NuQClq: An Effective Local Search Algorithm for Maximum Quasi-Clique Problem

Jiejiang Chen[1], Shaowei Cai[2,4], Shiwei Pan[1], Yiyuan Wang[1,3*], Qingwei Lin[5], Mengyu Zhao[1], Minghao Yin[1,3*]

[1]School of Computer Science and Information Technology, Northeast Normal University, China
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China
[3]Key Laboratory of Applied Statistics of MOE, Northeast Normal University, Changchun, China
[4]School of Computer Science and Technology, University of Chinese Academy of Sciences, China
[5]Microsoft Research, China
{chenjj016, pansw779, zhaomy588, ymh}@nenu.edu.cn, caisw@ios.ac.cn, yiyuanwangjlu@126.com, qlin@microsoft.com

Abstract

The maximum quasi-clique problem (MQCP) is an important extension of maximum clique problem with wide applications. Recent heuristic MQCP algorithms can hardly solve large and hard graphs effectively. This paper develops an efficient local search algorithm named NuQClq for the MQCP, which has two main ideas. First, we propose a novel vertex selection strategy, which utilizes cumulative saturation information to be a selection criterion when the candidate vertices have equal values on the primary scoring function. Second, a variant of configuration checking named BoundedCC is designed by setting an upper bound for the threshold of forbidding strength. When the threshold value of vertex exceeds the upper bound, we reset its threshold value to increase the diversity of search process. Experiments on a broad range of classic benchmarks and sparse instances show that NuQClq significantly outperforms the state-of-the-art MQCP algorithms for most instances.

Introduction

Given a graph, a clique is a subset of vertices in which each pair of vertices are adjacent. Clique is an important concept in graph theory, and the maximum clique problem (MCP) is a classic graph-theoretic problem. The clique model is well known for its applications in social network analysis (Luce and Perry 1949; Wasserman, Faust et al. 1994), and there are also interests in its applications to cluster detection in different fields such as bioinformatics, chemoinformatics, financial networks, and telecommunications (Wu and Hao 2015). Nevertheless, the concept of clique is too strict for many real-world applications, as it requires each pair of vertices in the clique to be adjacent. In fact, most information arisen from real world is incomplete or erroneous, which motivates the development of clique relaxations. There have been various clique relaxations: k-clubs (Pajouh and Balasundaram 2012), k-defective cliques (Trukhanov et al. 2013), k-plexes (Gao et al. 2018), and γ-quasi-cliques (Zhou, Benlic, and Wu 2020). In this paper, we focus on the maximum quasi-clique problem (MQCP), which has been used in various real-world domains (On et al. 2006; Tsourakakis et al. 2013; Lee and Lakshmanan 2016). For example, Zheng et al. (2019) proposed a new method for author set identification, i.e., matching the information of reviewers and papers by encoding this problem to the MQCP.

Given a graph $G = (V, E)$ and a fixed constant $\gamma \in (0,1]$, a γ-quasi-clique is a subset S of V such that the edge density of the subgraph induced by S is at least γ. The MQCP aims to identify the γ-quasi-clique with the maximum size in a graph. It is easy to see that MCP is a special case of MQCP. The MQCP is NP-hard for every fixed constant $\gamma \in (0,1]$ (Pattillo et al. 2013).

There are mainly two types of algorithms for the MQCP, i.e., exact algorithms and heuristic algorithms. Recently, many exact algorithms (Pattillo et al. 2013; Pajouh, Miao, and Balasundaram 2014; Veremyev et al. 2016; Pastukhov et al. 2018; Ribeiro and Riveaux 2019; Marinelli, Pizzuti, and Rossi 2020; Miao and Balasundaram 2020) have been proposed for solving the MQCP. Although exact algorithms can guarantee the optimality of their solutions, they may fail to solve large-size instances.

To deal with hard instances of large scale, lots of fast heuristic algorithms have been designed for solving the MQCP, which can obtain a good approximate solution within reasonable time. An early heuristic algorithm for solving the MQCP was proposed in (Abello, Resende, and Sudarsky 2002), which relied on efficient semi-external memory algorithms and greedy randomized adaptive search procedures. Brunato et al. (2007) introduced two high-performance stochastic local search algorithms according to new data structures and some basic operators. Khosraviani and Sharifi (2011) developed an MQCP algorithm based on MapReduce programming model. Tsourakakis et al.(2013) designed two efficient algorithms for the MQCP, including a greedy algorithm and a heuristic based on the local-search paradigm. Oliveira et al. (2013) presented a restart iterative greedy algorithm named RIG*. Afterwards, some different versions of biased random-key genetic algorithm (BRKGA) for the MQCP were proposed (Pinto et al. 2015, 2018, 2019). Among these versions, BRKGA-LSQClique

*Corresponding author

(2019) had the best performance. Djeddi et al. (2019) used an extension of adaptive multistart tabu search to approximate the MQCP solution, resulting in the TSQC algorithm. Very recently, Zhou et. al. (2020) presented an opposition-based memetic algorithm named OBMA, relied on three novel ideas. According to the literature, because the above three heuristic algorithms are proposed in parallel, the current best heuristic algorithms for the MQCP are BRKGA-LSQClique,TSQC and OBMA.

In this paper, we develop an efficient local search algorithm named NuQClq, which achieves the best results on almost all benchmarks used in the literature. There are two main novel ideas in our algorithm.

The first idea aims to address the issue about tie-breaking in the primary scoring function. Although tie-breaking mechanism seems a relatively minor concern at first sight, our experiments show that the common scoring function based on vertex degree usually results in more than one vertices with the best score, making tie breaking an important factor of the algorithm. To break ties, we introduce a novel function, which indeed, can be seen as the secondary scoring function. This function is based on a key concept named cumulative saturation, which measures the sum of primary scoring value of a vertex during the search history. The idea is that the larger cumulative saturation value a vertex owns, the higher probability it would be selected. In some sense, this cumulative saturation value can be seen as an "integration" of the primary scoring function over the steps since the last time it changed its state.

The second strategy is a new configuration checking (CC) strategy named BoundedCC which considers the characteristics of the MQCP. CC proposed by Cai (2011) was used to overcome the cycling issue of local search. Different variants of CC have been used in many combinational optimization problems (Luo et al. 2015; Wang et al. 2018; Li et al. 2018; Wang et al. 2020; Chen et al. 2020). The BoundedCC strategy distinguishes itself from previous CC variants by introducing an upper bound for the threshold which represents the forbidding strength of CC. If the threshold value of vertex exceeds the upper bound, then its threshold is reset in order to increase the diversity of search process.

Extensive experiments are carried out to evaluate NuQClq on the benchmarks used in the literature. Compared with three state-of-the-art heuristic algorithms, NuQClq obtains the best results for almost all benchmarks. Besides, our experimental analyses show that the proposed strategies play crucial roles in the outstanding performance of NuQClq.

