^* remains in G', so $w^* = w(C^*_{G'})$. ∎

The above proposition shows that any specialization of the general reduction rule (for any valid upper bound) is sound with respect to keeping the optimal solution of the instance. Additionally, the proposition leads to the following corollary.

Corollary 1. *Let G' be the resulting graph after applying $Rule(UB,C_0)$ on vertex-weighted graph G, if $V(G') = \emptyset$, then C_0 is a maximum weight clique of G.*

5.1 The Trivial Upper Bound and a Branching-Based Upper Bound

Given a clique in hand (found by a clique-finding algorithm), in order to apply reduction rules, the focus is how to compute an upper bound. Since any clique grown from vertex v can only contain vertices in $N(v)$, a trivial upper bound function is

$$UB_0(v) = w(N[v]).$$

A tighter bound can be obtained by considering a branching-based upper bound function. For a vertex v, we consider its neighboring vertex with the maximum weight (denoted as n^*). The idea is

that, for any clique C containing v, it either contains n^* or it does not. For either case, we can have a tighter upper bound than $UB_0(v)$, and finally we get the larger (worse) one as the upper bound. We divide the cases on n^* in order to balance the bounds of the two cases. Formally, we propose a branching-based upper bound as follows:

$$\begin{aligned} UB_1(v) &= \max\{w(N[v]) - w(n^*), w(v) + w(n^*) + w(N(v) \cap N(n^*))\} \\ &= w(v) + \max\{w(N(v)) - w(n^*), w(n^*) + w(N(v) \cap N(n^*))\}. \end{aligned}$$

This upper bound can be easily understood: for any clique containing v, if it does not contain n^*, then an upper bound on the weight of the clique is $w(N[v]) - w(n^*)$; if the clique contains n^*, then, besides v and n^*, the clique can only contain those vertices that are adjacent to both v and n^*. Thus, an upper bound on the weight of the clique is $w(v) + w(n^*) + w(N(v) \cap N(n^*))$. Finally, the worst case can be used as an upper bound on the weight of any clique containing v.

Note that we use an adjacency list instead of an adjacency matrix representation, to ensure that the graph fits into memory. So, checking whether a vertex $y \in N(v)$ is in $N(n^*)$, i.e., whether y and n^* are neighbors, requires $O(\min\{d(y), d(n^*)\})$ time. Note that a straightforward, but inefficient, way to compute $N(v) \cap N(n^*)$ is by checking for each vertex $y \in N(v)$ if $y \in N(n^*)$—which is quadratic in the degree. Rather than use the above implementation, we compute $N(v) \cap N(n^*)$ in linear time with two scans on the smaller set and one on the larger one, using indicators.

5.2 Improving Branching-Based Upper Bound by Weighted Coloring

The branch-based upper bound UB_1 can be improved by further using coloring on each situation, leading to a tighter upper bound. We describe this new upper bound in this subsection.

For a vertex-weighted graph G, suppose we have a proper vertex coloring $A_1, A_2, \ldots, A_k$ of the graph, where A_i is the set of vertices with color i, then an upper bound on the weight of the maximum clique weight is as follows:

$$UB_c(G) = \sum_{i=1}^{k} \max_{u \in A_i}\{w(u)\}.$$

This is easy to understand, as vertices with the same color composes an independent set and thus at most one of them can be selected in a valid clique. By integrating this weighted-coloring-based upper bound, we can further make the branching-based upper bound tighter:

$$UB_2(v) = w(v) + \max\{UB_c(G[N(v) \setminus \{n^*\}]), w(n^*) + UB_c(G[N(v) \cap N(n^*)])\}.$$

The quality of the UB_2 upper bound depends on the coloring solution to obtain the UB_c values. To leverage the power of this upper bound function, we need to compute a UB_c value as small as possible. This indeed is a variant of the vertex coloring problem, which is referred to as the weighted-vertex coloring problem (Demange et al., 2002) or max-coloring problem (Hsu & Chang, 2016).

Definition 2 (Weighted-Coloring Problem)**.** *The max-coloring problem is, for a given vertex-weighted graph, to find a proper vertex coloring $A_1, A_2, ..., A_k$ which minimizes $\sum_{i=1}^{k} \max_{u \in A_i}\{w(u)\}$.*

In this work, since the algorithm for solving Weighted-Coloring is called many times, we employ a light-weight algorithm, which is described as follows. Sort the vertices in a weight decreasing

order, breaking ties by vertex degree. According to this order, the vertices are colored one by one. For each vertex v, suppose that k is the number of colors have been used so far, we find a color with the lowest value in $[1,k]$ that gives a proper coloring for v. If no such proper coloring exists, we color v with a new color $k+1$.

5.3 The Graph Reduction Algorithm

The graph reduction algorithm is depicted in Algorithm 4. All the three upper bounds are used. UB_0 requires little overhead, while UB_1 and UB_2 require more computation time but are tighter. Therefore, when considering a vertex, we first use the UB_0-based reduction rule, and if this cannot delete the vertex then we apply the rule based on UB_1 and UB_2[2].

Algorithm 4: ReduceGraph(G, C_0)

Input: vertex-weighted graph $G = (V,E,w)$, a clique C_0
Output: A simplified graph of G

```
1  foreach v ∈ V(G) do
2      if UB_0(v) ≤ w(C_0) or UB_1(v) ≤ w(C_0) or UB_2(v) ≤ w(C_0) then
3          RmQueue := RmQueue ∪ {v};
4  while RmQueue ≠ ∅ do
5      u := dequeue a vertex from RmQueue;
6      delete u and its incident edges from G;
7      foreach v ∈ N_r(u) do
8          if UB_0(v) ≤ w(C_0) or UB_1(v) ≤ w(C_0) or UB_2(v) ≤ w(C_0) then
9              RmQueue := RmQueue ∪ {v};
10 return G;
```

Our reduction algorithm works in an iterative fashion, with a queue called *RmQueue* which contains vertices to be deleted. In the beginning, the algorithm enqueues into *RmQueue* all vertices satisfying at least one of the reduction rules. Then, a loop is carried out until *RmQueue* becomes empty. Each iteration of the loop dequeues a vertex u from *RmQueue*, and deletes u and all its incident edges from G. After a vertex u is deleted, we check its remained neighborhood $N_r(u)$ (the set containing all neighbors of u that have not been removed from the graph yet), and enqueue to *RmQueue* all vertices in $N_r(u)$ that satisfy at least one of the reduction rules.

According to Corollary 1, if the ReduceGraph algorithm returns an empty graph, that means the clique found is a maximum weight clique of the input graph. However, there are cases that FastWClq finds a maximum weight clique but ReduceGraph cannot reduce the graph to empty, because the reduction rules are incomplete[3].

2. In practice, a trick to accelerate the procedure (slightly) for large-sized graphs is to first use UB_0 to reduce the graph to a certain size, after which UB_1 and UB_2 are used.
3. Note that if this weren't the case and the reduction rules always resulted in an empty graph, then this would give a polynomial time algorithm for the decision variant of MWCP and P = NP.

6. Experimental Evaluation

In this section, we carry out experiments to evaluate our proposed algorithm, FastWClq, on a broad range of large sparse real-world graphs. First, we introduce the benchmarks used in our experiments, and present some preliminaries about our experimental evaluation. We then compare FastWClq with state-of-the-art exact and heuristic algorithms. Our experiments show that FastWClq performs very well on a considerable portion of the selected instances.

6.1 Benchmarks

For our experiments, we collect two benchmarks of large sparse graphs, which are originally unweighted, and then we generate weights for vertices in these graphs. The two benchmarks of large sparse graphs are introduced as follows:

- **Network Data Repository:** This benchmark was downloaded from the Network Data Repository[4] (Rossi & Ahmed, 2015). We consider the 187 graphs used in testing WLMC (Jiang et al., 2017), and select the graphs with at least 100 000 vertices and 1000 000 edges. Some graphs in this benchmark are bipartite graphs, and we choose to ignore them. All these graphs are generated from real-world applications, which can be grouped into eleven classes, including biological networks, collaboration networks, interaction networks, infrastructure networks, Amazon recommendation networks, scientific computation networks, social networks, technological networks, and web-crawl graphs. Many of these real-world graphs have millions of vertices and tens of millions of edges, and each graph is quite sparse. Calculating the density $m/\binom{n}{2}$ of each graph, the average density of these graphs is 0.00859, while the maximum density is 0.347. We also calculate the average degree $2m/n$ for each graph; over all graphs, this figure is 26.15 on average, while the maximum is 181.19.

- **Koblenz Network Collection:** This benchmark was downloaded from the Koblenz Network Collection (KONECT)[5] (Kunegis, 2013), which was collected by the Institute of Web Science and Technologies at the University of Koblenz-Landau in order to perform research in network science and related fields. The networks of KONECT cover many diverse areas such as social networks, hyperlink networks, authorship networks, physical networks, interaction networks, and communication networks. We downloaded all the graphs from KONECT and converted them into undirected graphs, except the bipartite graphs. These large graphs are also quite sparse. The average density is 0.000916, while the maximum one is 0.022332. The average degree is 18.25, while the maximum one is 211.44.

In our experiments, we focus on the large graphs, so we select those graphs with at least 100 000 vertices from each benchmark, resulting in 65 graphs from Network Data Repository and 80 graphs from KONECT benchmark. The sizes of these large graphs are given in Tables 1 and 2.

To obtain the corresponding vertex-weighted graphs, we generate vertex weights to the graphs of the above two benchmarks in two different ways.

- **v_mod_200:** For the i^{th} vertex v_i, $w(v_i) = (i \mod 200) + 1$.

