Journal of Artificial Intelligence Research 74 (2022) 1485-1513 Submitted 10/2021; published 07/2022

Improving Simulated Annealing for Clique Partitioning Problems

Jian Gao GAOJIAN@DLMU.EDU.CN
College of Information Science and Technology,
Dalian Maritime University, Dalian, China

Yiqi Lv LVYQMAIL@GMAIL.COM
Minghao Liu LIUMH@IOS.AC.CN
Shaowei Cai *(Corresponding author)* SHAOWEICAI.CS@GMAIL.COM
State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China

Feifei Ma *(Corresponding author)* MAFF@IOS.AC.CN
Laboratory of Parallel Software and Computational Science,
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China

Abstract

The Clique Partitioning Problem (CPP) is essential in graph theory with a number of important applications. Due to its NP-hardness, efficient algorithms for solving this problem are very crucial for practical purposes, and simulated annealing is proved to be effective in state-of-the-art CPP algorithms. However, to make simulated annealing more efficient to solve large-scale CPPs, in this paper, we propose a new iterated simulated annealing algorithm. Several methods are proposed in our algorithm to improve simulated annealing. First, a new configuration checking strategy based on timestamp is presented and incorporated into simulated annealing to avoid search cycles. Afterwards, to enhance the local search ability of simulated annealing and speed up convergence, we combine our simulated annealing with a descent search method to solve the CPP. This method further improves solutions found by simulated annealing, and thus compensates for the local search effect. To further accelerate the convergence speed, we introduce a shrinking factor to decline initial temperature and then propose an iterated local search algorithm based on simulated annealing. Additionally, a restart strategy is adopted when the search procedure converges. Extensive experiments on benchmark instances of the CPP were carried out, and the results suggest that the proposed simulated annealing algorithm outperforms all the existing heuristic algorithms, including five state-of-the-art algorithms. Thus the best-known solutions for 34 instances out of 94 are updated. We also conduct comparative analyses of the proposed strategies and show their effectiveness.

1. Introduction

Analyzing network structures is an important field in graph theory and data mining. Particularly identifying cliques (complete subgraphs) in a graph is a basic problem to specify cohesive structures with a wide range of practical applications, including community search in social networks, logistics, and biological computing (Ouyang et al., 2020; Fox et al., 2020; Alduaiji et al., 2018). However, identifying cliques from a graph cannot specify the structure of the entire graph, so the Clique Partitioning Problem (CPP) was proposed and discussed to analyze graph structures.

Given a complete edge-weighted undirected graph, a CPP is defined as to find a partition of vertices of the graph that maximizes the sum of edge weights that belong to each subset (clique). Since the problem has a strong relation with clustering in data mining (Wang et al., 2004; Matsunaga et al., 2009), it is widely used in various applications. By partitioning vertices into cliques, a number of important graphs from real-world applications, ranging from social network analysis and biology to industrial fields, such as manufacturing systems and logistics, can be well analyzed. In fact, the clique partitioning problem has received great success in modeling and solving real-world problems (Oosten et al., 2001; Grötschel & Wakabayashi, 1989; Dorndorf et al., 2008; Wang et al., 2006). As an example, in the biology field, clique partitioning was used for classification, where partitioning features with qualitative descriptions were discussed by Grötschel and Wakabayashi (1989). They also provided some benchmark instances from biology, economics, and the political sciences. It is also worth mentioning that Dorndorf (2008) employed the CPP to solve the airport gate assignment problem. The problem is to assign flights to airport gates, which is a key technique in airport scheduling. They modeled the assignment problem as a CPP, where aircraft have to be assigned to clusters and aircraft in a cluster can be assigned to the same gate. To avoid the conflict that two aircraft stay at a gate at the same time, a large negative value was enforced on an edge weight. Another interesting application of the CPP is flexible manufacturing systems. Wang et al. (2006) solved the group technology problem by clique partitioning to group machines and parts together so that the time and cost can be reduced.

1.1 Previous Works

We give a review of algorithms for the CPP in this subsection. Both exact algorithms and heuristic algorithms were proposed and studied extensively. Exact algorithms are used to find optimal solutions. Along this line, a number of exact algorithms have been proposed in the last three decades. The main contributions include cutting plane-based algorithms and branch-and-bound algorithms. Dorndorf and Pesch (1994) used a heuristic to compute lower bounds and guide branching in a branch-and-bound algorithm. They showed their method is better than other exact methods. Moreover, Ji and Mitchell (2007) used the branch-and-price algorithm to solve the CPP, and proposed effective cutting planes in the algorithm. They also indicated that their algorithm can solve randomly generated instances. Later, Jaehn and Pesch (2013) proposed a novel branch-and-bound algorithm by using a new upper bound based on the triangular restrictions. In addition, constraint propagation techniques were incorporated into the algorithm to reduce search space.

However, exact methods sometimes spend much time to complete search on the whole solution space to find and prove the best solution. Though they are able to guarantee the optimality, exact methods fail to solve large-scale instances from real-world applications in a reasonable time. An alternative way is to use heuristic algorithms. Heuristics, especially local search methods that move from solution to solution in the space of candidate solutions by applying local changes, play an important role in solving NP-hard problems. For the last decades, various heuristic algorithms have been proposed to find high-quality solutions for large-scale instances in an acceptable time and computer memory (Wang et al., 2017; Lu et al., 2020; Zhang et al., 2021; Lu et al., 2021; Bilandi et al., 2021). Among heuristic algorithms, tabu search (Glover, 1989, 1990) and simulated annealing (Skiscim & Golden, 1983; Kirkpatrick et al., 1983), which solved many graph problems successfully (Palubeckis et al., 2014; Bilandi et al., 2021; Lu et al., 2019), showed their efficiency in solving many benchmark instances of the CPP. In the early work of (De Amorim et al., 1992), De Amorim et al. applied tabu search and simulated annealing to this problem. They compared their algorithms with commonly used heuristics, and showed the effectiveness of these heuristic algorithms on a number of CPP instances which were not only generated randomly but also from real-life problems. Based on tabu search, Brusco and Köhn proposed two versions of new neighborhood search algorithms (Brusco & Köhn, 2009) and used a relocation algorithm to improve the solution quality, and then proposed an embedded tabu search method. The improved algorithm has a good performance for solving large and difficult instances compared with simulated annealing and a tabu search-based algorithm.

Recent works that have been published mainly aim at solving large-scale instances. Palubeckis et al. (2014) presented a new iterated algorithm, denoted by ITS, combining the tabu strategy, local search, and solution perturbation procedures. They carried out computational experiments and indicated that their algorithm can solve instances up to 2000 vertices efficiently, and outperforms existing methods. Later, Brimberg et al. (2017) transformed the CPP to the maximally diverse grouping problem, and then solved it with a skewed general variable neighborhood search algorithm (SGVNS). They showed significant better results than other heuristics proposed previously (e.g., the tabu search algorithm proposed by Brusco and Köhn (2009)), and they tested graph instances with 100 to 2000 vertices. Besides, Zhou et al. (2016) proposed a three-phase local search framework, called CPP-P^3, where a descent procedure, a tabu-based search phase and a directed perturbation phase were incorporated into the algorithm. The algorithm was compared with an iterated tabu search algorithm and SGVNS algorithm, and it has a good ability to deal with large instances up to 5000 vertices. Hudry (2019) discussed a descent algorithm with a mutation meta-heuristic to the CPP. The simple meta-heuristic was devised on the base of a local search framework. The meta-heuristic was tested on a small set of benchmarks and compared with a simulated annealing method, and is at least as efficient as the comparative simulated annealing method, but there is a lack of comparisons with recently proposed heuristic algorithms. Later, Hu et al. (2021) proposed a two-model local search algorithm with a self-adaptive parameter mechanism called TMLS_SA. They employed an exchange strategy and a perturbation method to exploit solution space, where a self-adaptive approach was used to determine the number of perturbed pairs. The local search algorithm was combined with a lock mechanism, which was used to prevent from immediately returning to a visited state. They also showed that their algorithm has a comparable performance with CPP-P^3 and outperforms other existing algorithms on benchmark instances with 2000 vertices at most. Very recently, Lu et al. (2021) have proposed an evolutionary algorithm, denoted by MDMCP, combining a crossover operator to improve individuals by preserving good properties of parents and incorporating an effective simulated annealing algorithm to enhance local search. They tested a wide range of benchmarks, including most instances used in previous experiments as well as some new large-scale instances whose vertex number is up to 7000, and carefully analyzed experimental results, showing that MDMCP dominates existing algorithms, such as ITS, SGVNS, CPP-P^3.