Preliminaries

An undirected graph $G = (V, E)$ consists of a vertex set $V = \{v_1, \ldots, v_n\}$ and an edge set $E = \{e_1, \ldots, e_m\}$. Each edge $e = (u, v)$ is a 2-element subset of V. Two vertices are neighbors if they belong to a same edge. For a vertex v, the set of edges incident to v is denoted as $E(v)$. The density of graph G is given by $dens(G) = |E|/\binom{|V|}{2}$. For a vertex v, its neighborhood is $N_G(v) = \{u \mid u \in V, (u, v) \in E\}$, and the degree of vertex v is denoted as $d_G(v) = |N_G(v)|$. Δ_G is the maximum number of $d_G(v)$ among V. The closed neighborhood is $N_G[v] = N_G(v) \cup \{v\}$. For a vertex set $S \subseteq V$, we use $N_G(S) = \bigcup_{v \in S} N_G(v) \setminus S$ and $N_G[S] = \bigcup_{v \in S} N_G[v]$ to denote the neighborhood and the closed neighborhood of S, respectively. The induced subgraph $G[S] = (V_S, E_S)$ is a subgraph of G whose vertex set is S and whose edge set includes all the edges in E_S that have both endpoints in S.

Given a graph $G = (V, E)$ and a fixed constant $\gamma \in$ (0,1], a γ-quasi-clique is any subset $S \subseteq V$ such that $dens(G[S]) \geq \gamma$. A γ-quasi-clique S is maximal if there is no other γ-quasi-clique S' that strictly contains S. The maximum quasi-clique problem (MQCP) is to find a γ-quasi-clique S with the most vertices.

Cumulative Saturation Heuristic

We propose a vertex selection function based on the concept of cumulative saturation. Before presenting the heuristic, we introduce the previous scoring functions.

Previous Scoring Functions

Local search algorithms for MQCP maintain a current candidate solution denoted as $S \subseteq V$. The adjacency $d_S(v)$ of a vertex $v \in V$ with S is the number of vertices in S that are connected to v, i.e.,

$$d_S(v) = |\{u \mid u \in S, (u, v) \in E\}|$$

In previous heuristic search algorithms for MQCP, the scoring functions are mainly based on this $d_S(v)$ property. We review these scoring functions below.

During the search process, OBMA (2020) and TSQC (2019) maintain an infeasible solution S of size k where the current best found size is k-1. These algorithms exchange two vertices iteratively until S becomes a feasible solution, where exchanging two vertices means removing one vertex u from S and adding another vertex v into it. The score of exchanging two vertices, denoted as $swap\_score(u, v)$, is used to choose the pair of vertices to exchange.

$$swap\_score(v, u) = d_S(v) - d_S(u) - e_{uv},$$

where $u \in S$, $v \in V \setminus S$, and $e_{uv} = 1$ if $(u, v) \in E$, and $e_{uv} = 0$ otherwise.

Another scoring function was used in some versions of BRKGA (Pinto et al. 2015, 2018, 2019). For example, the BRKGA-IG* algorithm (2018) works as follows: during each iteration, starting from a candidate solution S, it first removes some vertices from S according to the d_S value, and then adds some vertices based on the diff information.

$$\text{diff}(v) = d_{CL}(v) + |CL|(d_S(v) - \gamma(|S| + 1))$$

where $v \in CL$ and $CL = \{u \mid u \in V \setminus S, dens(G[S \cup \{u\}]) \geq \gamma\}$. Another BRKGA algorithm called BRKGA-LSQClique (2019) hybridizes an exact enumeration algorithm (Ribeiro and Riveaux 2019) with BRKGA-IG*. The difference between BRKGA-IG* and BRKGA-LSQClique is that during the addition process, the latter one calls the exact algorithm with a certain probability and otherwise uses the same scoring function as BRKGA-IG* to add vertices.

The Saturation Based Function

Previous heuristic algorithms for MQCP usually consider $d_S(v)$ in the primary scoring function. In our algorithm, we directly use the d_S information as the primary scoring function to decide which vertex should be operated. However, the values of d_S are often the same during the search. According to our experiments, 14.64% candidate vertices on average have the same best score values in our algorithm. To further select a vertex among these vertices with the same best d_S, we design the secondary scoring function.

We distinguish the edges into three types, according to a measurement on the number of endpoints included in S, denoted as $\lambda(e)$ (S is the maintained candidate solution). Specifically, for $\forall e \in E$, $\lambda(e)$ has three possible values.

- $\lambda(e) = 0$ means none of endpoints of e is included in S.
- $\lambda(e) = 1$ means only one endpoint of e is included in S. An edge is said $critical$ if and only if $\lambda(e) = 1$.
- $\lambda(e) = 2$ means both endpoints of e are in S. An edge is said $full$ if and only if $\lambda(e) = 2$.

As MQCP mainly concerns about the density of all subsets of the selected vertices, it is intuitive to pick vertices which share more edges. A mechanism that encourages *critical* edges to transfer to *full* edges would help to achieve this end. Therefore, the algorithm prefers the vertices that are incident to more critical edges when choosing a vertex $v \notin S$ to be added to S, and prefers vertices that are incident to fewer full edges when choosing a vertex to be removed from S. This has been embodied in $d_S(v)$, which serves as the primary scoring function in our algorithm.

Nevertheless, the $d_S(v)$ function only considers the current state and ignores the situation of the passed steps. In our opinion, we should also take into account the history information about $d_S(v)$ over a recent period. Based on these considerations, we propose a function which can be seen as the cumulative effect of this measurement over a recent period. Before we present the function, we first define a vertex property named *saturation* as follows.

Definition 1. *For a local search algorithm with an input graph $G = (V, E)$, the saturation of a vertex v at step t is $\Gamma_t(v) = \sum_{e \in E(v)} I(e)$, where $I(e)$ is an indicator for critical edges, i.e, it is 1 if $\lambda(e) = 1$ at step t and 0 otherwise.*

Based on the concept of *saturation* of vertices, we design a function named *cumulative saturation*, which serves as the secondary scoring function for our local search algorithm.

Definition 2. *For a local search algorithm with an input graph $G = (V, E)$, the cumulative saturation of a vertex v is defined as*

$$\Gamma(v) = \sum_{t=t_0(v)}^{T} \Gamma_t(v) - T_S(v) \cdot \Delta_G,$$

where T is the current step number, t_0 is the step number at which v changed its state most recently (being added or removed w.r.t. S), while $T_S(v)$ is the number of steps at which $v \in S$ since t_0.

To understand and use this function, we need to distinguish the two cases $v \notin S$ and $v \in S$ (at the current step).