4. http://www.networkrepository.com
5. http://konect.uni-koblenz.de

Table 1. Details of graphs with at least 100 000 vertices from Network Data Repository

Instance	$\|V\|$	$\|E\|$
bn-human-BNU_1_0025865_session_1-bg	1 398 408	42 296 922
bn-human-BNU_1_0025865_session_2-bg	1 717 207	22 855 526
ca-coauthors-dblp	540 486	15 245 729
ca-dblp-2012	317 080	1 049 866
ca-hollywood-2009	1 069 126	56 306 653
channel-500x100x100-b050	4 802 000	42 681 372
dbpedia-link	11 621 692	78 621 046
delaunay_n22	4 194 304	12 582 869
delaunay_n23	8 388 608	25 165 784
delaunay_n24	16 777 216	50 331 601
friendster	8 658 744	45 671 471
hugebubbles-00020	21 198 119	31 790 179
hugetrace-00010	12 057 441	18 082 179
hugetrace-00020	16 002 413	23 998 813
inf-europe_osm	50 912 018	54 054 660
inf-germany_osm	11 548 845	12 369 181
inf-road-usa	23 947 347	28 854 312
inf-roadNet-CA	1 957 027	2 760 388
inf-roadNet-PA	1 087 562	1 541 514
rec-dating	168 792	17 351 416
rec-epinions	755 761	13 396 042
rec-libimseti-dir	220 970	17 233 144
rgg_n_2_23_s0	8 388 608	63 501 393
rgg_n_2_24_s0	16 777 216	132 557 200
rt-retweet-crawl	1 112 702	2 278 852
sc-ldoor	952 203	20 770 807
sc-msdoor	415 863	9 378 650
sc-pwtk	217 891	5 653 221
sc-rel9	5 921 786	23 667 162
sc-shipsec1	140 385	1 707 759
sc-shipsec5	179 104	2 200 076
soc-FourSquare	639 014	3 214 986
soc-LiveMocha	104 103	2 193 083

Instance	$\|V\|$	$\|E\|$
soc-buzznet	101 163	2 763 066
soc-delicious	536 108	1 365 961
soc-digg	770 799	5 907 132
soc-dogster	426 820	8 543 549
soc-flickr-und	1 715 255	15 555 041
soc-flickr	513 969	3 190 452
soc-flixster	2 523 386	7 918 801
soc-lastfm	1 191 805	4 519 330
soc-livejournal-user-groups	7 489 073	112 305 407
soc-livejournal	4 033 137	27 933 062
soc-ljournal-2008	5 363 186	49 514 271
soc-orkut-dir	3 072 441	117 185 083
soc-orkut	2 997 166	106 349 209
soc-pokec	1 632 803	22 301 964
soc-sinaweibo	58 655 849	261 321 033
soc-twitter-higgs	456 631	12 508 442
soc-youtube-snap	1 134 890	2 987 624
soc-youtube	495 957	1 936 748
socfb-A-anon	3 097 165	23 667 394
socfb-B-anon	2 937 612	20 959 854
socfb-uci-uni	58 790 782	92 208 195
tech-as-skitter	1 694 616	11 094 209
tech-ip	2 250 498	21 643 497
twitter_mpi	9 862 152	99 940 317
web-arabic-2005	163 598	1 747 269
web-baidu-baike	2 141 300	17 014 946
web-it-2004	509 338	7 178 413
web-uk-2005	129 632	11 744 049
web-wikipedia-growth	1 870 709	36 532 531
web-wikipedia2009	1 864 433	4 507 315
web-wikipedia_link	2 936 413	86 754 664
wikipedia_link_en	27 154 756	31 024 475

Table 2. Details of graphs with at least 100 000 vertices from KONECT benchmark

Instance	$\|V\|$	$\|E\|$
actor-collaboration	382 219	15 038 083
amazon0601	403 394	2 443 408
as-skitter	1 696 415	11 095 298
citeseer	384 413	1 736 145
com-amazon	334 863	925 872
com-dblp	317 080	1 049 866
com-youtube	1 134 890	2 987 624
dbpedia-all	3 966 895	12 610 982
dbpedia-link	18 268 991	126 890 209
digg-friends	1 923 999	3 192 495
douban	154 908	327 162
elec	105 144	200 184
email-EuAll	265 214	364 481
enron	307 636	517 819
epinions	132 768	713 088
facebook-wosn-links	397 655	1 150 959
facebook-wosn-wall	914 890	1 051 350
flickrEdges	105 938	2 316 948
flickr-growth	2 303 059	22 838 410
flickr-links	1 715 254	15 551 249
flixster	2 523 386	7 918 801
hyves	1 402 673	2 777 419
lasagne-yahoo	653 260	2 931 698
libimseti	220 970	10 131 013
link-dynamic-dewiki	19 521 519	61 151 432
link-dynamic-frwiki	14 371 085	44 187 494
link-dynamic-itwiki	8 549 267	28 858 941
link-dynamic-nlwiki	6 126 550	18 873 552
link-dynamic-plwiki	6 231 281	20 768 067
link-dynamic-simplewiki	380 731	1 137 693
livejournal-links	5 204 175	48 709 621
livemocha	104 103	2 193 083
lkml-reply	909 627	1 041 696
loc-gowalla_edges	196 591	950 327
munmun_digg_reply	114 340	169 097
munmun_twitter_social	465 017	833 540
orkut-links	3 072 441	117 184 899
patentcite	3 774 768	16 518 947
petster-carnivore	623 766	15 695 166
petster-cat-uniq	149 700	5 448 197
petster-dog-uniq	426 820	8 543 549
roadNet-CA	1 965 206	2 766 607
roadNet-PA	1 088 092	1 541 898
roadNet-TX	1 379 917	1 921 660
slashdot-threads	141 428	206 918
soc-LiveJournal1	4 846 609	42 851 237
soc-pokec-relationships	1 632 803	22 301 964
trec-wt10g	1 601 787	6 679 248
web-BerkStan	685 230	6 649 470
web-Google	875 713	4 322 051
web-NotreDame	325 729	1 090 108
web-Stanford	281 903	1 992 636
wikipedia-growth	1 872 907	36 534 729
wikipedia_link_de	3 225 565	65 759 634
wikipedia_link_en	12 150 976	288 257 813
wikipedia_link_fr	3 023 165	83 455 052
wikipedia_link_it	1 865 965	68 022 541
wikipedia_link_ja	1 610 638	56 231 610
wikipedia_link_pl	1 529 135	42 188 631
wikipedia_link_pt	1 603 222	38 633 429
wikipedia_link_ru	2 853 118	63 058 425
wikisigned-k2	138 593	482 888
wiki-Talk	2 394 385	4 659 565
wiki_talk_ar	2 868 931	3 307 474
wiki_talk_de	6 757 916	7 744 939
wiki_talk_en	26 373 842	31 532 851
wiki_talk_es	2 661 000	3 228 049
wiki_talk_fr	5 637 678	6 484 425
wiki_talk_it	3 671 312	4 309 561
wiki_talk_nl	1 615 381	1 900 124
wiki_talk_pt	2 694 832	3 533 288
wiki_talk_ru	2 470 379	2 862 384
wiki_talk_zh	3 175 241	3 644 578
wordnet-words	146 005	656 999
youtube-links	1 138 494	2 990 287
youtube-u-growth	3 223 788	9 375 577
zhishi-baidu-internallink	2 141 300	17 014 946
zhishi-baidu-relatedpages	415 641	2 374 044
zhishi-hudong-internallink	1 984 484	14 428 382
zhishi-hudong-relatedpages	2 452 715	18 690 759

- **normal random:** $w(v)$ subjects to truncated normal distribution: $w(v) \sim N(\mu, \sigma^2)$, where $w(v) \in (1, +\infty)$, $\mu = 100$ and $\sigma = 25$.

Therefore, we obtain totally four benchmarks of vertex-weighted graphs, which are denoted by "Repository_200", "Repository_normal", "KONECT_200" and "KONECT_normal", respectively.

We realize that the way of generating weights may have considerable impact on the performance of algorithms. Indeed, a recent paper has commented on this phenomenon (McCreesh et al., 2017). In particular, McCreesh et al. (2017) mentioned that, when weights are chosen to be between 1 and 200, the vertex weights would be expected to be more important than degree. So, we also use a normal distribution method to generate weights, for which we would expect degrees become more important. We use these two ways to generate weights, in order to evaluate algorithms with two typical situations.

6.2 Competitors

In the following experimental comparison, we compare FastWClq with three state-of-the-art MWCP algorithms, namely WLMC (Jiang et al., 2017), TSM-MWC (Jiang et al., 2018) and SCCWalk4L (Wang et al., 2020).

- WLMC (Jiang et al., 2017) is an exact algorithm for solving the MWCP problem on large graphs. It is the first exact solver that can obtain very good solution values on large graphs, compared with some heuristic solvers. WLMC contains a preprocess procedure to order the vertex and then to reduce the size of graph. It also uses the incremental vertex-weight splitting technique to reduce the number of branches in the search space. The source code of WLMC is available online[6].

- TSM-MWC (Jiang et al., 2018) is a recent BnB-based exact MWCP algorithm which incorporates a novel two-stage MaxSAT reasoning approach. According to the literature, TSM-MWC is the currently best exact algorithm for MWCP on large instances. As reported by Jiang et al. (2018), TSM-MWC significantly outperforms the WLMC algorithm on a broad range of large real-world graphs. This is the only exact algorithm that shows significantly better performance than the milestone exact MWCP algorithm WLMC on large real-world graphs. The source code of TSM-MWC is available online[7].

- SCCWalk4L (Wang et al., 2020) is a very recent local search algorithm for MWCP, which shows significantly better performance on large graphs than previous heuristic algorithms. SCCWalk4L integrates powerful local search techniques including configuration checking, walk perturbation as well as BMS. The source code of SCCWalk4L was provided by the authors.

Seen from the literature, WLMC and TSM-MWC establish the latest state of the art in exact algorithms for MWCP on large graphs. As for heuristic algorithms, SCCWalk4L represents the latest state of the art in solving large graphs. It is reported that SCCWalk4L outperforms other recent heuristic algorithms including RRWL (Fan et al., 2017), ReTS1 and ReTS2 (Zhou et al., 2017).

6. http://www.mis.u-picardie.fr/~cli/wlmc2.zip
7. http://home.mis.u-picardie.fr/~cli/tsm-release.zip

6.3 Experimental Preliminaries

FastWClq is implemented in C++, and its source code is available online[8]. Parameters t_0 and t_{max} for dynamic BMS heuristic are set to 4 and 64 ($= 2^6$). Three competitors including WLMC, TSM-MWC and SCCWalk4L were implemented in C++ by their authors. For SCCWalk4L, we used the following default values: $\ell = 4000$ and $r = 50$ (Wang et al., 2020). All algorithms are complied with g++ version 4.7 with the `-O3` optimization flag. FastWClq and SCCWalk4L both are run 10 times on each graph. For the exact algorithms TSM-MWC and WLMC, we run them once on each graph. We test the algorithms with time limit of 100 seconds, 300 seconds and 600 seconds, where the run time of each algorithm includes the time for reading graphs (as some algorithms do non-trivial works including building complex data structure and some initializations during reading the graph). We report the detailed results for the time limit of 100 seconds, while give a summary on all time limits.

The experiments are carried out on a computing cluster consisting of computing nodes equipped with dual 56-core, 2.00GHz Intel Xeon E7-4830 CPUs, 35 MB L3 cache and 256 GB RAM, running Ubuntu version 16.04.5 LTS.