1.2 Contributions

With the rapid growth of the Internet and data mining technology, graphs from real-world applications are growing rapidly, and those graphs usually have a large number of vertices and edges. Newly emerged massive graphs present a great challenge for CPP algorithms. Though there have been some heuristic algorithms proposed recently to solve the CPP, they are not good enough to deal with large-scale instances.

As we mentioned above, among those methods, simulated annealing, a local search method allowing non-improving moves, performs well in solving the problem. It has been incorporated into various meta-heuristics and shows good performance. However, the local search ability of this technique can be further improved so that better solutions can be found faster. In this paper, we propose several new methods to improve the simulated annealing algorithm, and present a novel hybrid heuristic algorithm with the newly proposed methods. We denote our new algorithm by SACC, which stands for two main techniques it employs, *i.e.,* Simulated Annealing and Configuration Checking. Our main contributions are summarized as follows:

(1) An improved simulated annealing algorithm is proposed by incorporating a configuration checking strategy to avoid searching in a local area repeatedly. The simulated annealing algorithm for the CPP, though it can perform the non-improving moves to escape a local space, still suffers from the cycling issues. The configuration checking strategy can release local search algorithms from heavily cycling issues, and has been successfully applied in a number of NP-hard problems from the satisfiability problem to graph problems (Luo et al., 2013, 2015; Chu et al., 2017; Abramé et al., 2017; Chen et al., 2020; Wang et al., 2021). However, the existing configuration checking strategy does not work effectively to tackle complete graphs. Therefore, we propose a new form of configuration checking strategy to deal with the CPP. Configurations of vertices are well-defined, and a vertex will be forbidden to move to a certain cluster if its configuration is not changed. We also provide an efficient implementation of the strategy with timestamp comparison, and discuss the time complexity and space complexity of our new strategy.

(2) Though our new configuration checking strategy avoids cycling problems effectively, when we use the simulated annealing algorithm in practice, it may only converge to a local optimum within a fixed time budget, especially for solving large-scale CPP instances. We propose an iterated approach that repeats simulated annealing to search for a better solution. Additionally, to enhance the local search ability for solving large-scale graphs, we employ a descent search method to search for a better solution around the solutions found by the simulated annealing algorithm, and thus make local search more effective. The combination of descent search method can usually find a better solution than the one found by simulated annealing and thus it compensates for the relatively weak ability of local search in simulated annealing.

Based on the main methods mentioned above, we propose an iterated search framework, incorporating the simulated annealing method with configuration checking and descent search. Also, we propose a shrinking mechanism for initial temperature to accelerate convergence speed. Hence, the effectiveness of the hybrid heuristic algorithm is enhanced. In addition, a restart mechanism is employed if the hybrid heuristic algorithm fails to make an improvement within a certain number of iterations. Extensive experiments are conducted to evaluate our hybrid algorithm by testing benchmark instances previously used, with 7000 vertices at maximum. Computational experiments on 94 instances show that our SACC outperforms the state-of-the-art algorithms, and the best-known solutions of 34 instances were updated by it. Specifically, SACC dominates SGVNS, CPP-P^3, and TMLS_SA, and finds better solutions than MDMCP for a large portion of instances. Also, additional analyses are carried out to evaluate the strategies we propose, indicating that they can improve the behavior of our hybrid heuristic algorithm.

The remainder of this paper is organized as follows. The next section introduces notations and definitions used in the paper. Section 3 proposes our new configuration checking strategy based on timestamps, and an improved simulated annealing algorithm with the strategy. In Section 4, we propose the framework of our iterated simulated annealing algorithm, and explain our shrinking mechanism for the temperature factor to decline initial temperature for iterating simulated annealing. Section 5 presents experimental evaluations and comparative results with the state-of-the-art algorithms. Finally, we conclude our work.

2. Preliminaries

In this section, we provide notions and notations about the CPP used in this paper.

A complete edge-weighted undirected graph is defined as a triple $G = (V, E, W)$, where $V = \{1, 2, ..., n\}$ is the set of vertices and E is a set of edges containing all possible edges between each pair of vertices. Each edge $e = (i, j) \in E$ is associated with weight $w_{ij} \in W$, where i and j are the two vertices of the ends of edge e and w_{ij} is a real number, which is usually defined as the similarity of vertices.

Given a complete edge-weighted undirected graph $G = (V, E, W)$, the CPP is to find a partition that assigns all vertices of the given graph into k mutually disjoint subcliques (clusters) such that the sum of the edge weights within clusters is maximized, where k is an unfixed positive integer. Formally, we define $C = (c_1, c_2, ..., c_k)$ as a partition of the vertices, where $c_f(1 \leq f \leq k)$ is a cluster including a subset of vertices of the graph G. The objective function is defined as follow:

$f(C) = \sum_{c \in C}(\sum_{i,j \in c}(w_{ij}))$.

The goal is to maximize the function $f(C)$.

In general case, the decision version of the CPP is NP-complete if the given graph has both negative and positive edge weights (Wakabayashi, 1986), and if all the edge weights are non-negative or non-positive, solving the problem becomes a trivial task.

It is also noted that in some works, the concept of clique partitioning problem refers to another NP-hard problem that aims to partition the vertices of a given graph into the minimum number of subsets such that each subset is a clique (Sundar & Singh, 2017).

3. Simulated Annealing with Timestamp-based Configuration Checking

In this section, we first give a brief introduction to configuration checking and analyze why the classic configuration checking cannot be used for the CPP solving. Then, we propose a new version of configuration checking, called timestamp-based configuration checking, to settle the cycling issue. Afterwards, we present our enhanced simulated annealing method with the new configuration checking strategy.

3.1 The Configuration Checking Strategy

This subsection gives a brief introduction to configuration checking. The strategy has been successfully deployed in many applications to improve local search algorithms since it was first proposed by Cai et al. (2011). It is well known that many local search algorithms suffer from an issue called cycling problem during search. Greedy local search strategies may frequently visit a local optimum whose corresponding solution has been found recently (Michiels et al., 2007). So a local search algorithm may revisit some sub-optimal solutions and cost too much time on useless search. This may weaken its ability to find better solutions, and sometimes the algorithm gets trapped in a local area, leading to early convergence. To settle this issue, several strategies for forbidding cycling moves have been proposed, such as random walk and non-improving moves in the local search algorithms. Also, the tabu method proposed by Glover (1989, 1990) is a frequently used approach to forbid a recent reverse change, and thus prevent returning back to a previous state. Moreover, the configuration checking strategy was proposed to handle the cycling problem. It checks the configuration of a vertex and forbids a vertex to move back to its previous configuration. For more detailed relationships of the configuration checking strategy and the tabu method, we refer to (Cai et al., 2011). The configuration checking strategy has proven effective to solve problems in graph theory and SAT problems (Cai & Su, 2013; Luo et al., 2013, 2015; Wang et al., 2017; Zhang et al., 2021). The main idea of the configuration checking strategy is to prevent a stochastic local search algorithm from revisiting the same scenario by checking whether the states or neighbors' states of a vertex have changed or not. The configuration checking strategy stores search states for vertices, and forbids moving a vertex back to a former state if its neighbors' states do not change. A lot of experiments show that the configuration checking strategy has achieved great success in many combinatorial optimization problems, such as SAT, the minimum vertex cover problem and the maximum clique problem.