(1) If $v \notin S$ at the current step, then $\Gamma(v)$ equals to $\sum_{t=t_0(v)}^{T} \Gamma_t(v)$, as $T_S(v) = 0$. In this case, $\Gamma(v)$ is the sum of $\Gamma_t(v)$ over steps since $t_0(v)$, so it is obvious that we should prefer to add vertices with bigger $\Gamma(v)$.

(2) For the case $v \in S$, however, the offset $T_S(v) \cdot \Delta_G \neq 0$. Indeed, in this case we always have $\Gamma(v) \leq 0$, as shown below.

$$\begin{aligned}\Gamma(v) &= \sum_{t=t_0(v)}^{T} \Gamma_t(v) - T_S(v) \cdot \Delta_G \\ &= \sum_{t=t_0(v)}^{T} (\Gamma_t(v) - \Delta_G) \leq 0\end{aligned}$$

For choosing a vertex $v \in S$ to be removed from S, we should prefer the vertices with smaller $\Gamma(v)$ (i.e., bigger $|\Gamma(v)|$), as will be described later. This is not so obvious as the case (1), and we explain it below. i) We introduce the offset to make $\Gamma(v)$ have negative relation to the number of passed steps for v staying in S since t_0, which means that the longer time a vertex v stays in S, the smaller $\Gamma(v)$ value it would have. For the sake of diversification, we believe a vertex staying too long in S should be preferable to removing from S. ii) On the other hand, the $\Gamma_t(v)$ value for the vertex $v \in S$ measures the number of its neighbors outside S. Thus, a vertex with the small $\sum_{t=t_0}^{t} \Gamma_t(v)$ means that it is of lower potential to expand S via this vertex, as it is adjacent to relatively fewer vertices outside S.

To strike a good balance, the above two factors are taken into account for the case $v \in S$ by setting the function as $\Gamma(v) = \sum_{t=t_0}^{t} (\Gamma_t(v) - \Delta_G)$. Particularly, the offset at each step is set as Δ_G for a good reason, as it is sufficient to make $\Gamma_t(v) - \Delta_G$ be negative and also it is close enough to the biggest possible value of $\Gamma_t(v)$.

Vertex Selection Rules

Put the above considerations together, we present two selection rules for adding and removing operations respectively.

Add Rule. Select a vertex v with the highest $d_S(v)$ value, breaking ties by preferring the one with the highest $\Gamma(v)$ value, further ties are broken randomly.

Remove Rule. Select a vertex v with the lowest $d_S(v)$ value, breaking ties by preferring the one with the highest $|\Gamma(v)|$ value, further ties are broken randomly.

According to the analysis of our experimental results, resolving ties randomly after the secondary scoring function has a 7.2% probability on average. When our algorithm doesn't use the secondary scoring function, resolving ties randomly has a 60.7% probability on average.

Bounded Configuration Checking Strategy

Configuration checking (CC) firstly proposed by Cai (2011) is a diversification strategy that has been widely used in many NP-hard problems. It can be described as following:

when selecting a vertex to be added into S, for a vertex $v \notin S$, if the configuration of v (usually denoted by the states of vertices in $N_G[v]$, i.e., in S or not in S) has not changed since v's last removing from S, which means the circumstance of v has not changed, then v should not be added back to S. In practice, it is usually implemented with an array named $confChange$, where $confChange(v) = 1$ means v is allowed to be added into S, and $confChange(v) = 0$ means it is forbidden to be added.

CC Strategies for Clique Problems

Among the variants of CC, two of them have been developed in the context of clique problems, including strong configuration checking (SCC) strategy for the maximum weight clique problem (Wang et al. 2020) and dynamic-threshold configuration checking (DCC) strategy for the maximum k-plex problem (Chen et al. 2020).

Different from the original CC strategy, SCC allows a vertex v to be added into the candidate solution if some (more than one) neighbors of v have been added since the last removal of v.

DCC can be considered as a quantitative version of SCC, which can better exploit the vertex degree information in the strategy and has been shown more effective for the max k-plex problem. It allows to add a vertex v if a certain amount of vertices in $N_G(v)$ have been added, which is controlled by a dynamic threshold. Specifically, a vertex v is allowed to be added into the candidate solution only when $confChange(v) \geq threshold(v)$ where $threshold$ is used to control the forbidding strength. Whenever v is added into the solution, $threshold(v)$ is increased by one.

The BoundedCC Strategy

We propose a new CC strategy named BoundedCC. The idea is to impose stricter CC condition to the vertices that are more frequently operated. This could be implemented by a CC strategy similar to DCC with dynamic thresholds. Nevertheless, we notice that DCC does not limit the maximum value of the threshold of CC condition, which makes some frequently operated vertices have very large threshold compared to other vertices and thus are forbidden for a long period of time.

We conducted experiments to investigate the distribution of the threshold values of CC condition in DCC. It was observed that, by taking average over all instances in this work, the maximum threshold value is 6.3 times more than the average threshold value. Particularly, the gap is significant for dense graphs. For example, for *frb59-26-4* with $\gamma = 0.9$, the maximum threshold value is 20.4 times more than the average value, which means there are some vertices with very high threshold of CC condition and thus would be forbidden for a long period of time. This consists with the intuition: the configurations of vertices with high degree are likely to change, as the change on any of its neighbors would cause an increase on the total change of the $confChange$ values. This makes the threshold values increase quickly for such vertices. In fact, as our experiments show, BoundedCC shows particularly improvements over DCC for dense graphs. Specially, for all graphs with density larger than 80%, our algorithm replacing DCC with BoundedCC performs better for 6 graphs and no worse solutions for others. Moreover, the proposed BoundedCC makes the maximum threshold value only 1.5 times more than the average threshold value.

In this work, we set an upper bound on the threshold of CC condition (denoted by $ub\_threshold$), so that no vertex would be forbidden for too long time. Moreover, when the threshold exceeds $ub\_threshold$, we reset the threshold to 1 to increase the diversity of search process. Specifically, our CC strategy can be described as follows.

BoundedCC-InitialRule. For $\forall v \in V$, $confChange(v)$ and $threshold(v)$ are both initialized to 1.

BoundedCC-AddRule. When v is added into the candidate solution, for $\forall u \in N_G(v)$, $confChange(u)$ and $threshold(v)$ is increased by one, respectively. If $threshold(v) > ub\_threshold$, $threshold(v)$ should be reset to 1.

BoundedCC-RemoveRule. When v is removed from the candidate solution, $confChange(v)$ is set to 0.

Note that for $v \in V \setminus S$, if $confChange(v) \geq threshold(v)$, then we say this vertex satisfies the BoundedCC constraint and v can be selected into S.