For each graph, we report the largest clique weight ("best") found by each algorithm, the average clique weight over all runs ("avg") and the average run time when algorithm obtains the largest clique weight. If an algorithm fails to provide a solution for an instance, then the corresponding column is marked as "N/A". If an algorithm **proves** the optimality of its solution, the corresponding column is marked with a "*".

6.4 Experimental Results

In this subsection, we will report the experiment results of FastWClq and the state-of-the-art competitors on large sparse graphs.

6.4.1 RESULTS ON REPOSITORY_200

The results on Repository_200 are presented in Table 3. FastWClq performs better on 29 instances than SCCWalk4L under the same cutoff time (100s). For the remaining instances, FastWClq finds better average solution values than SCCWalk4L, with only three exceptions. Observed from the results, FastWClq computes larger-weight cliques than the two exact solvers TSM-MWC and WLMC for 18 and 11 instances, respectively, while TSM-MWC and WLMC find better cliques for only the two instances soc-flickr-und and web-wikipedia_link. FastWClq computes the optimal solution for 53 instances, more than any other algorithm; although, it only *proves* the optimality of its solutions in 36 cases. Although they find fewer optimal solutions, TSM-MWC, and WLMC are highly successful at proving optimality, proving optimality in 45 and 52 instances, respectively.

6.4.2 RESULTS ON REPOSITORY_NORMAL

The comparison of FastWClq, TSM-MWC, WLMC and SCCWalk4L on Repository_normal is given in Table 4. Once again, FastWClq has the best performance. Firstly, FastWClq finds cliques of larger weight than SCCWalk4L for 32 graphs, while FastWClq finds a worse solution than SCCWalk4L for only one graph. FastWClq finds solutions of the same or higher quality than the two exact solvers TSM-MWC and WLMC on all but one instance (instance soc-flickr-und), and on 46

8. `http://lcs.ios.ac.cn/~caisw/CLQ.html`

Table 3. Experiment results of FastWClq and the state-of-the-art competitors on Repository_200 benchmark.

Instance	FastWClq		TSM-MWC		WLMC		SCCWalk4L	
	size	time	size	time	size	time	size	time
bn-human...1-bg	**20184** (**19678**)	31.08	19547 (19547)	89.88	19185 (19185)	81.88	14506 (13568.6)	99.53
bn-human...2-bg	**19189** (**19189**)	16.54	16396 (16396)	70.58	16102 (16102)	99.19	11804 (9478.8)	99.51
ca-coauthors-dblp	37884* (37884)	**5.07**	37884* (37884)	13.09	37884* (37884)	13.09	37884* (37884)	26.88
ca-dblp-2012	14108* (14108)	**0.96**	14108* (14108)	1.29	14108* (14108)	1.39	14108* (14108)	2.11
ca-hollywood-2009	**222720*** (**222720**)	55.18	**222720*** (**222720**)	67.36	**222720*** (**222720**)	69.48	15367 (11009.9)	99.61
channel-500x100x100-b050	796* (796)	**5.72**	796* (796)	43.78	796* (796)	41.89	796 (796)	69.47
dbpedia-link	**4156** (**2507.2**)	93.54	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
delaunay_n22	796* (**796**)	5.58	796* (**796**)	15.22	796* (**796**)	14.72	796* (750.3)	47.86
delaunay_n23	798* (**798**)	12.78	798* (**798**)	29.99	798* (**798**)	29.68	798* (672)	80.99
delaunay_n24	**797*** (**797**)	25.36	N/A (N/A)	N/A	**797*** (**797**)	65.73	566 (465.9)	97.28
friendster	**5511*** (4982.4)	86.42	**5511*** (**5511**)	79.09	**5511*** (**5511**)	81.97	567 (500)	99.91
hugebubbles-00020	**400*** (**400**)	49.83	N/A (N/A)	N/A	**400*** (**400**)	60.6	399 (384.8)	92.18
hugetrace-00010	**400*** (**400**)	19.5	N/A (N/A)	N/A	**400*** (**400**)	28.19	399 (399)	56.15
hugetrace-00020	**400*** (**400**)	29.17	N/A (N/A)	N/A	**400*** (**400**)	43.19	**400** (398.9)	79.99
inf-europe_osm	**646*** (**646**)	55.99	N/A (N/A)	N/A	**646*** (**646**)	85.98	267 (267)	91.65
inf-germany_osm	**597*** (**597**)	14.3	N/A (N/A)	N/A	**597*** (**597**)	19.89	**597*** (524.6)	63.96
inf-road-usa	**766*** (**766**)	29.89	N/A (N/A)	N/A	**766*** (**766**)	51.19	**766*** (537.4)	88.23
inf-roadNet-CA	752* (752)	**1.56**	752* (752)	4.19	752* (752)	4.38	752* (752)	6.98
inf-roadNet-PA	669* (669)	**0.92**	669* (669)	2.18	669* (669)	2.39	669 (669)	3.12
rec-dating	1699 (1524.2)	40.39	1699* (**1699**)	24.59	1699* (**1699**)	25.14	1699 (**1699**)	24.96
rec-epinions	1054 (998.4)	42.36	1054* (**1054**)	73.79	1054* (**1054**)	76.12	1054 (**1054**)	50.33
rec-libimseti-dir	1938 (1781.2)	44.83	1938* (**1938**)	29.16	1938* (**1938**)	28.90	1938 (**1938**)	27.94
rgg_n_2_23_s0	**2407*** (**2407**)	27.18	**2407*** (**2407**)	65.94	**2407*** (**2407**)	64.92	1103 (1042.67)	97.21
rgg_n_2_24_s0	**2514*** (**2514**)	59.37	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
rt-retweet-crawl	1367* (1367)	4.47	1367* (1367)	**3.63**	1367* (1367)	3.85	1367 (1367)	19.4
sc-ldoor	4081* (**4081**)	3.91	4081* (**4081**)	17.39	4081* (**4081**)	17.48	4081 (4054.4)	60.65
sc-msdoor	4088* (**4088**)	1.46	4088* (**4088**)	7.49	4088* (**4088**)	7.49	4088 (4086.6)	29.51
sc-pwtk	4620* (4620)	**0.49**	4620* (4620)	4.34	4620* (4620)	4.48	4620 (4620)	8.28
sc-rel9	572* (**572**)	34.39	572* (**572**)	29.14	572* (**572**)	28.79	572 (417.2)	62.06
sc-shipsec1	3540* (3540)	**0.23**	3540* (3540)	1.21	3540* (3540)	1.48	3540* (3540)	2.57
sc-shipsec5	4524* (4524)	**0.18**	4524* (4524)	1.74	4524* (4524)	1.91	4524* (4524)	3.36
soc-FourSquare	**3064** (**3064**)	50.64	**3064*** (**3064**)	12.08	**3064*** (**3064**)	12.07	2025 (1873.9)	93.66
soc-LiveMocha	1784 (1784)	11.71	1784* (1784)	**2.58**	1784* (1784)	2.63	1784 (1784)	7.88
soc-buzznet	2981 (**2981**)	33.14	2981* (**2981**)	8.4	2981* (**2981**)	8.59	2981 (2977)	56.51
soc-delicious	1547 (**1547**)	1.03	1547* (**1547**)	1.25	1547* (**1547**)	1.48	1547 (1546.3)	34
soc-digg	5303 (5302.6)	48.5	5303* (**5303**)	14.41	5303* (**5303**)	14.98	5303 (5051.8)	58.17
soc-dogster	4418 (4400.8)	65.82	4418* (**4418**)	10.16	4418* (**4418**)	9.86	4418 (4364.8)	87.2
soc-flickr	7083 (7083)	40.22	7083* (7083)	**4.08**	7083* (7083)	4.55	7083 (7083)	18.53
soc-flickr-und	10126 (9860.6)	32.88	**10127*** (**10127**)	44.94	9329 (9329)	89.83	6906 (5309)	99.35
soc-flixster	3805 (3805)	**10.29**	3805* (3805)	10.87	3805* (3805)	10.93	3805 (3805)	32.43
soc-lastfm	1773 (1773)	10.62	1773* (1773)	6.26	1773* (1773)	**5.72**	1773 (1773)	14.51
soc-livejournal	**21368*** (**21368**)	43.02	**21368*** (**21368**)	39.31	**21368*** (**21368**)	39.27	9183 (3009.6)	95.03
soc-livejournal-user-groups	**957** (**860.4**)	76.83	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
soc-ljournal-2008	**40432** (**40432**)	57.37	**40432*** (**40432**)	68.06	**40432*** (**40432**)	70.63	2709 (1649.5)	95.48
soc-orkut	**4439** (**3628.8**)	91.29	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
soc-orkut-dir	**4262** (**3656.8**)	90.43	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
soc-pokec	**3191*** (**3191**)	37.64	**3191*** (**3191**)	33.42	**3191*** (**3191**)	31.85	2994 (1990.9)	86.46
soc-sinaweibo	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
soc-twitter-higgs	**8039*** (**8039**)	13.31	**8039*** (**8039**)	17.98	**8039*** (**8039**)	17.88	4727 (4699.7)	68.63
soc-youtube	1961* (1961)	2.49	1961* (1961)	**2.39**	1961* (1961)	2.88	1961 (1961)	8.13
soc-youtube-snap	1787* (1787)	**4.71**	1787* (1787)	4.78	1787* (1787)	4.88	1787 (1787)	13.87
socfb-A-anon	**2872*** (**2872**)	36.84	**2872*** (**2872**)	34.27	**2872*** (**2872**)	34.42	2295 (1992.3)	75.76
socfb-B-anon	**2662** (**2662**)	41.77	**2662*** (**2662**)	30.48	**2662*** (**2662**)	29.88	2513 (2075.4)	81.32
socfb-uci-uni	**453** (**331.6**)	93.28	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
tech-as-skitter	5703 (**5703**)	9.42	5703* (**5703**)	22.55	5703* (**5703**)	22.89	5703 (5666.8)	78.48
tech-ip	**668** (**585**)	48.4	N/A (N/A)	N/A	N/A (N/A)	N/A	249 (162)	44.21
twitter_mpi	**6510** (**3085.2**)	91.25	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
web-arabic-2005	10558* (10558)	**0.16**	10558* (10558)	1.49	10558* (10558)	1.28	10558* (10558)	2.17
web-baidu-baike	**3814*** (**3814**)	22.46	**3814*** (**3814**)	25.89	**3814*** (**3814**)	26.09	1743 (1656.2)	81.93
web-it-2004	45477* (45477)	**1.28**	45477* (45477)	5.89	45477* (45477)	5.99	45477* (45477)	15.68
web-uk-2005	54850* (54850)	**1.55**	54850* (54850)	9.28	54850* (54850)	9.39	54850* (54850)	15.08
web-wikipedia-growth	**4741** (**4741**)	54.21	**4741*** (**4741**)	63.98	**4741*** (**4741**)	64.08	1714 (1380.3)	97.7
web-wikipedia2009	3891* (3891)	8.68	3891* (3891)	**7.38**	3891* (3891)	7.59	3891 (3891)	15.18
web-wikipedia_link	51331 (25964.8)	76.25	89544 (89544)	99.78	**89545** (**89545**)	99.99	N/A (N/A)	N/A
wikipedia_link_en	**4624** (4618.1)	75.55	N/A (N/A)	N/A	**4624*** (**4624**)	68.78	1002 (707)	93.85

instances it finds these solutions faster. Focusing on the ability of proving optimality, FastWClq, TSM-MWC and WLMC prove the optimality of 36, 45 and 52 instances, respectively. The total number of instances provably solved to optimality by the exact algorithms TSM-MWC and WLMC is more than FastWClq for 17 instances. FastWClq proves the optimality of instance rgg_n_2_24_s0, while the two exact algorithms fail to find any solution within the time limit.