Though the configuration checking strategy is used in many local search algorithms in graph theory, the graphs in those problems are usually not complete. For instance, in the minimum vertex cover problem, the configuration is defined for each vertex as the set of states of all its neighboring vertices, where the state of a vertex indicates whether the vertex is included in the candidate solution. For example, given a graph with 4 vertices v_1, v_2, v_3 and v_4, where v_1 has two neighbors v_2 and v_4, suppose v_2 and v_3 are included in the candidate solution (denoted by a vector $[v_1 = 0, v_2 = 1, v_3 = 1, v_4 = 0]$), and then states of v_1's neighbors $[v_2 = 1, v_4 = 0]$ compose its configuration. If v_2 is selected and moved out of the candidate solution, we say v_2 changes its state and the vector changes to $[v_1 = 0, v_2 = 0, v_3 = 1, v_4 = 0]$, so the new configuration of v_1 is $[v_2 = 0, v_4 = 0]$. However, this form of configuration is not suitable for the CPP local search algorithms, because the neighbors of a vertex are all the other vertices if a graph is complete, and according to the configuration definition, the configuration of a vertex is the set of the states of all the other vertices in the graph. In this case, configurations of all vertices change if any move is made. Thus, configuration checking cannot forbid any move and the cycling problem cannot be alleviated. The existing version of the configuration checking strategy cannot work effectively for complete graphs. Therefore, it is desirable to propose a new configuration checking strategy that can deal with complete graphs for the CPP.

3.2 Timestamp-based Configuration Checking Strategy

We propose a new configuration checking strategy to tackle the cycling problem in local search of the CPP, and discuss how a timestamp can make the strategy be implemented efficiently.

According to the definition of the CPP, a partition specifies each vertex to be put into a cluster, so all vertices in a cluster compose a partial search state. We should avoid coming back to a previous partial state because reducing cycles in local search may lead to reducing cycles on the search state. Suppose a vertex v was moved out of a cluster c at a previous step. If no new vertices have be moved to c since then, moving v back to c is useless as it faces some old vertices and turns back to a previous partial state that has been visited. In that case, a vertex cannot be moved back to a cluster if no new vertices are added to the cluster. So, naturally, the configuration of a vertex *w.r.t.* a cluster should be all the other vertices in the cluster at the moment of the vertex moving out of the cluster.

With the above discussion, we introduce the following notions to define a basic version of the configuration of a vertex *w.r.t.* a cluster, and later we will discuss how to define a timestamp-based version that can be implemented efficiently.

Definition 1 *Given an undirected graph G = (V,E), and a partition* $C = (c_1, c_2, ..., c_k)$*, the configuration of a vertex* $v \in V$ *w.r.t.* c_i*, denoted as* $config(v, c_i)$*, is the set of all vertices in* c_i *excluding* v*.*

We say a configuration changes if a new vertex that was not included in the configuration is added into it, *i.e.,* $config(v, c'_i) \backslash config(v, c_i) \neq \emptyset$, where c'_i is the cluster c_i after moving some vertices.

Definition 2 *The configuration of a vertex* v *w.r.t. a partition* C*, denoted as* $config(v, C)$*, is the set of the configurations of the vertex w.r.t. all clusters.*

With the definitions of configurations, we can check whether a vertex can be moved to another clique before moving the vertex. However, based on the form of configurations defined above, storing configurations and checking the configuration of a vertex v are no easy tasks. We need to compare the current configuration with the configuration at the step of v moving out, and examine all vertices in a cluster in the worst case so as to find out whether any has been moved since v was moved out of this cluster. The time complexity of checking a configuration *w.r.t.* a cluster is $O(|V|)$. Thus, this makes steps in the local search too expensive to perform when solving large-scale instances with thousands of vertices.

To tackle this problem, we introduce timestamps to store changes in history configurations, and then propose a timestamp-based configuration checking strategy to handle the cycling issue in CPP solving. It is noted that when we examine the configuration of a vertex v *w.r.t.* a cluster c, we only care about whether a new vertex has been added into

c compared with the configuration at the moment of v's last moving out. Therefore, we do not need to store all vertices of a cluster and check which one is the newly added. We only need to know whether there is any vertex added into c after v moved out. So a simple way to achieve configuration check is to store when a vertex moves, and when a cluster is updated by adding a vertex, hence configuration changes can be detected by comparing two timestamps.

A local search algorithm improves the solutions iteratively, and makes a move of a vertex at each step. So we can count the number of steps with a positive integer, namely the timestamp of the step. Along this idea, we can define a vector with K timestamps for each vertex, where K is the total number of clusters. A timestamp in the vector is defined as the time that the vertex is moved out of a cluster. It is noted that there may be $K = n$ clusters (each vertex is a cluster) at most, so we need $O(n^2)$ space at worst to store configurations of all vertices. To make configuration checking more efficient, we propose a simple method to record changes. In fact, we can store the timestamps when vertices are moved as well as the timestamps when states of clusters change, and compare timestamps of a vertex and a cluster to check whether a state changes, instead of checking when a vertex is moved out of a certain cluster.

Furthermore, from experimental observation, we can see the following case in simulated annealing: a vertex may perform a decreasing move to another cluster, and at this time the vertex is not stable. It cannot stay in the cluster for a long time, because when we select it again, it will be always moved back to its original cluster through a non-decreasing move since the original cluster is probably a better cluster for the vertex. Therefore, we do not consider a decreasing move as a change of the target cluster. If we consider it as a change, it may weaken the effort on avoiding move cycles, because the target cluster will come to a previous partial state after the unstable vertex is moved back.

We take an example to illustrate the case. Figure 1 (a) provides a partition with 3 clusters. Suppose a vertex v_1 in cluster 1 is moved out of it (suppose it is moved to cluster 3) at the moment t_1, and then another vertex v_2 in cluster 2 performs a decreasing move to cluster 1 at the moment t_2. Afterwards, clusters after changes is shown in Figure 1 (b). At this time, if we consider the movement of v_2 to cluster 1 as a change of cluster 1, then moving v_1 back to cluster 1 is not forbidden by configuration checking. However, v_2 is not stable in cluster 1, as it may return to cluster 2 by a non-decreasing move, so we will revisit the previous state as shown in Figure 1 (c), if v_1 is moved back at the moment t_3 and after that v_2 returns to cluster 2. Hence, it may lead to a search cycle if the configuration checking strategy does not forbid this case.

On the other hand, when we perform the simulated annealing method, we observed that the algorithm sometimes revisits the best solution ever found because the solution is a local optimum. To avoid search cycles around the local optimum, for a cluster c, it is desirable to add a new vertex to c that does not belong to c in the best found solution, rather than to move a vertex originally in c back. Therefore, the timestamp of a cluster c is the last time that a new vertex not belonging to c in the best solution ever found is moved to c. So we define $tscluster(c_i)$ as the moment when c_i changes. It is a timestamp (positive integer) of the last time that a vertex v ($v \notin c_i$ in the partition C_b) is moved to c_i with a non-decreasing objective value, where C_b is the best solution ever found by the algorithm. Furthermore, we define $tsvertex(v)$ as the timestamp when v preforms a non-decreasing move.