The NuQClq Algorithm

According to the above novel strategies, we propose a local search algorithm for the MQCP named NuQClq, which is outlined in Algorithm 1. Initially, the best found γ-quasi-clique, denoted as S^*, is initialized to an empty set. In each loop (lines 2–5), after constructing an initial candidate solution (line 3), the algorithm calls the local search procedure (line 4). If the local best solution S_{lbest} in this search trajectory is better than the global best solution S^*, then S^* is updated by S_{lbest} (line 5). At last, S^* is returned (line 6).

Before introducing our *InitConstruct* function, we need to define an additional property of each vertex $v \in V$, denoted as $freq(v)$, which is used to represent the frequency of operation. The $freq$ value of each vertex is initialized as 0. During the search process, if vertex v is operated (i.e., to be removed or added), then $freq(v)$ will be increased by one. The *InitConstruct* function works as follows: given an empty candidate solution S, the algorithm first selects a vertex with the lowest $freq$ value, breaking ties randomly. Then it performs a series of iterations until S cannot be expanded. In each iteration, if $N_G(S)$ is not empty, it picks a vertex $v \in N_G(S)$ that has the lowest $freq$ value with a probability of β, and with a probability of $1 - \beta$, a random vertex $v \in N_G(S)$ is chosen. If $N_G(S)$ is empty, then a random vertex $v \in V \setminus S$ is selected. If the density of the subgraph induced by $S \cup \{v\}$ is smaller than γ, then the algorithm returns S, and otherwise the vertex v is added into S. The complexity of *InitConstruct* function is $O(|V||S|)$.

QCSearch iteratively modifies the candidate solution until the unimproved step count ($stepUnimpr$) reaches the limit $stepMax$, as is shown in Algorithm 2. At the beginning, $stepUnimpr$ is set to 0 and S_{lbest} is initialized to S. During each step, if S is a γ-quasi-clique, S_{lbest} is updated by S and $stepUnimpr$ is reset to 0 (line 4). Then the algorithm calls

Algorithm 1: the NuQClq Algorithm

Input: graph $G = (V, E)$, the *cutoff* time, parameter γ
Output: the best γ-quasi-clique S^* found

$S^* := \emptyset$;
while *elapsed time* $<$ *cutoff* **do**
  $S := InitConstruct()$;
  $S_{lbest} := QCSearch(S)$;
  if $|S_{lbest}| > |S^*|$ **then** $S^* := S_{lbest}$;
return S^*;

Algorithm 2: QCSearch(S)

Input: a feasible solution S
Output: the best solution S_{lbest} found

1 $stepUnimpr := 0$, $S_{lbest} := S$;
2 **while** $stepUnimpr < stepMax$ **do**
3 **if** $dens(G[S]) \geq \gamma$ **then**
4 $S_{lbest} := S$, $stepUnimpr := 0$;
5 $v := SelectVertexToAdd(S)$;
6 $S := S \cup \{v\}$;
7 update $threshold$ and $confChange$ according to **BoundedCC-AddRule**;
8 **continue**;
9 $v := SelectVertexToAdd(S)$;
10 $S := S \cup \{v\}$;
11 select $u \in candRemove(S)$ according to **Remove Rule** and $S := S \setminus \{u\}$;
12 update $threshold$ and $confChange$ according to **BoundedCC-AddRule**, **BoundedCC-RemoveRule**;
13 update $\Gamma(v)$ for $\forall v \in N_G[S]$;
14 $stepUnimpr := stepUnimpr + 1$;
15 **return** S_{lbest};

Algorithm 3: SelectVertexToAdd(S)

Input: the current solution S
Output: the candidate added vertex u

1 **if** $N_G(S)$ *is empty* **then**
2 select $u \in V \setminus S$ randomly;
3 **else**
4 **if** $hasCand(v) = 0$ *for* $\forall v \in S$ **then**
5 select a random vertex $v \in S$;
6 $candAdd(S) := \{u \mid u \in N_G(v) \setminus S\}$;
7 **else**
8 select a random vertex v with $hasCand(v) \neq 0$;
9 $candAdd(S) := \{u \mid confChange(u) \geq threshold(u), u \in N_G(v) \setminus S\}$;
10 select $u \in candAdd(S)$ according to **Add Rule**;
11 **return** u;

the SelectVertexToAdd(S) function to get the added vertex which is described in Algorithm 3. First, if $N_G(S)$ is empty, the algorithm randomly selects a vertex $v \in V \setminus S$ and returns it. Otherwise, we use a Boolean indicator $hasCand(v)$ to denote whether there are vertices in $N_G(v)$ that satisfy the BoundedCC constraint. If the $hasCand$ values of all vertices in S are false, then a random vertex v will be selected and $candAdd(S)$ will be constructed. If there exist some vertices in S whose $hasCand$ values are true, then a random vertex from these vertices will be selected and $candAdd(S)$ will contain the vertices in $N_G(v) \setminus S$ which satisfy the BoundedCC constraint. After that, the algorithm chooses a vertex from $candAdd(S)$ according to Add Rule and returns the selected vertex. During this process, the maximum size of added candidate set is only Δ_G. After adding the selected vertex v into S, the value of $threshold$ and $confChange$ are updated accordingly. The QCSearch algorithm skips to the next iteration (line 8).

When S is not a feasible solution, our algorithm executes an exchange step (lines 9–14) until S becomes a feasible solution or $stepUnimpr$ reaches $stepMax$. Each exchanging step consists of an adding stage and a removing stage. In the adding stage, the algorithm first calls SelectVertexToAdd function to get the selected vertex and then adds it into S. Afterwards, in the removing stage, the algorithm chooses a vertex $u \in candRemove(S)$ according to Remove Rule (line 11). A trick is used to accelerate the removing stage. If $dens(G) > 0.5$, $candRemove(S)$ is composed of all vertices in S except the just added vertex. Otherwise, we adopt the BMS strategy (Cai, Lin, and Luo 2017), i.e., randomly selecting $|S|/2$ vertices to compose $candRemove(S)$. This strategy works well when the graph is very sparse, and makes the solution more diversified. After removing the vertex u from solution S, $threshold$, $confChange$, and Γ need to be updated accordingly (lines 12–13). At last, $stepUnimpr$ is increased by one. The complexity of each iteration in the local search process is $O(max(\Delta_G^2, |V|))$.

Experimental Evaluation

We evaluate NuQClq on a broad range of classic benchmarks as well as sparse instances, and compare it with three state-of-the-art heuristic algorithms.

Since previous works use different instances, we select all used instances from (Pinto et al. 2019; Djeddi, Haddadene, and Belacel 2019; Zhou, Benlic, and Wu 2020). To be specific, we consider 289 instances, which are mainly divided into two parts: (1) 187 classic instances from DIMACS benchmark (Johnson 1993)[1] and BHOSLIB benchmark (Xu et al. 2007)[2]; (2) 102 sparse instances whose density is from 0.00014% to 3.869% from Florida Sparse Matrix Collection (Davis and Hu 2011)[3] and Stanford Large Network Dataset Collection (Rossi and Ahmed 2015)[4]. Note that, the same graph with different γ is seen as the different instances.