6.4.3 RESULTS ON KONECT_200

The experimental results are presented in Table 5, which shows that FastWClq is clearly the best solver on the KONECT_200 benchmark. FastWClq finds the largest-weight cliques for all instances in KONECT_200 except for the two instances flickr-link and libimseti. Furthermore, FastWClq finds larger-weight cliques than TSM-MWC, WLMC and SCCWalk4L for 31, 32, and 38 instances, respectively. For instances where FastWClq and a corresponding competitor obtain the same solution quality (i.e., the same maximum and same average solution values), FastWClq is faster than the other three algorithms, except for six instances. FastWClq proves the optimality of nine solutions that neither TSM-MWC and WLMC prove to be optimal, but FastWClq fails to prove optimality on 16 instances where TSM-MWC and WLMC both succeed in proving optimality.

Table 5. Experiment results of FastWClq and the state-of-the-art competitors on KONECT_200 benchmark.

Instance	FastWClq		TSM-MWC		WLMC		SCCWalk4L	
	size	time	size	time	size	time	size	time
actor-collaboration	25500* (25500)	**9.54**	25500* (25500)	25.99	25500* (25500)	25.55	25500 (25500)	76.5
amazon0601	2127* (2127)	**1.6**	2127* (2127)	2.79	2127* (2127)	2.96	2127* (2127)	3.9
as-skitter	5703 (5703)	**8.32**	5703* (5703)	22.79	5703* (5703)	22.49	5703 (5703)	73.96
citeseer	1679* (1679)	**1.72**	1679* (1679)	2.29	1679* (1679)	2.38	1679 (1679)	5.34
com-amazon	1301* (1301)	**0.51**	1301* (1301)	1.18	1301* (1301)	1.46	1301* (1301)	1.38
com-dblp	10668* (10668)	**0.32**	10668* (10668)	1.26	10668* (10668)	1.39	10668* (10668)	1.48
com-youtube	1967* (1967)	5.6	1967* (1967)	**4.28**	1967* (1967)	4.88	1967 (1967)	11.3
dbpedia-all	**2547*** (**2547**)	15.95	N/A (N/A)	N/A	N/A (N/A)	N/A	**2547*** (1196)	97.25
dbpedia-link	**1258** (**701.111**)	95.1	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
digg-friends	**3668** (**3666.5**)	43.72	N/A (N/A)	N/A	N/A (N/A)	N/A	203 (141.2)	12.15
douban	1197* (1197)	0.06	1197* (1197)	**0.01**	1197* (1197)	0.71	1197 (1197)	0.73
elec	1936 (1936)	**0.14**	1936* (1936)	2.79	1936* (1936)	2.88	1936 (1936)	7.32
email-EuAll	2082* (2082)	**0.19**	2082* (2082)	0.79	2082* (2082)	0.99	2082 (2082)	0.81
enron	3496 (3496)	**0.24**	3496* (3496)	18.59	3496* (3496)	18.75	3496 (3496)	16.41
epinions	10037 (10037)	**0.14**	10037* (10037)	0.98	10037* (10037)	1.08	10037 (10037)	6.88
facebook-wosn-links	3618 (3618)	**0.35**	3618* (3618)	42.18	3618* (3618)	42.31	3618 (3618)	27.5
facebook-wosn-wall	**1175*** (**1175**)	0.35	N/A (N/A)	N/A	N/A (N/A)	N/A	526 (232.1)	49
flickr-growth	**11508** (**11268.7**)	36.04	10917 (10917)	97.25	10678 (10678)	96.47	7924 (5498.4)	85.11
flickr-links	10027 (9904)	47.29	**10036*** (**10036**)	44.09	9242 (9242)	94.46	9225 (7940.7)	99.86
flickrEdges	60280* (60280)	**1.07**	60280* (60280)	2.48	60280* (60280)	2.58	60280* (60280)	14.7
flixster	3399 (3399)	10.29	3399* (3399)	10.29	3399* (3399)	**10.27**	3399 (3399)	26.39
hyves	1953* (1953)	**2.5**	1953* (1953)	4.67	1953* (1953)	4.88	1953 (1953)	7.76
lasagne-yahoo	**457*** (**457**)	10.76	**457*** (**457**)	32.5	**457*** (**457**)	31.83	444 (431.9)	52.35
libimseti	1092 (999)	60.69	**1099*** (**1099**)	94.78	**1099*** (**1099**)	95.47	**1099** (**1099**)	31.22
link-dynamic-dewiki	**11669** (**11669**)	70.89	N/A (N/A)	N/A	N/A (N/A)	N/A	178 (102.556)	94.81
link-dynamic-frwiki	**8637*** (**8637**)	66.24	N/A (N/A)	N/A	N/A (N/A)	N/A	662 (174)	71.39
link-dynamic-itwiki	**5776** (**5776**)	25.58	N/A (N/A)	N/A	N/A (N/A)	N/A	538 (203.4)	43.15
link-dynamic-nlwiki	**5118*** (**5118**)	15.18	N/A (N/A)	N/A	N/A (N/A)	N/A	199 (126.4)	28.93
link-dynamic-plwiki	**5202** (**5202**)	52.07	N/A (N/A)	N/A	N/A (N/A)	N/A	191 (135.5)	30.78
link-dynamic-simplewiki	3988* (3988)	**0.48**	3988* (3988)	23.44	3988* (3988)	23.7	3988 (3988)	19.79

(Continued on next page)

Table 5. (Continued) Experiment results of FastWClq and the state-of-the-art competitors on KONECT_200 benchmark.

Instance	FastWClq		TSM-MWC		WLMC		SCCWalk4L	
	size	time	size	time	size	time	size	time
livejournal-links	**38550** (**38550**)	66.03	**38550*** (**38550**)	71.09	**38550*** (**38550**)	69.23	5106 (2126)	96.6
livemocha	1784 (1784)	10.63	1784* (1784)	**2.24**	1784* (1784)	2.9	1784 (1784)	7.04
lkml-reply	**4763** (**4763**)	0.59	N/A (N/A)	N/A	N/A (N/A)	N/A	202 (191.5)	41.54
loc-gowalla_edges	3053* (**3053**)	0.24	3053* (**3053**)	1	3053* (**3053**)	1.19	3053 (2834.6)	20.51
munmun_digg_reply	686* (686)	**0.02**	686* (686)	2.9	686* (686)	3.08	686 (686)	6.92
munmun_twitter_social	750* (750)	**0.4**	750* (750)	1.12	750* (750)	1.3	750 (750)	1.95
orkut-links	**5290** (**4429.4**)	90.24	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
patentcite	**1438*** (**1438**)	32.47	**1438*** (**1438**)	27.99	**1438*** (**1438**)	28.19	1355 (1006.8)	79.43
petster-carnivore	**119297*** (**119297**)	11.79	119188 (119188)	98.38	119210 (119210)	99.39	5251 (4802.7)	95.82
petster-friendships-cat-uniq	**9364** (9293.8)	40.05	**9364*** (**9364**)	45.79	**9364*** (**9364**)	52.09	1962 (1858.7)	93.33
petster-friendships-dog-uniq	4418 (4411.8)	75.91	4418* (**4418**)	22.58	4418* (**4418**)	21.99	4418 (4404.2)	77.56
roadNet-CA	790* (790)	**1.46**	790* (790)	4.09	790* (790)	4.19	790* (790)	5.88
roadNet-PA	774* (774)	**0.71**	774* (774)	2.09	774* (774)	2.2	774* (774)	2.77
roadNet-TX	772* (772)	**0.9**	772* (772)	2.79	772* (772)	2.58	772* (772)	3.41
slashdot-threads	720* (720)	**0.06**	720* (720)	3.19	720* (720)	3.39	720 (720)	4.87
soc-LiveJournal1	**31814** (**31814**)	58.17	**31814*** (**31814**)	68.18	**31814*** (**31814**)	63.99	5881 (2904.5)	95.94
soc-pokec-relationships	**2993*** (**2993**)	24.59	**2993*** (**2993**)	34.09	**2993*** (**2993**)	34.19	2584 (1992)	75.2
trec-wt10g	9180* (9180)	**1.99**	9180* (9180)	7.99	9180* (9180)	8.49	9180 (9180)	14.78
web-BerkStan	21079* (21079)	**1.25**	21079* (21079)	25.49	21079* (21079)	26.78	21079* (21079)	38.13
web-Google	5044* (5044)	**2.25**	5044* (5044)	4.79	5044* (5044)	4.74	5044* (5044)	6.86
web-NotreDame	19133* (19133)	**0.27**	19133* (19133)	1.19	19133* (19133)	1.4	19133* (19133)	2.47
web-Stanford	7753* (7753)	**0.48**	7753* (7753)	4.69	7753* (7753)	4.75	7753 (7753)	16.65
wiki-Talk	2985 (2985)	**10.37**	2985* (2985)	15.09	2985* (2985)	15.09	2985 (2985)	21.62
wiki_talk_ar	**3292*** (**3292**)	1.74	N/A (N/A)	N/A	N/A (N/A)	N/A	354 (172)	3.65
wiki_talk_de	**4391** (**4390**)	33.56	N/A (N/A)	N/A	N/A (N/A)	N/A	215 (136.5)	8.3
wiki_talk_en	**4452** (**4431.3**)	72.65	N/A (N/A)	N/A	N/A (N/A)	N/A	3371 (2590.8)	99.71
wiki_talk_es	**3831** (**3831**)	2.29	N/A (N/A)	N/A	N/A (N/A)	N/A	594 (188.8)	3.24
wiki_talk_fr	**4141** (**4141**)	4.25	N/A (N/A)	N/A	N/A (N/A)	N/A	179 (115.6)	7.22
wiki_talk_it	**4665** (**4665**)	3.69	N/A (N/A)	N/A	N/A (N/A)	N/A	306 (131.2)	4.5
wiki_talk_nl	**4771** (**4771**)	0.96	N/A (N/A)	N/A	N/A (N/A)	N/A	277 (173.5)	5.34
wiki_talk_pt	**4916** (**4916**)	1.76	N/A (N/A)	N/A	N/A (N/A)	N/A	318 (135.3)	3.65
wiki_talk_ru	**3364** (**3364**)	1.69	N/A (N/A)	N/A	N/A (N/A)	N/A	270 (134)	8.13
wiki_talk_zh	**3897*** (**3897**)	2.02	N/A (N/A)	N/A	N/A (N/A)	N/A	254 (136)	16.39
wikipedia-growth	**4741** (**4741**)	48.08	N/A (N/A)	N/A	N/A (N/A)	N/A	1927 (1784.2)	98.07
wikipedia_link_de	**83708*** (**76772.2**)	89.54	N/A (N/A)	N/A	N/A (N/A)	N/A	1382 (860)	99.46
wikipedia_link_en	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
wikipedia_link_fr	**106234** (**37249.6**)	96.1	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
wikipedia_link_it	**87969*** (**87969**)	71.33	N/A (N/A)	N/A	N/A (N/A)	N/A	2097 (1152.5)	98.28
wikipedia_link_ja	**89015*** (**89015**)	73.69	N/A (N/A)	N/A	N/A (N/A)	N/A	2447 (1769.4)	97.33
wikipedia_link_pl	**82993** (**82993**)	42.08	N/A (N/A)	N/A	N/A (N/A)	N/A	1946 (1782.5)	97.76
wikipedia_link_pt	**103625*** (**103625**)	40.46	N/A (N/A)	N/A	N/A (N/A)	N/A	14706 (4158.2)	86.07
wikipedia_link_ru	**82613** (**82613**)	71.78	N/A (N/A)	N/A	N/A (N/A)	N/A	1820 (1388.62)	98.98
wikisigned-k2	1192* (1192)	**0.21**	1192* (1192)	3.39	1192* (1192)	4.08	1192 (1192)	14.15
wordnet-words	5037* (5037)	**0.21**	5037* (5037)	0.78	5037* (5037)	0.79	5037* (5037)	0.94
youtube-links	1837 (1837)	**4.13**	1837* (1837)	4.49	1837* (1837)	4.59	1837 (1837)	10.37
youtube-u-growth	2321 (2321)	**17.68**	2321* (2321)	27.89	2321* (2321)	27.09	2321 (2321)	55.32
zhishi-baidu-internallink	**3814*** (**3814**)	21	**3814*** (**3814**)	48.09	**3814*** (**3814**)	46.98	1823 (1624.6)	64.88
zhishi-baidu-relatedpages	8938 (8938)	**1.14**	8938* (8938)	49.78	8938* (8938)	49.8	8938 (8938)	37.7
zhishi-hudong-internallink	26110* (26110)	**20.25**	26110* (26110)	33.19	26110* (26110)	32.79	26110 (26110)	81.06
zhishi-hudong-relatedpages	2262* (**2262**)	28.38	2262* (**2262**)	50.19	2262* (**2262**)	49.97	2262 (1710.1)	81.73