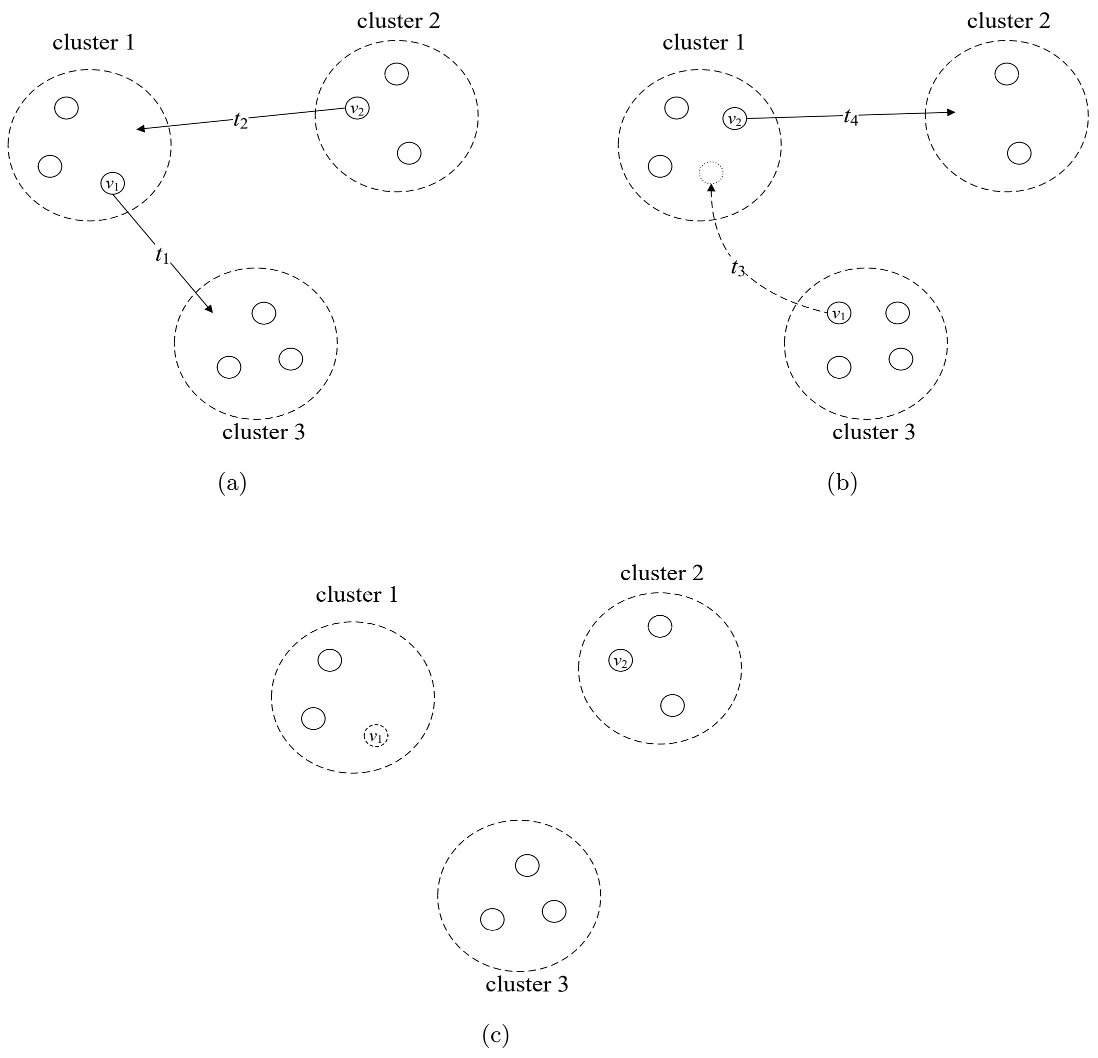


Figure 1: An example of a move cycle

Then, we define a new form of configuration called timestamp-based configuration for a single vertex *w.r.t.* a cluster c_i as follow:

Definition 3 *Given an undirected graph $G = (V,E)$ and a partition C, the timestamp-based configuration of a vertex $v \in V$ w.r.t. $c_i \in C$ is a pair of timestamps ($tsvertex(v)$, $tscluster(c_i)$).*

We say v's configuration *w.r.t.* c_i changes if $tsvertex(v) < tscluster(c_i)$, and configurations for all clusters compose the configuration of v. Hence, to store all configurations of vertices, we should store timestamps when moving vertices, one timestamp for a vertex, and timestamps when clusters change, one for a cluster, so it is clear that we need $O(n)$ space for the entire graph.

Based on the above discussion, we can perform configuration checking using the following rules:

- In the case that $tsvertex(v) > tscluster(c_i)$, the cluster c_i does not change after the step of $tsvertex(v)$, so the move of v to c_i is forbidden.
- In the case that $tsvertex(v) < tscluster(c_i)$, the cluster c_i has changed after the step of $tsvertex(v)$, so the move of v to c_i is allowed.

Given a vertex and a cluster, checking whether the move of the vertex to the cluster is forbidden can be done in $O(1)$ time. Therefore, checking configuration changes is an efficient operation, and the configuration checking strategy can be incorporated into a local search algorithm without much extra time and space.

3.3 Configuration Checking Enhanced Simulated Annealing for CPP

As we discussed in previous sections, some versions of simulated annealing algorithms showed good performance to solve the CPP in previous works, since they can find high-quality solutions efficiently. In this subsection, we improve simulated annealing by incorporating our timestamp-based configuration checking strategy so as to further forbid search cycles, and thus present an improved simulated annealing method.

The framework of simulated annealing used in our algorithm is derived from the version in (Lu et al., 2021). Algorithm 1 depicts the detailed procedure of the proposed simulated annealing.

Before explaining the procedure, we define some functions for describing operations in our simulated annealing algorithm. We denote the cluster that v is currently in by $c(v)$, and define the candidate cluster set as $candi(v) = C \cup \{\emptyset\} \backslash \{c(v)\}$, in which the cluster may be either a cluster in C excluding $c(v)$ or a new empty cluster.

The algorithm starts from the input partition. It initializes the configurations $tsvertex(v)$ with timestamp 0, and $tscluster(c_i)$ with timestamp 1, so all vertices are not forbidden at the beginning. Then, it searches for a better solution iteratively. With the initial temperature, it selects a vertex in each iteration and tries to move it to another cluster. Suppose the algorithm selects the vertex v, then it calls the function $BestCluster()$ to find the best move with the maximum increment after moving v. We will explain the function in detail later. Line 8 computes C' that is the result of function $move(C, v, c_b)$, which is defined as a function that returns the partition by moving v to c_b in C, where $c_b \in candi(v)$ is the target cluster. The new partition C' is then compared with the current one, and the increment of the objective value is calculated, denoted as Δ.

Thereafter, we have two cases to be discussed: Δ is a non-negative number and otherwise. On the one hand, the first case is a non-decreasing move. So, according to the configuration checking principle, after moving the selected vertex, the algorithm will update the timestamp of the moved-to cluster as well as the timestamp of the vertex (line 11). On the other hand, if the move declines the objective value, the standard simulated annealing algorithm allows to make such a move with a probability according to control parameters (*i.e.*, the acceptance probability is calculated with Δ and the current temperature).

After performing the vertex movement, it comes to updating the best solution. If a better solution is found, we apply the descent search method $DescentSearch(C)$, with the aim of further enhancing the solution to find a local optimum (line 14). The descent algorithm is employed under the following consideration. With the fact that simulated annealing accepts

```
Algorithm 1: SA with timestamp-based configuration checking $SA\_CC(C, T_{init})$
   Input: a partition $C$, an initial temperature $T_{init}$
   Output: the resultant partition $C_b$
1  set the temperature as the initial value $T_{init}$;
2  $C_b \leftarrow C$;
3  set timestamp $ts \leftarrow 1$, and initialize configurations by $ts$;
4  while not reaching termination condition do
5      $ts \leftarrow ts + 1$;
6      select a vertex $v$ randomly; $c_b \leftarrow BestCluster(C, v)$;
7      if $c_b = null$ then continue;
8      $C' \leftarrow move(C, v, c_b)$; $\Delta \leftarrow f(C') - f(C)$;
9      if $\Delta \geq 0$ then
10         $C \leftarrow move(C, v, c_b)$;
11         update $tsconfig(v)$ and $tscluster(c_b)$ by $ts$;
12     else
13         with an acceptance probability, $C \leftarrow move(C, v, c_b)$;
14     if $f(C) > f(C_b)$ then $C_b \leftarrow DescentSearch(C)$ ;
15     update the temperature and other control parameters of simulated annealing;
16 return $C_b$;
17 Function $BestCluster(C, v)$
18 $c_b \leftarrow null$; $\Delta_b \leftarrow$ the minimum possible increment;
19 foreach cluster $c'$ in $candi(v)$ do
20     $C' \leftarrow move(C, v, c')$; $\Delta \leftarrow f(C') - f(C)$;
21     if $tscluster(c') < tsvertex(v)$ and $\Delta < 0$ then continue;
22     if $\Delta > \Delta_b$ then
23         $\Delta_b \leftarrow \Delta$; $c_b \leftarrow c'$;
24 return $c_b$;
```

worse moves of vertices, it may escape a local area and miss a local optimum. Therefore, even if the current solution is a new better solution, it may still not be locally optimal. We employ a descent search method to further optimize the best found solution, and accelerate searching for better solutions in the local area around the current best solution.