According to the literature, the best values found by three parallel works, i.e., BRKGA-LSQClique (2019), TSQC (2019) and OBMA (2020), are better than those obtained by previous algorithms. Thus, we compare NuQClq with these three state-of-the-art heuristic algorithms. Since the source or binary codes of BRKGA-LSQClique and OBMA are not available to us, we have to reimplement them. The code of

[1] http://archive.dimacs.rutgers.edu/pub/challenge/

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

[3] https://sparse.tamu.edu/

[4] http://networkrepository.com

Parameters	Ranges	Final values
$ub\_threshold$	$\{3, 5, 7, 9\}$	7
β	$\{0.5, 0.6, 0.7, 0.8, 0.9\}$	0.8
$stepMax$	$\{3000, 4000, 5000\}$	4000

Table 1: Tuned NuQClq parameters based on the irace tool.

TSQC is kindly provided by the authors. NuQClq, OBMA, and BRKGA-LSQClique are all implemented in C++ and complied by g++ with -O3 option, while TSQC is compiled by MATLAB.

All experiments are run on Intel Xeon E5-2640 v4 @ 2.40GHz CPU with 128GB RAM under CentOS 7.5. As for BRKGA-LSQClique, TSQC and OBMA, we set the parameters as same as what described in the corresponding literature, respectively. For each instance, all algorithms are executed 10 times with random seeds from 1 to 10 and a cutoff time of 1000 seconds. For each instance, *max* denotes the best size found (i.e., maximal solution value), and *avg* denotes the average size obtained over 10 runs. When *max*=*avg*, we do not report *avg*. The bold values in the tables indicate the best solution among all the algorithms. Several instances of two selected benchmarks are so easy that all algorithms obtain the same solution quality. Due to the limitation of space, we do not present these results in our tables. Detailed results and our sourced code are reported in github[5].

The automatic configuration tool irace (2016) is used to tune parameters for NuQClq. For the training set, we collect 60 instances randomly chosen from all instances. The tuning process is given a budget of 3000 runs with a time budget of 1000 seconds per run. The final values obtained by irace are represented in Table 1. We have also run the combinations of all the possible parameters on the training subset. Experiments show that the maximal solution size obtained by the best parameter settings have on average 0.1686 larger than the combinations of other parameters. Thus, our method is quite stable depending on these parameters.

Results on DIMACS and BHOSLIB Benchmarks

The results are reported in Table 2. For all classic instances, NuQClq finds better solution values than BRKGA-LSQClique, TSQC, and OBMA for 84, 30, and 14 instances, except only one. For instances where both algorithms find the same best solution values, the average solution value found by NuQClq is always larger than that of competitors, except for two instances. When NuQClq and the corresponding algorithm find the same quality values (i.e., same maximal and average values), we also present the comparisons of run time on these benchmarks in Figure 1, which clearly shows the superiority of NuQClq.

Results on Sparse Instances

Table 3 summarizes the results on all sparse instances. Once again, the performance of NuQClq is better than BRKGA-LSQClique, TSQC, and OBMA for 20, 4, and 23 instances, with only one exception. For the instances where NuQClq gets a solution with the same maximal value as the corresponding competitor, NuQClq generates better average values for most instances with 6 exceptions. The superiority of NuQClq on the sparse instances is also clearly illustrated by Figure 2, which summarizes the run time distributions of the MQCP algorithms when both NuQClq and one competitor find the same quality. For all instances, the speed-up ratio of average run time is represented in Table 4.

[5]https://github.com/yiyuanwang1988/NuQClq.git

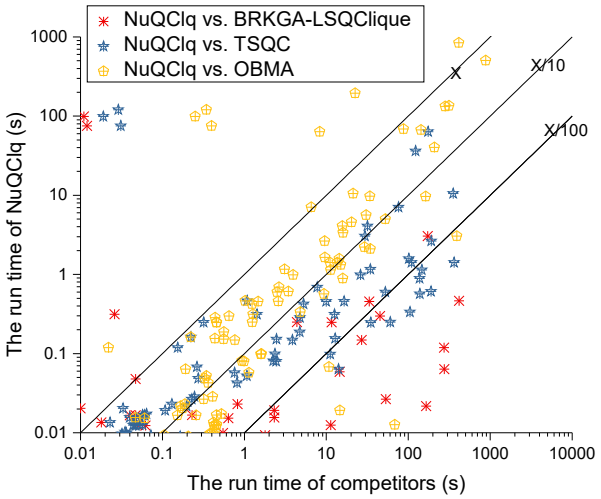

Figure 1: The comparisons of run time on classic benchmarks.

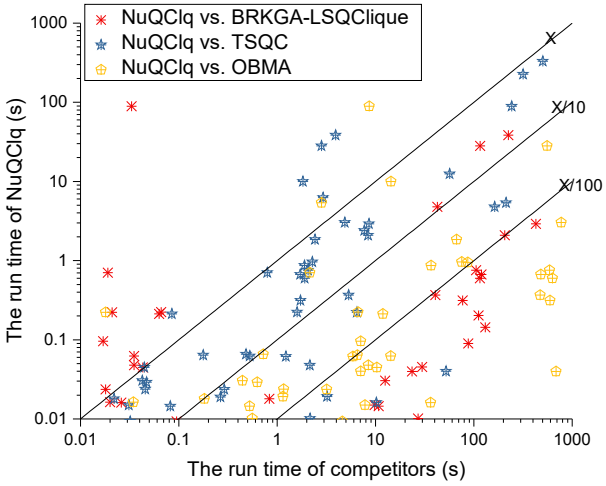

Figure 2: The comparisons of run time on sparse instances.