Table 4. Experiment results of FastWClq and the state-of-the-art competitors on Repository_normal benchmark.

Instance	FastWClq		TSM-MWC		WLMC		SCCWalk4L	
	size	time	size	time	size	time	size	time
bn-human...1-bg	**19840** (**19395.9**)	41.78	17886 (17886)	89.03	17792 (17792)	83.39	15556 (12576.9)	99.04
bn-human...2-bg	**19981** (**19981**)	15.8	15761 (15761)	74.44	15761 (15761)	94.46	12268 (9862.4)	99.52
ca-coauthors-dblp	33511* (33511)	**4.87**	33511* (33511)	12.92	33511* (33511)	12.94	33511* (33511)	29.74
ca-dblp-2012	11761* (11761)	**0.9**	11761* (11761)	1.28	11761* (11761)	1.25	11761* (11761)	2.26
ca-hollywood-2009	**221751*** (**221751**)	54.05	**221751*** (**221751**)	67.31	**221751*** (**221751**)	68.95	43588 (17032)	96.95
channel-500x100x100-b050	**683*** (**683**)	18.54	**683*** (**683**)	45.89	**683*** (**683**)	46.29	654 (601.7)	85.36
dbpedia-link	**2971** (**2478.1**)	92.99	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
delaunay_n22	612* (**612**)	5.22	612* (**612**)	15.19	612* (**612**)	15.11	612* (486.6)	69.33
delaunay_n23	**610*** (**610**)	11.33	**610*** (**610**)	30.39	**610*** (**610**)	31.06	491 (454.3)	72.54
delaunay_n24	**612*** (**612**)	22.47	N/A (N/A)	N/A	**612*** (**612**)	63.58	448 (413.889)	91.03
friendster	**3886** (**3886**)	84.21	**3886*** (**3886**)	77.84	**3886*** (**3886**)	77.98	794 (493)	98.84
hugebubbles-00020	**434*** (**434**)	52.86	N/A (N/A)	N/A	**434*** (**434**)	61.48	316 (290.3)	82.02
hugetrace-00010	**434*** (**434**)	22.26	N/A (N/A)	N/A	**434*** (**434**)	30.59	**434** (344.8)	60.92
hugetrace-00020	**410*** (**410**)	33.51	N/A (N/A)	N/A	**410*** (**410**)	42.99	317 (301)	75.88
inf-europe_osm	**515*** (**515**)	61.91	N/A (N/A)	N/A	**515*** (**515**)	90.69	263 (201.2)	98.41
inf-germany_osm	**447*** (**447**)	14.48	N/A (N/A)	N/A	**447*** (**447**)	20.19	**447*** (**447**)	44.81
inf-road-usa	**521*** (**521**)	31.59	N/A (N/A)	N/A	**521*** (**521**)	53.99	**521*** (351.9)	90.63
inf-roadNet-CA	507* (507)	**1.72**	507* (507)	4.29	507* (507)	4.29	507 (507)	6.22
inf-roadNet-PA	507* (507)	**0.86**	507* (507)	2.23	507* (507)	2.49	507* (507)	3.25
rec-dating	1305 (1203.1)	37.59	1305* (**1305**)	25.45	1305* (**1305**)	26.37	1305 (**1305**)	27.12
rec-epinions	923 (889.4)	55.6	**947*** (**947**)	71.73	**947*** (**947**)	75.67	**947** (**947**)	45.18
rec-libimseti-dir	1465 (1409.2)	28.52	1465* (**1465**)	27.79	1465* (**1465**)	27.77	1465 (**1465**)	30.27
rgg_n_2_23_s0	**2245*** (**2245**)	25.8	**2245*** (**2245**)	62.89	**2245*** (**2245**)	64.76	1194 (961.75)	99.3
rgg_n_2_24_s0	**2359*** (**2359**)	58.1	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
rt-retweet-crawl	1477* (**1477**)	4.41	1477* (**1477**)	3.74	1477* (**1477**)	3.67	1477 (1411.5)	26.43
sc-ldoor	**2589*** (**2589**)	5.45	**2589*** (**2589**)	18.55	**2589*** (**2589**)	18.68	2554 (2485.8)	67.93
sc-msdoor	**2592*** (**2592**)	2.21	**2592*** (**2592**)	8.28	**2592*** (**2592**)	8.57	2591 (2528.1)	41.77
sc-pwtk	2876* (**2876**)	0.74	2876* (**2876**)	4.79	2876* (**2876**)	5.01	2876 (2831.1)	53.78
sc-rel9	**441*** (**441**)	39.1	**441*** (**441**)	28.74	**441*** (**441**)	28.44	378 (359.1)	78.15
sc-shipsec1	2514* (2514)	**0.18**	2514* (2514)	1.14	2514* (2514)	1.21	2514 (2514)	4.42
sc-shipsec5	2754* (2754)	**0.31**	2754* (2754)	1.93	2754* (2754)	2.09	2754 (2754)	4.37
soc-FourSquare	**3198** (**3198**)	17.54	**3198*** (**3198**)	12.19	**3198*** (**3198**)	11.84	2008 (1955.2)	95.76
soc-LiveMocha	1599 (1599)	7.21	1599* (1599)	**2.36**	1599* (1599)	2.61	1599 (1599)	8.18
soc-buzznet	2944 (2943)	65.3	2944* (**2944**)	6.33	2944* (**2944**)	6.48	2944 (**2944**)	36.85
soc-delicious	2158 (2158)	**0.87**	2158* (2158)	1.25	2158* (2158)	1.25	2158 (2158)	11.82
soc-digg	5036 (**5036**)	12.38	5036* (**5036**)	15.02	5036* (**5036**)	14.48	5036 (4965)	63.64
soc-dogster	4581 (4574.1)	91.23	4581* (**4581**)	10.18	4581* (**4581**)	9.82	4581 (4579.4)	83.73
soc-flickr	5930 (5927.9)	14.97	5930* (**5930**)	4.13	5930* (**5930**)	4.48	5930 (**5930**)	19.77
soc-flickr-und	9856 (9712.5)	33.85	**9929*** (**9929**)	75.5	9317 (9317)	99.99	9222 (7654.8)	99.74
soc-flixster	3200 (3200)	10.87	3200* (3200)	**10.68**	3200* (3200)	10.99	3200 (3200)	35.95
soc-lastfm	1720* (1720)	**4.82**	1720* (1720)	6.03	1720* (1720)	6.03	1720 (1720)	15.41
soc-livejournal	21935* (**21935**)	42.96	21935* (**21935**)	39.17	21935* (**21935**)	38.35	21935* (5755.6)	89.72
soc-livejournal-user-groups	**877** (**828.7**)	76.43	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
soc-ljournal-2008	**39789** (**39789**)	57.33	**39789*** (**39789**)	59.38	**39789*** (**39789**)	60.74	1448 (948.222)	97.51
soc-orkut	**4980** (**3559.8**)	91.36	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
soc-orkut-dir	**4263** (**3596.22**)	88.26	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
soc-pokec	**2925*** (**2925**)	29.58	**2925*** (**2925**)	36.08	**2925*** (**2925**)	36.1	2003 (1613.6)	65.06
soc-sinaweibo	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
soc-twitter-higgs	**7054*** (**7054**)	12.56	**7054*** (**7054**)	18.27	**7054*** (**7054**)	18.09	3892 (3865.3)	55.38
soc-youtube	1645 (1645)	2.87	1645* (1645)	**2.38**	1645* (1645)	2.79	1645 (1645)	8.72
soc-youtube-snap	1796 (1796)	**4.25**	1796* (1796)	4.59	1796* (1796)	4.79	1796 (1796)	12.73
socfb-A-anon	2582* (**2582**)	49.02	2582* (**2582**)	35.47	2582* (**2582**)	34.58	2582 (1847.2)	81.47
socfb-B-anon	**2451*** (**2451**)	24.43	**2451*** (**2451**)	32.51	**2451*** (**2451**)	31.28	2322 (1722.4)	78.65
socfb-uci-uni	**307** (**238.833**)	91.19	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
tech-as-skitter	6627* (6627)	**7.32**	6627* (6627)	22.69	6627* (6627)	22.79	6627 (6627)	76.24
tech-ip	**475** (**456.6**)	61.49	N/A (N/A)	N/A	N/A (N/A)	N/A	276 (237.7)	42.95
twitter_mpi	**4229** (**2922.9**)	90.29	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
web-arabic-2005	10266* (10266)	**0.11**	10266* (10266)	1.42	10266* (10266)	1.49	10266* (10266)	2.17
web-baidu-baike	**2967*** (**2967**)	22.02	**2967*** (**2967**)	25.89	**2967*** (**2967**)	25.59	1509 (1434.3)	66.08
web-it-2004	43259* (43259)	**1.08**	43259* (43259)	5.89	43259* (43259)	5.99	43259* (43259)	17.54
web-uk-2005	49947* (49947)	**1.6**	49947* (49947)	10.39	49947* (49947)	9.77	49947* (49947)	12.98
web-wikipedia-growth	**3107** (**3107**)	64.75	**3107*** (**3107**)	66.91	**3107*** (**3107**)	68.09	1675 (1382.33)	98.06
web-wikipedia2009	3308* (3308)	8.02	3308* (3308)	7.79	3308* (3308)	**7.49**	3308 (3308)	16.68
web-wikipedia_link	**85807** (37968.8)	78.38	N/A (N/A)	N/A	85694 (**85694**)	99.58	N/A (N/A)	N/A
wikipedia_link_en	**4389** (4386.5)	75.8	N/A (N/A)	N/A	**4389*** (**4389**)	70.19	604 (453.7)	91.14