Finally, the algorithm updates the temperature and control parameters (line 15). The temperature is cooled down after each iteration. We use the same termination condition as used in (Lu et al., 2021), and parameters for controlling the simulated annealing algorithm are set to the same values as MDMCP, since the parameters were well-tuned and MDMCP was proved to work well for the CPP.

The function $BestCluster()$ aims to find the best cluster not forbidden by the configuration checking strategy in $candi(v)$. Given a partition C and a vertex v, $BestCluster$ examines each cluster and finds the best cluster for v. For each cluster c', it computes the increment of moving v to c'. We do not forbid a good move that does not decrease the goal, so if Δ is non-negative, the algorithm compares the increment with Δ_b, and update

Algorithm 2: Function *DescentSearch*(C)

```
    Input: a partition $C$
    Output: the improved partition $C$
 1  $impr \leftarrow true$;
 2  while $impr = ture$ do
 3      $impr \leftarrow false$;
 4      foreach vertex $v$ do
 5          $c_b \leftarrow c(v)$; $f_b \leftarrow f(C)$;
 6          foreach cluster $c'$ in $candi(v)$ do
 7              if $f(move(C, v, c')) > f_b$ then
 8                  $c_b \leftarrow c'$; $f_b \leftarrow f(move(C, v, c'))$;
 9          if $c_b \neq c(v)$ then
10              $C \leftarrow move(C, v, c')$; $impr \leftarrow true$;
11  return $C$;
```

Δ_b if c' is a better cluster (lines 22-23). However, if moving to c' declines the goal, we should check whether the move is forbidden by the configuration checking strategy, so it performs configuration checking and examines whether the timestamp of c' is older than the timestamp of v. If so, it will skip c' and try the next cluster (line 21).

The descent search method is depicted in Algorithm 2. It selects a vertex randomly. After the selection, the method tries to move the vertex to all the other clusters, and the cluster with the largest increment is picked out as the target cluster. If moving the vertex to the cluster can improve the objective value, the algorithm updates the current solution. Selection and move are repeated until there is no vertex move that can improve the objective. The descent search method is another essential difference from the existing simulated annealing algorithms.

4. Framework of the Enhanced Simulated Annealing Algorithm

As discussed previously, a simulated annealing algorithm is a probabilistic technique for approximating the global optimum, but the solution produced by a run of simulated annealing in a fixed amount of time is usually not the global best solution and there probably exist some better solutions the algorithm cannot find. To enhance the ability of search diversity and intensification, we further augment the simulated annealing algorithm with an iterated local search framework, a shrinking mechanism and a restart strategy, and thus present an enhanced simulated annealing algorithm in this section.

Generally speaking, a local search algorithm should have a good ability to find the high-quality solution in a local solution space quickly, and can also search the entire space so as to find the global optimum. To achieve this goal, our algorithm uses an iterated local search framework, where a shrinking mechanism is introduced to speed up convergence. Moreover, the improved simulated annealing method is desirable to exploit a large solution space and aim at finding a high-quality solution, so we use a restart strategy when the local search algorithm gets trapped in a local optimum and the best found solution cannot be updated further. Algorithm 3 depicts the detailed procedure, which is the top level of the iterated simulated annealing algorithm, called SACC.

SACC first initializes a partition C_{res} as an empty set (line 1), then performs search strategies repeatedly until a predefined termination condition is satisfied. We use a simple criterion (*i.e.,* cutoff time) as the termination criterion. In each iteration, it generates a random solution and initializes parameters used in local search. After that, it comes to the iteration step. It will repeat calling simulated annealing until the initial temperature of simulated annealing declines to 0 (line 5). Here, we stipulate that when T_{init} declines below 1% of its first call of simulated annealing, the algorithm can stop the iteration, since with such a low initial temperature simulated annealing cannot accept a decreasing move, and thus it converges quickly. In the inner while loop (lines 5-7), the simulated annealing with configuration checking (the function $SA\_CC(C, T_{init})$) is performed iteratively. If T_{init} declines to a certain value, SACC stops its iteration of simulated annealing and restarts the algorithm. It returns the new solution if it can find a better one or returns the original one if no improvement can be made.

Moreover, we introduce our shrinking mechanism for SA_CC. At beginning, we hope the algorithm explores a large solution space, but as the search goes on, we hope it has a good convergence speed. Thus, after each iteration we shrink the search space gradually in order to accelerate convergence. According to our experimental observation, the parameter T_{init} determines the convergence speed of the simulated annealing strategy. With a smaller initial temperature, a call of simulated annealing will converge more quickly. Based on this consideration, we control the process of simulated annealing through the variants of initial temperature for each SA_CC call. Therefore, after each iteration, we decrease the initial temperature T_{init} by a predefined factor α_{temp} (line 7). As T_{init} decreases, the search space of SA_CC becomes smaller. Hence the entire algorithm can achieve a faster convergence. We will analyze the factor α_{temp} in the experimental part.

Additionally, we employ a restart strategy to begin a new search with a new randomly generated initial solution (line 10). This method can force the algorithm to explore a new solution space if it has been trapped in a local optimum. We will analyze the efforts made by the restart strategy in the experimental part and show detailed results.

5. Computational Experiments

In this section, we performed an extensive computational experiment to evaluate the new algorithm. We carried out computational experiments on a number of benchmark instances, analyzed strategies and parameters carefully, and made an extensive comparison with CPP algorithms recently published in the literature, as well as variants of our algorithm including several versions of configuration checking strategies.

5.1 Experiment Settings

We perform all experiments in parallel on a workstation with two AMD EPYC 7763 CPUs (2.45GHz and 128 processors) and 1024GB RAM, running Ubuntu 20.04.4 LTS. All algorithms were executed 20 times independently for each instance. To study the effectiveness of our algorithm and provide a fair comparison, we implemented our algorithm in C++

Algorithm 3: The framework of iterated local search algorithm SACC

```
Input: a clique partition problem $G = (V, E, W)$
Output: the best partition result $C_{res}$
1  $C_{res} \leftarrow \emptyset$;
2  while not reaching termination condition do
3      $C \leftarrow$ a random initial solution;
4      compute initial temperature, denoted as $T_{init}$;
5      while $T_{init}$ declines to 0 do
6          $C \leftarrow SA\_CC(C, T_{init})$; // Simulated Annealing with Configuration Checking
7          $T_{init} \leftarrow T_{init} \times \alpha_{temp}$;
8      if $C_{res} = \emptyset$ or $f(C) > f(C_{res})$ then
9          $C_{res} \leftarrow C$;
10     reset control parameters and restart search;
11 return $C_{res}$;
```

language, and compiled it with gcc 9.4.0 under the option -O3. There is a parameter to control the speed of convergence in our algorithm. We set it as follow: $\alpha_{temp} = 0.98$.

We will analyze the parameter setting in following experiments. Moreover, in the part of our simulated annealing algorithm we use the same parameter values as MDMCP does.

We take 5 existing algorithms proposed recently to make a careful analysis and comparison. The 5 algorithms are: ITS (Palubeckis et al., 2014), SGVNS (Brimberg et al., 2017), CPP-P^3 (Zhou et al., 2016), TMLS_SA (Hu et al., 2021) and MDMCP (Lu et al., 2021). Hu et al. (2021) and Lu et al. (2021) have shown respectively that their algorithms perform better than CPP-P^3, SGVNS and ITS, but there is no comparison between TMLS_SA and MDMCP. The C++ source codes of comparative algorithms (ITS, CPP-P^3, TMLS_SA and MDMCP) are provided by their authors. Those algorithms were compiled with the same compiler and the same option. The SGVNS algorithm was provided as a binary file by its authors.

We set all algorithms with the same cutoff times as those used in Lu et al. (2021). The cutoff times are set depending on the size of the instances. The more vertices in an instance, the more time is given for searching. The detailed cutoff values are listed in Table 1.