Instances	γ	BRKGA-LSQClique max(avg)	TSQC max(avg)	OBMA max(avg)	NuQClq max(avg)	Instances	γ	BRKGA-LSQClique max(avg)	TSQC max(avg)	OBMA max(avg)	NuQClq max(avg)
brock400_1	0.999	25	**27**(25.8)	26(25.1)	**27(26)**	frb40-19-5	0.999	39(38.2)	39(38.9)	**40**(39.4)	**40(39.7)**
brock400_2	0.999	25	**29**(28.2)	27(25.3)	**29**		0.95	100	**101**	**101**	**101**
	0.8	**187**(186.2)	**187**	**187**	**187**		0.9	333	**334**	**334**	**334**
brock400_3	0.999	**31**(26.2)	**31**	28(26.2)	**31**	frb45-21-1	0.999	43(42.3)	44(43.1)	**46**	**46**
brock800_1	0.9	**43**(42.1)	**43**	**43**	**43**		0.95	125(124.4)	**127**	**127**	**127**
brock800_2	0.999	21	21	21	**24**		0.9	508	**509**	**509**	**509**
	0.8	95	**96**	**96**	**96**	frb45-21-2	0.999	43(42.2)	**46**(43.4)	**46**(45.4)	**46**
brock800_3	0.999	**22**(21.1)	**22**	**22**	**22**		0.95	123(122.1)	**124**	**124**	**124**
	0.9	**43**(42.3)	**43**	**43**	**43**		0.9	491	**492**	**492**	**492**
	0.8	93	**94**	**94**	**94**	frb45-21-4	0.999	42	44(43.3)	**46**(45.9)	**46**
C1000.9	0.999	68(67.7)	**70**(69.6)	**70(69.9)**	**70(69.9)**		0.95	131(130.7)	**132**	**132**	**132**
	0.95	220(219)	**222**	**222**	**222**		0.9	554	**555**	**555**	**555**
C2000.9	0.999	76(74.7)	80(78.8)	81(**79.5**)	**82(79.5)**	frb45-21-5	0.999	43(42)	**46**(43.3)	45(44.4)	**46**
	0.95	283(280.3)	**288**(284.4)	**288**	**288**		0.95	123(122.8)	**125**	**125**	**125**
C4000.5	0.999	17	17(16.4)	**18**	**18**	frb50-23-1	0.999	48(47.1)	49(48.5)	**51(50.4)**	**51**(50.2)
	0.9	29(28.8)	30(29.3)	**31(30.1)**	30		0.95	162(160.6)	**164**	**164**	**164**
	0.8	51(49.9)	**53**(51.8)	**53**(52.1)	**53(52.9)**	frb50-23-2	0.999	48(47.4)	50(48.6)	**51(50.1)**	**51(50.1)**
C500.9	0.999	57	**58**	**58**	**58**	frb50-23-4	0.999	48(47.2)	49	**51(50.9)**	**51(50.9)**
	0.95	158	**159**	**159**	**159**		0.95	160(158.7)	**162**	**162**	**162**
DSJC1000.5	0.999	**15**	**15**(14.7)	**15**	**15**		0.9	**772**(771.6)	**772**	**772**	**772**
	0.8	40	**41**	**41**	**41**	frb50-23-5	0.999	47	49(48.6)	**51**(50.5)	**51(50.8)**
DSJC500.5	0.999	**13**	**13**(12.9)	**13**	**13**		0.95	162(161.8)	**165**	**165**	**165**
gen200_p0.9_44	0.999	42(41.3)	**44**	**44**	**44**	frb53-24-1	0.999	50(49.5)	51(50.6)	**53**(52.7)	**53(52.9)**
gen400_p0.9_55	0.999	53	**55**	**55**	**55**		0.95	202(201.8)	**204**	**204**	**204**
gen400_p0.9_65	0.999	**66**(62)	**66**	**66**	**66**	frb53-24-2	0.999	51(50)	52(51.6)	**54**(52.8)	**54(53)**
hamming10-4	0.95	84(83.6)	87(85.3)	87(86.5)	**88(87.3)**		0.95	175(173.4)	**177**	**177**	**177**
keller6	0.95	279(275.8)	285	284(281)	**286(285.2)**		0.9	**926**(925.9)	**926**	**926**	**926**
MANN_a27	0.999	**135**	**135**(134.8)	**135**(134.2)	**135**	frb53-24-4	0.999	50(49.6)	52(51.1)	53(52.6)	**54(52.7)**
MANN_a45	0.999	435(430.9)	435(434.6)	433(431.7)	**441**		0.95	182(181.6)	**185**	**185**	**185**
san200_0.7_1	0.95	**57**	**57**(55)	**57**(52.8)	**57**	frb53-24-5	0.999	50(49.6)	52(51.1)	**54**(52.7)	**54(52.8)**
san200_0.7_2	0.95	**34**	**34**	**34**(32.4)	**34**		0.95	172(171.3)	**175**	**175**	**175**
san400_0.7_2	0.95	64(62.3)	62	62(55.8)	**65(64)**	frb56-25-1	0.999	53(51.9)	55(53.6)	**56**(55.3)	**56(55.5)**
san400_0.7_3	0.95	**40**(38.9)	**40**(37.9)	37(35.4)	**40(39.6)**		0.95	228(227.5)	**231**	**231**	**231**
frb30-15-4	0.999	29	**30**	**30**	**30**	frb56-25-2	0.999	53(52.2)	54(53.4)	**56**(55.1)	**56(55.3)**
	0.95	**61**(60.8)	**61**	**61**	**61**		0.95	217(214.9)	**220**(219.1)	**220**	**220**
frb30-15-5	0.999	29	**30**	**30**	**30**	frb56-25-4	0.999	53(52.3)	55(54)	**57(55.8)**	**57**(55.4)
frb35-17-1	0.999	33	34	**35**	**35**		0.95	199(198)	**202**(201.7)	**202**	**202**
	0.95	78	**79**	**79**	**79**		0.9	1125	**1126**	**1126**	**1126**
frb35-17-2	0.999	34(33.1)	**35**	**35**	**35**	frb56-25-5	0.999	53(52.2)	55(54.1)	**57**(55.6)	**57(55.7)**
	0.9	207	**208**	**208**	**208**		0.95	204(203.1)	**208**(207.3)	**208**	**208**
frb35-17-4	0.999	33	**35**(34.1)	**35**	**35**		0.9	1138	**1139**	**1139**	**1139**
	0.95	**81**(80.8)	**81**	**81**	**81**	frb59-26-1	0.999	56(54.7)	57(56.3)	**59**(58.2)	**59(58.3)**
frb35-17-5	0.999	34	**35**(34.9)	**35**	**35**		0.95	245(243.6)	**248**(247.6)	**248**	**248**
	0.95	**80**(79.5)	**80**	**80**	**80**	frb59-26-2	0.999	56(55)	57(56.6)	58	**59(58.1)**
frb40-19-1	0.999	39(38.1)	**40**(39)	**40**	**40**		0.95	240(239.7)	**244**(243.2)	**244**	**244**
	0.95	**114**(113)	**114**	**114**	**114**	frb59-26-4	0.999	55(54.9)	57(56.3)	**59(58.1)**	**59(58.1)**
frb40-19-2	0.999	39(38.1)	39(38.9)	**40**(39.5)	**40**		0.95	235(233.6)	**238**(237)	**238**	**238**
	0.95	104(103.2)	**105**	**105**	**105**	frb59-26-5	0.999	55(54.3)	57(56.5)	**60**(58.6)	**60(58.7)**
frb40-19-4	0.999	39(38)	39(38.9)	**40**(39.9)	**40**		0.95	228(226.7)	**231**(230.6)	**231**	**231**
	0.95	**97**(96.3)	**97**	**97**	**97**		0.9	1360	**1361**	**1361**	**1361**
	0.9	292(291.7)	**293**	**293**	**293**	frb100-40	0.999	88(87.2)	95(94.1)	98(97.7)	**99(98.6)**
							0.95	1838(1837)	**1842**	**1842**	**1842**

Table 2: Experiment results on the classic benchmark.