6.4.4 RESULTS ON KONECT_NORMAL

The results on the KONECT_normal benchmark are given in Table 6. Once again, the results of FastWClq are significantly better than those of the three competitors for most instances. FastWClq finds cliques of the same quality or better for all graphs except two: instance wikipedia_link_en that no algorithm is able to solve in the time limit, and instance flickr-links for which SCCWalk4L finds the better solution. FastWClq again computes the most number of optimal solutions, in this case 56, although it only proves 44 of these solutions to be optimal. TSM-MWC and WLMC find fewer optimal solutions, at 48 each; however, all of these solutions are proved optimal. FastWClq also significantly outperforms the exact algorithms in terms of being able to produce *any* solution in the 100-second time limit: whereas FastWClq only fails to give a solution to one graph, TSM-MWC, WLMC fail on 29 instances each. SCCWalk4L fails to solve only 3 instances; however, its solution quality is worse than FastWClq on 38 instances.

Table 6. Experiment results of FastWClq and the state-of-the-art competitors on KONECT_normal benchmark.

Instance	FastWClq		TSM-MWC		WLMC		SCCWalk4L	
	size	time	size	time	size	time	size	time
actor-collaboration	29128* (29128)	**9.1**	29128* (29128)	27.39	29128* (29128)	27.69	29128 (29128)	80.24
amazon0601	1387* (1387)	**1.54**	1387* (1387)	2.88	1387* (1387)	3.09	1387* (1387)	4.07
as-skitter	6540* (6540)	**6.81**	6540* (6540)	22.71	6540* (6540)	22.27	6540 (6540)	72.63
citeseer	1292* (**1292**)	1.32	1292* (**1292**)	2.45	1292* (**1292**)	2.46	1292 (1267.1)	20.87
com-amazon	813* (813)	**0.52**	813* (813)	1.18	813* (813)	1.27	813* (813)	1.36
com-dblp	11025* (11025)	**0.26**	11025* (11025)	1.27	11025* (11025)	1.35	11025* (11025)	1.64
com-youtube	1672 (1672)	4.21	1672* (1672)	**4.19**	1672* (1672)	4.7	1672 (1672)	11.73
dbpedia-all	**1670*** (**1670**)	16.29	N/A (N/A)	N/A	N/A (N/A)	N/A	1116 (767.6)	92.63
dbpedia-link	**784** (**563.667**)	95.24	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
digg-friends	**3420** (**3420**)	25.4	N/A (N/A)	N/A	N/A (N/A)	N/A	323 (233.5)	9.64
douban	1073* (1073)	**0.06**	1073* (1073)	0.52	1073* (1073)	0.68	1073 (1073)	0.61
elec	1895* (1895)	**0.15**	1895* (1895)	2.68	1895* (1895)	2.79	1895 (1895)	8.89
email-EuAll	1637* (1637)	**0.12**	1637* (1637)	0.69	1637* (1637)	0.96	1637 (1637)	0.98
enron	3389* (3389)	**0.14**	3389* (3389)	18.55	3389* (3389)	18.64	3389 (3389)	19.41
epinions	10002 (10002)	**0.24**	10002* (10002)	0.96	10002* (10002)	1.05	10002 (10002)	4.88
facebook-wosn-links	2993* (2993)	**0.48**	2993* (2993)	42.29	2993* (2993)	42.29	2993 (2993)	27.45
facebook-wosn-wall	**1088*** (**1088**)	0.55	N/A (N/A)	N/A	N/A (N/A)	N/A	428 (313.7)	24.55
flickr-growth	**10094** (**9970.9**)	37.88	8797 (8797)	42.65	8797 (8797)	40.93	8749 (7097.2)	98.49
flickr-links	9760 (**9577.7**)	26.2	8699 (8699)	98.45	8272 (8272)	86.9	**9822** (8619)	99.3
flickrEdges	57102* (57102)	**1.27**	57102* (57102)	2.45	57102* (57102)	2.68	57102* (57102)	13.48
flixster	3255 (3255)	10.48	3255* (3255)	**10.18**	3255* (3255)	10.84	3255 (3255)	23.51
hyves	1798* (1798)	**2.62**	1798* (1798)	4.82	1798* (1798)	4.58	1798 (1798)	6.68
lasagne-yahoo	469* (**469**)	16.01	469* (**469**)	31.87	469* (**469**)	31.85	469 (457.6)	46.21
libimseti	2028 (1910.3)	56.71	2028* (**2028**)	94.61	2028* (**2028**)	94.34	2028 (**2028**)	27.41
link-dynamic-dewiki	**9961** (**9748**)	75.16	N/A (N/A)	N/A	N/A (N/A)	N/A	285 (239.714)	94.32
link-dynamic-frwiki	**7832** (**7816**)	81.91	N/A (N/A)	N/A	N/A (N/A)	N/A	312 (224.2)	74.16
link-dynamic-itwiki	**4808** (**4808**)	28.04	N/A (N/A)	N/A	N/A (N/A)	N/A	438 (275.7)	42.23
link-dynamic-nlwiki	**5059*** (**5059**)	15.42	N/A (N/A)	N/A	N/A (N/A)	N/A	234 (202.7)	29.23
link-dynamic-plwiki	**5183*** (**5183**)	33.43	N/A (N/A)	N/A	N/A (N/A)	N/A	229 (207.1)	32.42
link-dynamic-simplewiki	2697* (2697)	**0.43**	2697* (2697)	23.38	2697* (2697)	23.35	2697 (2697)	18.06
livejournal-links	**36124** (**36124**)	66.94	**36124*** (**36124**)	67.58	**36124*** (**36124**)	64.08	2150 (1422.1)	95.43
livemocha	1736 (1736)	7.41	1736* (1736)	**2.59**	1736* (1736)	2.73	1736 (1736)	7.04
lkml-reply	**4992** (**4992**)	0.78	N/A (N/A)	N/A	N/A (N/A)	N/A	325 (290.9)	42.32
loc-gowalla_edges	2767* (**2767**)	0.33	2767* (**2767**)	1.26	2767* (**2767**)	1.46	2767 (2662.8)	39.42
munmun_digg_reply	493* (493)	**0.06**	493* (493)	2.88	493* (493)	3	493 (493)	9.73
munmun_twitter_social	617* (617)	**0.41**	617* (617)	1.19	617* (617)	1.29	617 (617)	2.78

(Continued on next page)

Table 6. (Continued) Experiment results of FastWClq and the state-of-the-art competitors on KONECT_normal benchmark.