Table 1: Cutoff time of instances in seconds

#vertices	cutoff time(s)
100-300	200
400-500	500
700-800	1000
1000	2000
1500	4000
2000-2500	10000
3000-7000	20000

We consider the set of benchmarks that include all instances tested in the previous 5 algorithms to be compared. There are 94 instances in total with the number of vertices varying from 100 to 7000. In what follows, we give a brief introduction to these benchmarks.

Charon and Hudry (Charon & Hudry, 2006) and Brusco and Kohn (Brusco & Köhn, 2009) provided 13 small instances, where 7 instances (rand100-100, rand300-100, rand500-100, rand300-5, zahn300, sym300-50, and regnier300-50) are from (Charon & Hudry, 2006); and 6 instances (rand200-100, rand400-100, rand100-5, rand200-5, rand400-5, and rand500-5) are from (Brusco & Köhn, 2009). Among the instances, the number of vertices is 500 at most, where the weights of edges are randomly chosen between -5 and 5 (or between -100 and 100) with a uniform distribution.

Palubeckis et al. (2014) generated 20 instances by setting the number of vertices to 500. They also tested 15 large-scale instances that have more than 1000 vertices.

Zhou et al. (2016) provided 5 instances with 500 vertices with edge weights drawn from Gaussian distribution $N(0, 5^2)$. The prefix of the 5 instances is "gauss". Moreover, they provided 10 other instances prefixed by "unif", with 700 or 800 vertices, and the edge weights following the uniform distribution in the range of [-5, 5].

Furthermore, we consider the instances of Unconstrained Binary Quadratic Programming (UBQP) (Kochenberger et al., 2014) in the OR-Lib. The instances were tested by Lu et al. (2021), where the number of vertices is up to 7000.

We divide the 94 instances into two sets, where the small set includes all instances with less than 1000 vertices, and the large set contains the instances with more than or equal to 1000 vertices. All instances can be downloaded from the webpages: https://github.com/hellozhilu/MDMCP and https://leria-info.univ-angers.fr/~jinkao.hao/cpp.html.

For each algorithm and each instance, we report the objective value of the best solution f_{best} and the average objective f_{avg} of 20 runs of the algorithm. We also record the time cost on finding the best solution for each run, and calculate the average time over 20 runs for each instance. In addition, we count the hit time of finding the best solution over 20 runs.

5.2 Comparison with the State-of-the-art Algorithms

In this subsection, we evaluate the effectiveness of our new algorithm. Comparisons are made between our new algorithm and the state-of-the-art algorithms.

First, we analyze the best solutions found by each algorithm and hit times that reach the best solution over 20 runs. Also, we list the best-known solutions found so far, which are reported by Lu et al. (2021), or contributed by our algorithm, as well as other comparative algorithms.

Computational results of 94 instances are divided into two sets. Table 2 and Table 3 demonstrate the detailed results. Table 2 shows the first set of instances (small instances). For the small instances, all algorithms listed in the table are able to find equally good solutions, which are also the best solutions ever found, because these instances are relatively easy to solve. It is also seen that algorithms can achieve 20 hit times for half of those instances. This indicates that most runs can find the solutions equal to the best-known solutions, and there is no significant difference among the comparative algorithms on small instances.

Table 2: Best solutions over 20 runs and hit times of comparative algorithms on small instances

		SACC		MDMCP		TMLS_SA		CPP-P^3		SGVNS		ITS	
instance	best	f_{best}	hit	f_{best}	hit	f_{best}	hit	f_{best}	hit	f_{best}	hit	f_{best}	hit
rand100-5	1407	**1407**	19	**1407**	20	**1407**	20	**1407**	20	**1407**	20	**1407**	20
rand100-100	24296	**24296**	20	**24296**	20	**24296**	20	**24296**	20	**24296**	20	**24296**	20
rand200-5	4079	**4079**	20	**4079**	20	**4079**	13	**4079**	20	**4079**	19	**4079**	20
rand200-100	74924	**74924**	20	**74924**	20	**74924**	19	**74924**	20	**74924**	18	**74924**	20
rand300-5	7732	**7732**	20	**7732**	20	**7732**	8	**7732**	20	**7732**	13	**7732**	18
rand300-100	152709	**152709**	20	**152709**	20	**152709**	20	**152709**	20	**152709**	20	**152709**	20
sym300-50	17592	**17592**	20	**17592**	20	**17592**	7	**17592**	20	**17592**	15	**17592**	20
regnier300-50	32164	**32164**	20	**32164**	20	**32164**	20	**32164**	20	**32164**	20	**32164**	20
zahn300	2504	**2504**	20	**2504**	20	**2504**	10	**2504**	20	**2504**	20	**2504**	20
rand400-5	12133	**12133**	20	**12133**	20	**12133**	11	**12133**	20	**12133**	17	**12133**	19
rand400-100	222757	**222757**	20	**222757**	20	**222757**	19	**222757**	20	**222757**	16	**222757**	17
rand500-5	17127	**17127**	20	**17127**	20	**17127**	20	**17127**	20	**17127**	12	**17127**	11
rand500-100	309125	**309125**	3	**309125**	9	**309125**	9	**309125**	3	**309125**	1	**309125**	1
p500-5-1	17691	**17691**	20	**17691**	20	**17691**	20	**17691**	20	**17691**	2	**17691**	1
p500-5-2	17169	**17169**	20	**17169**	10	**17169**	6	**17169**	19	**17169**	2	**17169**	6
p500-5-3	16816	**16816**	9	**16816**	20	**16816**	2	**16816**	4	16814	1	**16816**	2
p500-5-4	16808	**16808**	20	**16808**	20	**16808**	22	**16808**	20	**16808**	12	**16808**	16
p500-5-5	16957	**16957**	20	**16957**	20	**16957**	20	**16957**	20	**16957**	11	**16957**	17
p500-5-6	16615	**16615**	20	**16615**	20	**16615**	16	**16615**	20	**16615**	9	**16615**	11
p500-5-7	16649	**16649**	20	**16649**	19	**16649**	7	**16649**	19	**16649**	1	**16649**	3
p500-5-8	16756	**16756**	20	**16756**	18	**16756**	20	**16756**	20	**16756**	17	**16756**	14
p500-5-9	16629	**16629**	20	**16629**	20	**16629**	20	**16629**	20	**16629**	7	**16629**	5
p500-5-10	17360	**17360**	20	**17360**	20	**17360**	20	**17360**	20	**17360**	12	**17360**	20
p500-100-1	308896	**308896**	18	**308896**	18	**308896**	19	**308896**	19	**308896**	4	**308896**	11
p500-100-2	310241	**310241**	18	**310241**	8	**310241**	16	**310241**	13	**310241**	2	**310241**	5
p500-100-3	310477	**310477**	20	**310477**	8	**310477**	19	**310477**	20	**310477**	3	**310477**	5
p500-100-4	309567	**309567**	20	**309567**	12	**309567**	9	**309567**	16	**309567**	1	**309567**	1
p500-100-5	309135	**309135**	20	**309135**	20	**309135**	20	**309135**	19	**309135**	9	**309135**	12
p500-100-6	310280	**310280**	20	**310280**	20	**310280**	20	**310280**	19	**310280**	17	**310280**	18
p500-100-7	310063	**310063**	20	**310063**	20	**310063**	14	**310063**	20	**310063**	9	**310063**	19
p500-100-8	303148	**303148**	20	**303148**	20	**303148**	20	**303148**	20	**303148**	12	**303148**	14
p500-100-9	305305	**305305**	20	**305305**	20	**305305**	20	**305305**	20	**305305**	15	**305305**	16
p500-100-10	314864	**314864**	20	**314864**	20	**314864**	20	**314864**	20	**314864**	5	**314864**	18
gauss500-100-1	265070	**265070**	19	**265070**	18	**265070**	20	**265070**	19	**265070**	3	**265070**	3
gauss500-100-2	269076	**269076**	19	**269076**	20	**269076**	16	**269076**	16	**269076**	5	**269076**	4
gauss500-100-3	257700	**257700**	12	**257700**	17	**257700**	5	**257700**	9	**257700**	3	**257700**	1
gauss500-100-4	267683	**267683**	20	**267683**	20	**267683**	18	**267683**	19	**267683**	8	**267683**	11
gauss500-100-5	271567	**271567**	20	**271567**	20	**271567**	20	**271567**	20	**271567**	9	**271567**	16
unif700-100-1	515016	**515016**	20	**515016**	20	**515016**	18	**515016**	15	**515016**	3	514895	2
unif700-100-2	519441	**519441**	20	**519441**	20	**519441**	17	**519441**	20	**519441**	7	**519441**	5
unif700-100-3	512351	**512351**	19	**512351**	8	**512351**	16	**512351**	15	**512351**	10	**512351**	1
unif700-100-4	513582	**513582**	20	**513582**	17	**513582**	20	**513582**	18	**513582**	16	**513582**	1
unif700-100-5	510585	**510585**	8	**510585**	3	**510585**	5	**510585**	1	**510585**	4	510234	3
unif800-100-1	639675	**639675**	20	**639675**	18	**639675**	3	**639675**	7	**639675**	4	639491	1
unif800-100-2	630704	**630704**	20	**630704**	19	**630704**	12	**630704**	4	**630704**	6	630628	1
unif800-100-3	629375	**629375**	1	629108	7	629108	2	629108	6	**629375**	1	628639	1
unif800-100-4	624728	**624728**	7	**624728**	6	**624728**	7	**624728**	9	**624728**	3	623907	1
unif800-100-5	625905	**625905**	16	**625905**	20	**625905**	3	**625905**	3	**625905**	3	625481	1
total/avg		233175.7	868	233170.1	835	233170.1	708	233170.1	792	233175.6	447	233119.2	492