The Effectiveness of the Proposed Strategies

In this subsection, we compare NuQClq with four alternative versions: (1) NuQ1 uses the random selection strategy when the candidate vertices have the same $score$ value, instead of our secondary scoring function Γ; (2) NuQ2 utilizes the DCC strategy (Chen et al. 2020) instead of our BoundedCC strategy; (3) NuQ3 utilizes the SCC strategy (Wang

Instances	γ	BRKGA-LSQClique max(avg)	TSQC max(avg)	OBMA max(avg)	NuQClq max(avg)	Instances	γ	BRKGA-LSQClique max(avg)	TSQC max(avg)	OBMA max(avg)	NuQClq max(avg)
as-skitter	0.9	**112**	**112**	66(52.6)	**112**	roadNet-CA	0.9	**4**(3.1)	3	3	**4**
	0.8	**152**(151.2)	**152**	77(62.4)	**152**		0.8	**5**(4.7)	**5**(4.3)	**5**(4.4)	**5**
	0.7	**209**	**209**	93(78.5)	**209**		0.7	5	5(4.8)	5(4.9)	**6**
	0.6	**271**(270.9)	**271**	140(114.9)	**271**		0.6	**6**	**6**(5.4)	**6**(5.9)	**6**
ca-GRQC	0.5	76(73.6)	**81**	**81**	**81**	roadNet-PA	0.9	3	3	**4**(3.2)	**4**
California	0.9	7	**12**(7.5)	**12**	**12**		0.8	**5**(4.5)	**5**(4.1)	**5**(4.4)	**5**
	0.8	10	**13**	**13**	**13**		0.7	**5**	**5**(4.8)	**5**	**5**
	0.7	12	**14**	**14**	**14**		0.6	**6**	**6**(5.7)	**6**(5.9)	**6**
	0.6	15	**16**	**16**	**16**	roadNet-TX	0.9	3	3	3	**4**
Harvard500	0.9	21	**23**	**23**	**23**		0.8	**5**(4.6)	**5**(4)	**5**(4.4)	**5**
	0.8	22	**24**	**24**	**24**		0.7	**5**	**5**(4.7)	**5**(4.9)	**5**
	0.7	24	**26**	**26**	**26**		0.6	**6**	**6**(5.6)	**6**(5.9)	**6**
	0.6	27	**29**	**29**	**29**	web-Google	0.9	**60**(44.3)	**60**(49.3)	29(25.9)	**60**
	0.5	33(32.4)	**37**	**37**	**37**		0.8	**69**(53.4)	**69**(52.6)	34(32)	**69(68.9)**
LiveJournal	0.9	**430**(406.4)	**430**(263.8)	24(20.2)	**430(427.2)**		0.7	**74**(61.7)	**74**(67.5)	41(37.2)	**74**
	0.8	**472**(449.7)	**472(471)**	29(24.9)	**472(471)**		0.6	82(73.7)	**83**(73.5)	49(47)	**83**
	0.7	549(463.7)	**550**	43(29.8)	**550**	wiki-Talk	0.9	**58**(57.9)	**58**	**58**	**58**
	0.6	**628**(537.3)	**628**	55(37.4)	**628**(619.8)		0.6	**222**	**222**	**222**(211.1)	**222**
netscience	0.6	**27**	**27**(26.9)	**27**	**27**	wiki-topcats	0.9	**47**(37.6)	**47(39.3)**	23(22.1)	**47**(38.4)
orkut	0.9	81(67.4)	**124**(99.9)	55(40.2)	**124(116.9)**		0.8	59(58.1)	**68(60.9)**	32(28.5)	**68**(54.5)
	0.8	118(111.4)	**166(162)**	97(61.2)	**166**(146.3)		0.7	101(92.1)	**102**	46(44.3)	**102**(83.1)
	0.7	172(171.3)	**202**	153(100.7)	201(194.9)		0.6	**171**(163.4)	**171**	101	**171**(143)
	0.6	**259**	**259**	243(161.4)	**259**						

Table 3: Experiment results on the sparse instances.

Instances	vs. BRKGA-LSQClique	vs. TSQC	vs. OBMA
Classic Benchmark	15.99x	5.92x	1.41x
Sparse Instances	10.54x	2.15x	33.9x

Table 4: The speed-up ratio of average run time of NuQClq and the corresponding competitors when they obtain the same solution quality. The speed-up ratio larger than 1 means that NuQClq is faster than the corresponding competitor.

	vs. NuQ_1	vs. NuQ_2	vs. NuQ_3	vs. NuQ_4
#Better	11	10	7	18
#Worse	0	1	1	2

Table 5: Comparing NuQClq with four modified versions on all benchmarks. #Better and #Worse denote the number of instances where NuQClq finds better and worse results, respectively.

et al. 2020) instead of our BoundedCC strategy; (4) NuQ4 replaces our initialization process with the previous restart initialization process, i.e., constructing the initial solution whose size is equal to the best obtained size plus one, which is similar to the initialization method used in TSQC and OBMA. As shown in Table 5, the results demonstrate that all proposed strategies are effective.

Conclusion

In this paper, we propose a novel cumulative saturation based vertex selection heuristic and a variant of configuration checking for the MQCP. Based on the above strategies, we develop a local search algorithm NuQClq. Experiments show that NuQClq significantly outperforms the state-of-the-art heuristic algorithms. The main technology of NuQClq is general with wide applications, and thus its core ideas would be applied in extensive real-world applications, such as social network, protein interaction network and resource optimization in cloud systems.

Acknowledgements

This work was supported by National Key R&D Program of China (2019AAA0105200), NSFC under Grant No. (61806050, 61972063, 61976050), the Fundamental Research Funds for the Central Universities (2412020FZ030, 2412018QD022), project of Jilin Provincial Science and Technology Department under Grant (20190302109GX), and Jilin Science and Technology Association (QT202005).

References

Abello, J.; Resende, M. G.; and Sudarsky, S. 2002. Massive quasi-clique detection. In *Latin American symposium on theoretical informatics*, 598–612.

Brunato, M.; Hoos, H. H.; and Battiti, R. 2007. On effectively finding maximal quasi-cliques in graphs. In *International conference on learning and intelligent optimization*, 41–55.