Instance	FastWClq		TSM-MWC		WLMC		SCCWalk4L	
	size	time	size	time	size	time	size	time
orkut-links	**5246** (**4109.6**)	92.27	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
patentcite	**1150*** (**1150**)	28.84	**1150*** (**1150**)	28.28	**1150*** (**1150**)	29.39	1004 (809.1)	71.91
petster-carnivore	**115803*** (**115803**)	11.49	**115803*** (**115803**)	53.78	**115803*** (**115803**)	53.88	6159 (5547.7)	99.38
petster-friendships-cat-uniq	**8453** (8415.9)	22.36	**8453*** (**8453**)	45.78	**8453*** (**8453**)	44.79	2435 (2325.8)	95.58
petster-friendships-dog-uniq	4622 (4622)	65.45	4622* (4622)	22.49	4622* (4622)	**22.09**	4622 (4622)	52.4
roadNet-CA	503* (503)	**1.52**	503* (503)	4.19	503* (503)	4.29	503* (503)	6.23
roadNet-PA	480* (480)	**0.77**	480* (480)	2.09	480* (480)	2.29	480* (480)	2.91
roadNet-TX	484* (484)	**0.89**	484* (484)	2.59	484* (484)	2.69	484 (484)	3.59
slashdot-threads	740* (740)	**0.08**	740* (740)	3.19	740* (740)	3.5	740 (740)	8.31
soc-LiveJournal1	**32219** (**32219**)	56.21	**32219*** (**32219**)	58.65	**32219*** (**32219**)	57.59	4426 (2005.4)	94.61
soc-pokec-relationships	**2940*** (**2940**)	24.95	**2940*** (**2940**)	33.69	**2940*** (**2940**)	34.49	2197 (1664.8)	66.32
trec-wt10g	9009* (9009)	**2.07**	9009* (9009)	7.89	9009* (9009)	8.59	9009 (9009)	16.11
web-BerkStan	20098* (20098)	**1.31**	20098* (20098)	25.68	20098* (20098)	25.49	20098 (20098)	41.84
web-Google	4110* (4110)	**2.33**	4110* (4110)	4.79	4110* (4110)	4.97	4110 (4110)	7.09
web-NotreDame	15838* (15838)	**0.17**	15838* (15838)	1.29	15838* (15838)	1.39	15838* (15838)	2.65
web-Stanford	6280* (6280)	**0.48**	6280* (6280)	4.59	6280* (6280)	4.67	6280 (6280)	20.59
wiki-Talk	2714 (2714)	16.98	2714* (2714)	14.79	2714* (2714)	**14.29**	2714 (2714)	23.78
wiki_talk_ar	**3338*** (**3338**)	2.12	N/A (N/A)	N/A	N/A (N/A)	N/A	336 (229.5)	10.65
wiki_talk_de	**5040** (**5040**)	8	N/A (N/A)	N/A	N/A (N/A)	N/A	244 (214.9)	8.5
wiki_talk_en	**4061** (**4055.8**)	68.7	N/A (N/A)	N/A	N/A (N/A)	N/A	1494 (1375.8)	97.68
wiki_talk_es	**3675** (**3675**)	4.21	N/A (N/A)	N/A	N/A (N/A)	N/A	388 (249.4)	6.57
wiki_talk_fr	**3879** (**3879**)	5.76	N/A (N/A)	N/A	N/A (N/A)	N/A	262 (211)	7.2
wiki_talk_it	**4784** (**4784**)	3.81	N/A (N/A)	N/A	N/A (N/A)	N/A	288 (224.5)	13.71
wiki_talk_nl	**3891** (**3891**)	2.62	N/A (N/A)	N/A	N/A (N/A)	N/A	341 (253.6)	6.62
wiki_talk_pt	**4498** (**4498**)	2.28	N/A (N/A)	N/A	N/A (N/A)	N/A	333 (222.9)	13.03
wiki_talk_ru	**3359** (**3359**)	2.07	N/A (N/A)	N/A	N/A (N/A)	N/A	341 (232.8)	2.99
wiki_talk_zh	**3062** (**3062**)	2.42	N/A (N/A)	N/A	N/A (N/A)	N/A	333 (220.7)	6.93
wikipedia-growth	**3286** (**3286**)	46.35	N/A (N/A)	N/A	N/A (N/A)	N/A	1638 (1478.9)	97.73
wikipedia_link_de	**83398*** (**83398**)	90.27	N/A (N/A)	N/A	N/A (N/A)	N/A	1276 (805.5)	99.62
wikipedia_link_en	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
wikipedia_link_fr	**105526** (**40333.1**)	95.66	N/A (N/A)	N/A	N/A (N/A)	N/A	N/A (N/A)	N/A
wikipedia_link_it	**84676*** (**84676**)	74.06	N/A (N/A)	N/A	N/A (N/A)	N/A	1710 (862.333)	99.1
wikipedia_link_ja	**84792*** (**84792**)	76.13	N/A (N/A)	N/A	N/A (N/A)	N/A	2561 (1687.5)	98.5
wikipedia_link_pl	**82737** (**82737**)	40.42	N/A (N/A)	N/A	N/A (N/A)	N/A	1985 (1585.9)	94.31
wikipedia_link_pt	**104461*** (**104461**)	40.61	N/A (N/A)	N/A	N/A (N/A)	N/A	13939 (3933.1)	88.49
wikipedia_link_ru	**80185** (**80185**)	73.62	N/A (N/A)	N/A	N/A (N/A)	N/A	1771 (872.125)	98.65
wikisigned-k2	1192 (1192)	**1.01**	1192* (1192)	3.99	1192* (1192)	4.09	1192 (1192)	17.92
wordnet-words	3274* (3274)	**0.2**	3274* (3274)	0.69	3274* (3274)	0.79	3274* (3274)	1.09
youtube-links	1648 (1648)	**2.86**	1648* (1648)	4.58	1648* (1648)	4.7	1648 (1648)	12.43
youtube-u-growth	**2015** (**2015**)	11.06	**2015*** (**2015**)	28.79	**2015*** (**2015**)	28.19	1999 (1999)	51.05
zhishi-baidu-internallink	**3166*** (**3166**)	21.65	**3166*** (**3166**)	49.58	**3166*** (**3166**)	47.39	1476 (1464)	66.6
zhishi-baidu-relatedpages	9154* (**9154**)	1.33	9154* (**9154**)	49.89	9154* (**9154**)	49.59	9154 (5904.3)	65.04
zhishi-hudong-internallink	26627* (**26627**)	21.2	26627* (**26627**)	33.19	26627* (**26627**)	31.79	26627* (24272.3)	82.93
zhishi-hudong-relatedpages	1566* (**1566**)	29.66	1566* (**1566**)	48.49	1566* (**1566**)	50.98	1566 (1121.5)	70.15

6.5 The Effectiveness of Three Upper Bound Functions

To determine the effectiveness of three upper bound functions (i.e., UB_0, UB_1 and UB_2), we compare FastWClq with its five alternative versions, each of which uses a different subset of the upper bound functions, given as follows: (1) FastWClq0 (only UB_0); (2) FastWClq1 (only UB_1); (3) FastWClq2 (only UB_2); (4) FastWClq01 (only UB_0 and UB_1); and (5) FastWClq02 (only UB_0 and

Table 7. Comparative results of solving ability of FastWClq and its variants with each upper bound and each combination of two upper bounds on all benchmarks

Benchmark	#inst.	FastWClq		FastWClq02		FastWClq01		FastWClq2		FastWClq1		FastWClq0	
		#win	time	#win	time	#win	time	#win	time	#win	time	#win	time
Repository_200	65	47	32.53	46	32.97	47	33.51	48	29.9	49	29.72	48	32.25
Repository_normal	65	49	32.34	49	33.32	48	33.24	48	29.2	51	29.42	48	32.54
KONECT_200	80	71	22.34	70	23.21	68	22.46	54	20.93	54	20.65	68	22.62
KONECT_normal	80	70	21.27	70	21.84	69	21.63	51	20.59	58	20.39	70	21.34

Table 8. Comparative results of proving ability of FastWClq and its variants with each upper bound and each combination of two upper bounds on all benchmarks

Benchmark	#inst.	FastWClq	FastWClq02	FastWClq01	FastWClq2	FastWClq1	FastWClq0
		#prov	#prov	#prov	#prov	#prov	#prov
Repository_200	65	36	36	22	35	22	12
Repository_normal	65	36	36	19	36	19	14
KONECT_200	80	42	41	17	34	14	16
KONECT_normal	80	44	44	14	38	11	13

UB_2). For each algorithm, we report the number of instances where it performs best among all algorithms on the metric of maximum and average weight of cliques, denoted by '#win', and the number of instances where the algorithm proves the optimality of its solution, denoted as '#prov'.

Table 7 summarizes the comparative results in terms of solution quality and run time for FastWClq and the five alternative versions on all four benchmarks. Overall, FastWClq gives the best performance on these benchmarks, thus showing that all upper bounds contribute to its performance.

Although FastWClq is slightly outperformed in terms of quality by other variations on some individual benchmarks, FastWClq outperforms these other variants on all benchmarks in terms of proving optimality. In Table 8 we present the number of solutions for which FastWClq and its five versions prove optimality. Observed from Table 8, FastWClq shows the best proving ability, while the version using UB_0 and UB_2 has very close proving ability with FastWClq. This indicates that with UB_0 and UB_2 , the contribution of UB_1 on the proving ability can be omitted. This can be well understood as UB_2 is an enhanced bound built on the basis of UB_1.

To further analyze the upper bound functions of FastWClq, we report the percent of remaining vertices after the last successful call of graph reduction and the time at which the last successful reduction procedure finishes (Table 9). The detailed information of all instances is presented in Figure 2. Observed from the results, the upper bound functions can make a significant contribution for FastWClq in a short time.

6.6 Summary of Experiments

Table 10 summarizes the experimental results of FastWClq and three competitors on the four benchmarks. To further provide a better understanding of the performance of the algorithms, besides the time limit of 100 seconds, we also test the algorithms with the time limit of 300 seconds and 600 seconds. For these four benchmarks of large graphs, FastWClq has the top performance in terms of solution quality, followed by TSM-MWC, WLMC and SCCWalk4L (in this order).

Table 9. Information on the effectiveness of graph reduction in FastWClq. (RV_{avg} is the average number of remaining vertices, $RV_{avg}\%$ is the percent of remaining vertices among all the vertices, and t_{avg} is the time at which the last successful reduction procedure finishes.)

Benchmark	FastWClq		
	RV_{avg}	$RV_{avg}\%$	t_{avg}
Repository_200	2025185.44	10.52%	40.12
Repository_normal	2031723.03	11%	41.13
KONECT_200	413092.01	4.65%	29.68
KONECT_normal	412874.82	4.58%	29.19

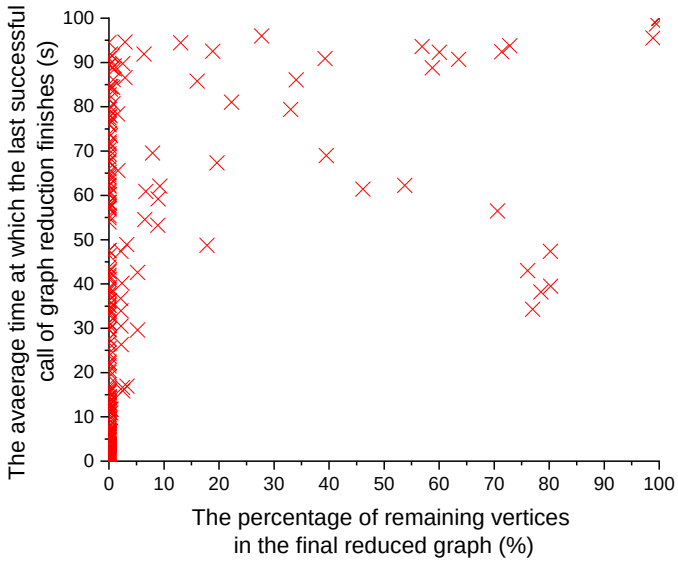

Figure 2. A plot comparing the time of the last successful graph reduction and the percentage of vertices remaining in the graph.

The density of these large real-world graphs ranges from 0.000004% to 0.139%. For 75 large real-world graphs whose density is less than 0.001%, FastWClq steadily obtains the same or better solution values, compared to all competitors. For the remaining graphs whose density, FastWClq is also competitive, and gives the best solution for those instances with only 7 exceptions.