However, as the number of vertices increases, our algorithm SACC has a better performance than others. In Table 3, it is easy to see that our algorithm finds the best solutions for large-scale instances, providing 44 best solutions among all algorithms. Other algorithms can produce better solutions for only a few instances. For example, MDMCP provides 8 best solutions, TMLS_SA provides only 4, and others find fewer best solutions. SGVNS cannot solve instances with more than 2000 vertices due to its memory allocation mechanism, so we mark N/A in the table. It is also worth to mention that our algorithm performs better for all instances with 2500 vertices or more except for b2500.9. Besides, our algorithm performs better than others on hit times, and for each large-scale instance, we can see the number of hits of our algorithm is the highest. So it is clear that our algorithm has a good ability to solve large-scale graphs.

Moreover, our algorithm updates 34 best-known solutions, and all of them are large-scale instances containing no less than 1000 vertices.

In the following, we discuss the average results over 20 runs. Also, we record solution time, *i.e.,* the runtime cost by an algorithm at the last time updating the best found solution. We calculate the average runtimes (in seconds) and list them together with average solutions in Table 4 and Table 5.

Similar to the former tables, there are only small differences in solving the first instance set, since for most instances all algorithms can achieve the same performance. SACC achieves a slightly better average solution in total than others. But results of the second instance set in Table 5 are different. It is clear that our algorithm has the best average value for 42 instances among all algorithms, MDMCP performs best on 4 instances, and others are worse than MDMCP and ours.

Moreover, there are only tiny differences in average runtime among these algorithms. SACC is slightly faster than MDMCP, where the average runtime of SACC is 7388.5s and 7850.7s for MDMCP. Although TMLS_SA has the smallest average runtime in total (i.e., 6581.1s) but its average solution quality is not satisfactory, so TMLS_SA converges too early. Besides, both the runtime and solution quality of ITS are not comparable with others.

Further, we show statistical analysis of computational results. As we mentioned above, SACC and MDMCP do not have a statistical difference on small instances, because most small instances are solved effectively and these algorithms can always find the best-known solutions. Therefore, we only analyze the second instance set (large instances).

We perform Wilcoxon signed-rank test with a significance level of 99%. Table 6 gives the p-value by testing SACC against others. The row “avg” is the results of average solutions over 20 runs, and the row “best” means the best found solutions over 20 runs. Clearly, the table indicates our algorithm is significantly better than others, because the p-values are all smaller than 0.01. Here we do not calculate the p-value against SGVNS because it fails to solve most instances in the second instance set.

Also, we draw box and whisker plots of the results to visualize summary statistics. We calculate the gap between an algorithm’s result and the best result of all algorithms for each instance (the ratio of the difference between the best result and an algorithm’s result to the best result). Figure 2 and Figure 3 depict the gaps, where Figure 2 is the gap between the average solution and the best solution of all algorithms, and Figure 3 is the gap between the best solution produced by an algorithm and the best of all algorithms. From the figures we can see that SACC has the smallest gaps to the best values, and the differences between our algorithm and others are significant. The figures also confirm the results reported in the existing works, which concluded that MDMCP and TMLS_SA are better than other previous algorithms.

Table 3: Best solutions over 20 runs and hit times of comparative algorithms on large-scale instances

		SACC		MDMCP		TMLS_SA		CPP-P^3		SGVNS		ITS	
instance	best	f_{best}	hit	f_{best}	hit	f_{best}	hit	f_{best}	hit	f_{best}	hit	f_{best}	hit
p1000-1	885281	**885281**	5	885016	2	**885281**	1	885016	3	**885281**	1	883885	1
p1000-2	881751	**881751**	14	**881751**	12	**881751**	8	**881751**	1	**881751**	2	879916	2
p1000-3	866488	**866488**	3	**866488**	1	**866488**	1	866441	1	866383	1	864309	2
p1000-4	869374	**869374**	20	**869374**	8	**869374**	16	**869374**	5	**869374**	12	866548	1
p1000-5	888960	**888960**	2	**888960**	1	888950	1	888878	2	888861	1	887727	1
p1500-1	1619470	**1619470**	2	1619421	1	1619271	1	1619281	1	1619016	1	1615258	1
p1500-2	1649778	**1649778**	16	**1649778**	1	1649248	1	1649066	1	1649702	1	1644753	1
p1500-3	1611197	**1611197**	11	1611184	1	1611050	1	1609115	1	1609370	1	1604369	1
p1500-4	1641933	**1641933**	16	**1641933**	9	1641751	1	1641751	1	1641848	2	1637038	1
p1500-5	1595627	**1595627**	20	**1595627**	5	1595472	1	1594013	1	**1595627**	1	1588522	1
p2000-1	2508005	**2508005**	3	2507942	1	2507831	1	2506692	1	2507427	1	2497193	1
p2000-2	2495994	**2495994**	1	2494812	1	2494553	1	2492237	1	2493102	1	2485489	1
p2000-3	2544136	2543724	3	2543679	1	2543731	1	2543195	1	**2544136**	1	2536071	1
p2000-4	2528721	**2528721**	8	2528681	1	2527931	1	2525060	1	2526925	1	2518833	1
p2000-5	2514009	**2514009**	2	2512457	1	2511765	1	2509681	1	2511154	1	2504236	1
b2500.1	1064366	**1064366**	1	1063386	1	1061822	1	1059641	1	N/A	N/A	1054601	1
b2500.2	1064428	**1064428**	2	1064142	1	1062852	1	1060845	1	N/A	N/A	1054727	1
b2500.3	1083209	**1083209**	1	1082193	1	1080746	1	1079814	1	N/A	N/A	1073870	1
b2500.4	1066258	**1066258**	1	1065356	1	1064415	1	1063700	1	N/A	N/A	1059365	1
b2500.5	1066226	**1066226**	1	1065007	1	1064507	1	1062485	1	N/A	N/A	1057916	1
b2500.6	1067531	**1067531**	1	1066629	1	1065825	2	1063660	1	N/A	N/A	1056756	1
b2500.7	1068324	**1068324**	1	1067566	1	1067313	1	1064457	1	N/A	N/A	1057995	1
b2500.8	1070534	**1070534**	1	1070072	1	1069304	1	1068167	1	N/A	N/A	1062615	1
b2500.9	1071460	1071447	1	**1071460**	1	1070082	1	1067336	1	N/A	N/A	1062298	1
b2500.10	1066871	**1066871**	2	1066771	1	1065944	1	1063011	1	N/A	N/A	1057894	1
p3000.1	3259900	**3259900**	6	3258378	1	3256471	1	3252394	1	N/A	N/A	3233145	1
p3000.2	4101652	**4101652**	1	4100246	1	4093370	1	4091478	1	N/A	N/A	4076479	2
p3000.3	4122814	**4122814**	6	4122405	1	4117624	1	4115631	1	N/A	N/A	4090761	1
p3000.4	4588584	**4588584**	7	4588352	1	4585296	1	4580716	1	N/A	N/A	4555927	1
p3000.5	4639266	**4639266**	1	4638953	1	4629046	1	4627606	1	N/A	N/A	4599415	1
p4000.1	5021579	**5021579**	1	5016051	1	5011199	1	4998104	1	N/A	N/A	4970023	1
p4000.2	6381090	**6381090**	2	6377704	1	6366674	1	6353441	1	N/A	N/A	6335155	1
p4000.3	6388024	**6388024**	1	6386642	1	6365419	1	6362378	1	N/A	N/A	6333558	1
p4000.4	7127592	**7127592**	1	7122180	1	7110855	1	7101030	1	N/A	N/A	7055869	1
p4000.5	7048838	**7048838**	1	7048183	1	7042514	1	7011064	1	N/A	N/A	6970690	1
p5000.1	7011355	**7011355**	1	7004710	1	6993503	1	6970941	1	N/A	N/A	6922474	1
p5000.2	8848190	**8848190**	1	8838082	1	8822227	1	8801127	1	N/A	N/A	8742862	1
p5000.3	8978790	**8978790**	1	8975477	1	8946060	1	8923086	1	N/A	N/A	8860487	1
p5000.4	9951747	**9951747**	1	9946401	1	9922806	1	9905517	1	N/A	N/A	9823737	1
p5000.5	9842989	**9842989**	1	9831557	1	9809376	1	9795784	1	N/A	N/A	9705216	1
p6000.1	9217584	**9217584**	1	9207435	1	9185247	1	9169654	1	N/A	N/A	9060203	1
p6000.2	11729985	**11729985**	1	11714145	1	11683518	1	11660212	1	N/A	N/A	11508249	1
p6000.3	13058463	**13058463**	1	13038071	1	13008406	1	12968645	1	N/A	N/A	12848247	1
p7000.1	11638146	**11638146**	1	11615327	1	11595488	1	11575917	1	N/A	N/A	11408433	2
p7000.2	14697515	**14697515**	1	14685148	1	14622821	1	14595665	1	N/A	N/A	14401012	1
p7000.3	16391377	**16391377**	1	16386765	1	16324992	1	16277903	1	N/A	N/A	16004301	1
total/avg		4561630.1	176	4558302.1	77	4549134.1	69	4540793.4	53	N/A	N/A	4502574.5	50