Cai, S.; Lin, J.; and Luo, C. 2017. Finding a small vertex cover in massive sparse graphs: construct, local search, and preprocess. *Journal of Artificial Intelligence Research* 59: 463–494.

Cai, S.; Su, K.; and Sattar, A. 2011. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. *Artificial Intelligence* 175(9-10): 1672–1696.

Chen, P.; Wan, H.; Cai, S.; Li, J.; and Chen, H. 2020. Local Search with Dynamic-Threshold Configuration Checking and Incremental Neighborhood Updating for Maximum k-plex Problem. In *AAAI*, 2343–2350.

Davis, T. A.; and Hu, Y. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38(1): 1–25.

Djeddi, Y.; Haddadene, H. A.; and Belacel, N. 2019. An extension of adaptive multi-start tabu search for the maximum quasi-clique problem. *Computers & Industrial Engineering* 132: 280–292.

Gao, J.; Chen, J.; Yin, M.; Chen, R.; and Wang, Y. 2018. An Exact Algorithm for Maximum k-Plexes in Massive Graphs. In *IJCAI*, 1449–1455.

Johnson, D. S. 1993. Cliques, coloring, and satisfiability: second DIMACS implementation challenge. *DIMACS series in discrete mathematics and theoretical computer science* 26: 11–13.

Khosraviani, A.; and Sharifi, M. 2011. A distributed algorithm for γ-quasi-clique extractions in massive graphs. In *International Conference on Innovative Computing Technology*, 422–431.

Lee, P.; and Lakshmanan, L. V. 2016. Query-driven maximum quasi-clique search. In *Proceedings of the 2016 SIAM International Conference on Data Mining*, 522–530.

Li, R.; Cai, S.; Hu, S.; Yin, M.; and Gao, J. 2018. NuMWVC: A Novel Local Search for Minimum Weighted Vertex Cover Problem. In *AAAI*, 8107–8108.

López-Ibáñez, M.; Dubois-Lacoste, J.; Cáceres, L. P.; Birattari, M.; and Stützle, T. 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives* 3: 43–58.

Luce, R. D.; and Perry, A. D. 1949. A method of matrix analysis of group structure. *Psychometrika* 14(2): 95–116.

Luo, C.; Cai, S.; Wu, W.; Jie, Z.; and Su, K. 2015. CCLS: An Efficient Local Search Algorithm for Weighted Maximum Satisfiability. *IEEE Trans. Computers* 64(7): 1830–1843.

Marinelli, F.; Pizzuti, A.; and Rossi, F. 2020. LP-based dual bounds for the maximum quasi-clique problem. *Discrete Applied Mathematics* 1–23.

Miao, Z.; and Balasundaram, B. 2020. An Ellipsoidal Bounding Scheme for the Quasi-Clique Number of a Graph. *INFORMS Journal on Computing* 1–30.

Oliveira, A.; Plastino, A.; and Ribeiro, C. 2013. Construction heuristics for the maximum cardinality quasi-clique problem. In *tenth metaheuristics international conference (mic 2013)*, 84.

On, B.-W.; Elmacioglu, E.; Lee, D.; Kang, J.; and Pei, J. 2006. Improving grouped-entity resolution using quasi-cliques. In *Sixth International Conference on Data Mining (ICDM'06)*, 1008–1015.

Pajouh, F. M.; and Balasundaram, B. 2012. On inclusionwise maximal and maximum cardinality k-clubs in graphs. *Discrete Optimization* 9(2): 84–97.

Pajouh, F. M.; Miao, Z.; and Balasundaram, B. 2014. A branch-and-bound approach for maximum quasi-cliques. *Annals of Operations Research* 216(1): 145–161.

Pastukhov, G.; Veremyev, A.; Boginski, V.; and Prokopyev, O. A. 2018. On maximum degree-based-quasi-clique problem: Complexity and exact approaches. *Networks* 71(2): 136–152.

Pattillo, J.; Veremyev, A.; Butenko, S.; and Boginski, V. 2013. On the maximum quasi-clique problem. *Discrete Applied Mathematics* 161(1-2): 244–257.

Pinto, B. Q.; Plastino, A.; Ribeiro, C. C.; and Rosseti, I. 2015. A biased random-key genetic algorithm to the maximum cardinality quasi-clique problem. In *eleventh metaheuristics international conference*, 1–4.

Pinto, B. Q.; Ribeiro, C. C.; Riveaux, J.-A.; and Rosseti, I. 2019. A BRKGA-based matheuristic for the maximum quasi-clique problem with an exact local search strategy. *RAIRO-Operations Research* 1–25.

Pinto, B. Q.; Ribeiro, C. C.; Rosseti, I.; and Plastino, A. 2018. A biased random-key genetic algorithm for the maximum quasi-clique problem. *European Journal of Operational Research* 271(3): 849–865.

Ribeiro, C. C.; and Riveaux, J. A. 2019. An exact algorithm for the maximum quasi-clique problem. *International Transactions in Operational Research* 26(6): 2199–2229.

Rossi, R. A.; and Ahmed, N. K. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. URL http://networkrepository.com.

Trukhanov, S.; Balasubramaniam, C.; Balasundaram, B.; and Butenko, S. 2013. Algorithms for detecting optimal hereditary structures in graphs, with application to clique relaxations. *Computational Optimization and Applications* 56(1): 113–130.

Tsourakakis, C.; Bonchi, F.; Gionis, A.; Gullo, F.; and Tsiarli, M. 2013. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 104–112.

Veremyev, A.; Prokopyev, O. A.; Butenko, S.; and Pasiliao, E. L. 2016. Exact MIP-based approaches for finding maximum quasi-cliques and dense subgraphs. *Computational Optimization and Applications* 64(1): 177–214.

Wang, Y.; Cai, S.; Chen, J.; and Yin, M. 2018. A Fast Local Search Algorithm for Minimum Weight Dominating Set Problem on Massive Graphs. In *IJCAI*, 1514–1522.

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

Wasserman, S.; Faust, K.; et al. 1994. *Social network analysis: Methods and applications*, volume 8. Cambridge university press.

Wu, Q.; and Hao, J.-K. 2015. A review on algorithms for maximum clique problems. *European Journal of Operational Research* 242(3): 693–709.

Xu, K.; Boussemart, F.; Hemery, F.; and Lecoutre, C. 2007. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artificial intelligence* 171(8-9): 514–534.

Zheng, Y.; Shi, C.; Kong, X.; and Ye, Y. 2019. Author Set Identification via Quasi-Clique Discovery. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 771–780.

Zhou, Q.; Benlic, U.; and Wu, Q. 2020. An opposition-based memetic algorithm for the maximum quasi-clique problem. *European Journal of Operational Research* 286(1): 63–83.