We now turn to compare the ablity of proving the optimality. Compared with FastWClq, exact algorithms TSM-MWC and WLMC prove the optimality of more solutions, which is not surprising. Nevertheless, we note that our semi-exact algorithm FastWClq also shows competitive proving ability. It proves the optimality of its solutions for about half of the tested graphs including graphs with millions of vertices, and the average time for doing this is less than 22 seconds.

7. Conclusions and Future Work

This paper presented a novel semi-exact method for the maximum weight clique problem (MWCP), which aims to solve large graphs within a short time limit. The method interleaves clique finding

Table 10. Experimental results of FastWClq and the competitors on all benchmarks with different time limits.

Benchmark	#inst.	Time	FastWClq			TSM-MWC			WLMC			SCCWalk4L	
		Limit	#win	time	#prov	#win	time	#prov	#win	time	#prov	#win	time
Repository_200	65	100s	55	32.53	36	45	26.19	45	53	31.32	52	22	54.7
Repository_normal	65		54	32.34	36	45	25.28	45	52	31.6	52	23	54.81
KONECT_200	80		75	22.34	42	48	22	48	47	23.09	47	37	39.34
KONECT_normal	80		75	21.27	44	48	20.9	48	48	20.58	48	34	39.67
Repository_200	65	300s	55	50.26	43	52	46.66	51	62	53.38	62	37	121.37
Repository_normal	65		54	45.35	45	51	43.58	51	62	48.77	62	33	131.74
KONECT_200	80		76	31.86	52	59	61.8	58	58	62.19	58	47	86.47
KONECT_normal	80		76	30.31	52	61	58.6	61	59	57.83	59	50	89.68
Repository_200	65	600s	59	67.88	43	52	46.66	51	62	55.49	62	38	182.67
Repository_normal	65		58	81.34	45	52	49.29	52	62	48.77	62	36	210.17
KONECT_200	80		77	39.52	52	62	70	62	61	75.23	61	57	140.92
KONECT_normal	80		77	50	52	62	63.07	62	61	80.43	61	55	140.7

and graph reduction. Several ideas were proposed to improve the clique finding algorithm and the graph reduction algorithm. The resulting algorithm is called FastWClq. Experiments on large real-world graphs show that FastWClq finds better solutions than state-of-the-art algorithms while using less time on most instances. Also, FastWClq proves the optimality of its solutions for about half of the tested graphs including graphs with millions of vertices, and the average time for doing this is less than 22 seconds.

As shown in this work, semi-exact algorithms take the advantage of solving and proving, and seem a promising direction for solving large combinatorial optimization problems. A significant direction for future work is to apply the semi-exact method to other combinatorial optimization problems problems.

Acknowledgement

This work is an improved and extended version of a conference paper (Cai & Lin, 2016). This work was supported by Beijing Academy of Artificial Intelligence (BAAI), Youth Innovation Promotion Association, Chinese Academy of Sciences [No. 2017150], NSFC Grant 61806050, and Jilin Science and Technology Association QT202005.

References

Balasundaram, B., & Butenko, S. (2006). Graph domination, coloring and cliques in telecommunications. In *Handbook of Optimization in Telecommunications*, pp. 865–890.

Ballard, D., & Brown, C. (1982). *Computer Vision*. New Jersey: Prentice Hall.

Batagelj, V., & Zaveršnik, M. (2003). An $O(m)$ algorithm for cores decomposition of networks. *CoRR*, *cs.DS/0310049*.

Benlic, U., & Hao, J.-K. (2013). Breakout local search for maximum clique problems. *Computers & Operations Research*, *40*(1), 192–206.

Busygin, S. (2006). A new trust region technique for the maximum weight clique problem. *Discrete Applied Mathematics*, *154*(15), 2080–2096.

Cai, S. (2015). Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In *Proceedings of IJCAI 2015*, pp. 747–753.

Cai, S., Li, Y., Hou, W., & Wang, H. (2019). Towards faster local search for minimum weight vertex cover on massive graphs. *Information Sciences*, *471*, 64–79.

Cai, S., & Lin, J. (2016). Fast solving maximum weight clique problem in massive graphs. In *Proceedings of IJCAI 2016*, pp. 568–574.

Cai, S., Luo, C., Zhang, X., & Zhang, J. (2021). Improving local search for structured sat formulas via unit propagation based construct and cut initialization. In *Proceedings of CP 2021*.

Demange, M., de Werra, D., Monnot, J., & Paschos, V. T. (2002). Weighted node coloring: when stable sets are expensive. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pp. 114–125. Springer.

Fan, Y., Li, N., Li, C., Ma, Z., Latecki, L. J., & Su, K. (2017). Restart and random walk in local search for maximum vertex weight cliques with evaluations in clustering aggregation. In *Proceedings of IJCAI 2017*, pp. 622–630.

Fang, Z., Li, C.-M., Qiao, K., Feng, X., & Xu, K. (2014). Solving maximum weight clique using maximum satisfiability reasoning. In *Proceedings of ECAI 2014*, pp. 303–308.

Fellows, M. R., & Downey, R. (1998). *Parameterized Complexity*. Springer.

Gomez Ravetti, M., & Moscato, P. (2008). Identification of a 5-protein biomarker molecular signature for predicting Alzheimer's disease. *PloS one*, *3*(9), e3111.

Guturu, P., & Dantu, R. (2008). An impatient evolutionary algorithm with probabilistic tabu search for unified solution of some NP-hard problems in graph and set theory via clique finding. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, *38*(3), 645–666.

Hsu, H., & Chang, G. J. (2016). Max-coloring of vertex-weighted graphs. *Graphs and Combinatorics*, *32*(1), 191–198.

Jiang, H., Li, C., Liu, Y., & Manyà, F. (2018). A two-stage MaxSAT reasoning approach for the maximum weight clique problem. In *Proceedings of AAAI 2018*, pp. 1338–1346.

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

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations*, pp. 85–103. Springer.

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

Kumlander, D. (2004). Fast maximum clique algorithms for large graphs. In *Proceedings of the fourth conference on engineering computational technology*, pp. 202–208.

Kunegis, J. (2013). Konect: the koblenz network collection. In *Proceedings of WWW 2013*, pp. 1343–1350.

Lamm, S., Schulz, C., Strash, D., Williger, R., & Zhang, H. (2019). Exactly solving the maximum weight independent set problem on large real-world graphs. In *Proceedings of ALENEX 2019*, pp. 144–158.

Li, C.-M., Fang, Z., & Xu, K. (2013). Combining MaxSAT reasoning and incremental upper bound for the maximum clique problem. In *Proceedings of ICTAI 2013*, pp. 939–946.

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

Li, C., Liu, Y., Jiang, H., Manyà, F., & Li, Y. (2018). A new upper bound for the maximum weight clique problem. *European Journal of Operational Research*, *270*(1), 66–77.

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

Li, R., Hu, S., Cai, S., Gao, J., Wang, Y., & Yin, M. (2020). Numwvc: A novel local search for minimum weighted vertex cover problem. *Journal of the Operational Research Society*, *71*(9), 1498–1509.

Massaro, A., Pelillo, M., & Bomze, I. M. (2002). A complementary pivoting approach to the maximum weight clique problem. *SIAM Journal on Optimization*, *12*(4), 928–948.

McCreesh, C., Prosser, P., Simpson, K., & Trimble, J. (2017). On maximum weight clique algorithms, and how they are evaluated. In *Proceedings of CP 2017*, pp. 206–225.

Östergård, P. R. J. (1999). A new algorithm for the maximum-weight clique problem. *Electronic Notes in Discrete Mathematics*, *3*, 153–156.

Pullan, W. (2006). Phased local search for the maximum clique problem. *Journal of Combinatorial Optimization*, *12*(3), 303–323.

Pullan, W. (2008). Approximating the maximum vertex/edge weighted clique using local search. *Journal of Heuristics*, *14*(2), 117–134.

Pullan, W. (2009). Optimisation of unweighted/weighted maximum independent sets and minimum vertex covers. *Discrete Optimization*, *6*(2), 214–219.

Pullan, W., & Hoos, H. H. (2006). Dynamic local search for the maximum clique problem. *Journal of Artificial Intelligence Research*, *25*, 159–185.

Rossi, R. A., & Ahmed, N. K. (2015). The network data repository with interactive graph analytics and visualization. In *Proceedings of AAAI 2015*, pp. 4292–4293.

Rossi, R. A., Gleich, D. F., Gebremedhin, A. H., & Patwary, M. (2014). Fast maximum clique algorithms for large graphs. In *Proceedings of WWW 2014*, pp. 365–366.

San Segundo, P., Furini, F., & Artieda, J. (2019). A new branch-and-bound algorithm for the maximum weighted clique problem. *Computers & Operations Research*, *110*, 18–33.

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

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

Seidman, S. (1983). Network structure and minimum degree. *Social Networks*, *5*(3), 269–287.

Singh, A., & Gupta, A. K. (2006a). A hybrid evolutionary approach to maximum weight clique problem. *International Journal of Computational Intelligence Research*, *2*(4), 349–355.

Singh, A., & Gupta, A. K. (2006b). A hybrid heuristic for the maximum clique problem. *Journal of Heuristics*, *12*(1-2), 5–22.

Tomita, E., & Kameda, T. (2007). An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global optimization*, *37*(1), 95–111.

Tomita, E., & Seki, T. (2003). An efficient branch-and-bound algorithm for finding a maximum clique. In *Discrete mathematics and theoretical computer science*, pp. 278–289.

Tomita, E., Sutani, Y., Higashi, T., Takahashi, S., & Wakatsuki, M. (2010). A simple and faster branch-and-bound algorithm for finding a maximum clique. In *Proceedings of WALCOM 2010*, pp. 191–203.

Verma, A., Buchanan, A., & Butenko, S. (2015). Solving the maximum clique and vertex coloring problems on very large sparse networks. *INFORMS Journal on Computing*, *27*(1), 164–177.

Wang, Y., Cai, S., Chen, J., & Yin, M. (2020). SCCWalk: An efficient local search algorithm and its improvements for maximum weight clique problem. *Artificial Intelligence*, *280*, 103230.

Wang, Y., Cai, S., & Yin, M. (2016). Two efficient local search algorithms for maximum weight clique problem. In *Proceedings of AAAI 2016*, pp. 805–811.

Wu, Q., Hao, J.-K., & Glover, F. (2012). Multi-neighborhood tabu search for the maximum weight clique problem. *Annals of Operations Research*, *196*(1), 611–634.

Zhou, Y., Hao, J., & Goëffon, A. (2017). PUSH: A generalized operator for the maximum vertex weight clique problem. *European Journal of Operational Research*, *257*(1), 41–54.

Zuckerman, D. (2007). Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, *3*(1), 103–128.