5.3 Analysis on Strategies

In this subsection, we analyze our configuration checking strategy as well as the descent search method. We test several versions of our algorithms with (without) the introduced strategies and a modified configuration checking strategy, aiming at providing an insightful analysis on effects of the strategies.

Table 4: Average behavior of comparative algorithms on small instances

	SACC		MDMCP		TMLS_SA		CPP-P^3		SGVNS		ITS	
instance	f_{avg}	runtime	f_{avg}	runtime	f_{avg}	runtime	f_{avg}	runtime	f_{avg}	runtime	f_{avg}	runtime
rand100-5	1406.4	0.0	**1407.0**	0.2	**1407.0**	0.6	**1407.0**	0.0	**1407.0**	0.2	**1407.0**	0.0
rand100-100	**24296.0**	0.1	**24296.0**	0.5	**24296.0**	11.3	**24296.0**	0.1	**24296.0**	1.6	**24296.0**	0.0
rand200-5	**4079.0**	5.0	**4079.0**	6.9	4078.2	40.8	**4079.0**	2.4	4079.0	23.7	**4079.0**	4.6
rand200-100	**74924.0**	5.9	**74924.0**	6.6	74910.2	47.0	**74924.0**	3.3	74896.3	51.5	**74924.0**	4.6
rand300-5	**7732.0**	14.9	**7732.0**	33.0	7728.1	76.0	**7732.0**	9.8	7730.6	61.9	7731.7	81.8
rand300-100	**152709.0**	0.5	**152709.0**	0.7	**152709.0**	12.9	**152709.0**	1.0	**152709.0**	5.8	**152709.0**	2.8
sym300-50	**17592.0**	38.4	**17592.0**	25.2	17587.2	77.7	**17592.0**	19.7	17590.5	49.4	**17592.0**	61.8
regnier300-50	**32164.0**	0.1	**32164.0**	0.3	**32164.0**	0.1	**32164.0**	0.1	**32164.0**	0.5	**32164.0**	0.2
zahn300	**2504.0**	2.3	**2504.0**	1.7	2503.4	28.5	**2504.0**	1.9	**2504.0**	13.3	**2504.0**	4.8
rand400-5	**12133.0**	30.7	**12133.0**	59.1	12129.5	228.2	**12133.0**	29.9	12132.8	207.4	12132.4	103.3
rand400-100	**222757.0**	65.0	**222757.0**	25.6	222747.0	159.9	**222757.0**	39.5	222716.5	196.5	222734.9	232.7
rand500-5	**17127.0**	45.6	**17127.0**	136.4	**17127.0**	27.0	**17127.0**	108.1	17124.0	112.0	17119.8	271.7
rand500-100	308914.2	38.2	308995.1	153.3	**309010.3**	150.5	308937.9	105.3	308841.6	209.1	308891.8	142.3
p500-5-1	**17691.0**	25.1	**17691.0**	28.8	**17691.0**	80.6	**17691.0**	107.5	17681.5	159.2	17678.4	224.4
p500-5-2	**17169.0**	19.5	17167.5	91.6	17166.9	148.3	17168.9	145.4	17158.3	183.1	17161.3	240.7
p500-5-3	16815.5	133.7	**16816.0**	177.8	16811.6	215.8	16815.1	152.7	16804.0	213.3	16813.3	171.2
p500-5-4	**16808.0**	15.4	**16808.0**	40.3	**16808.0**	29.9	**16808.0**	38.3	16795.7	139.3	16806.0	281.5
p500-5-5	**16957.0**	17.0	**16957.0**	42.2	**16957.0**	40.3	**16957.0**	33.2	16951.8	178.1	16956.4	183.1
p500-5-6	**16615.0**	25.6	**16615.0**	87.7	16614.3	109.5	**16615.0**	52.1	16610.8	250.9	16612.6	200.8
p500-5-7	**16649.0**	78.6	16648.9	105.9	16643.7	179.5	16648.9	127.2	16630.9	187.2	16639.9	285.1
p500-5-8	**16756.0**	29.6	16755.7	157.7	**16756.0**	44.8	**16756.0**	28.7	16752.3	156.8	16752.7	267.7
p500-5-9	**16629.0**	47.1	**16629.0**	119.6	**16629.0**	104.4	**16629.0**	61.8	16617.7	195.6	16620.0	348.4
p500-5-10	**17360.0**	5.2	**17360.0**	4.8	**17360.0**	17.0	**17360.0**	9.2	17359.5	143.7	**17360.0**	125.0
p500-100-1	308895.1	142.9	308895.1	101.1	308895.6	146.0	**308895.7**	160.5	308880.1	136.5	308878.7	146.9
p500-100-2	**310233.2**	185.8	310194.2	99.3	310225.4	98.9	310217.0	187.7	310063.8	153.4	310036.7	267.4
p500-100-3	**310477.0**	39.1	310441.6	102.3	310474.1	129.4	**310477.0**	145.4	310260.9	118.6	310329.8	288.4
p500-100-4	**309567.0**	106.5	309542.0	200.5	309508.2	217.5	309526.8	147.1	309316.7	123.6	309330.5	245.0
p500-100-5	**309135.0**	38.5	**309135.0**	26.6	**309135.0**	40.4	309126.3	150.8	308976.5	126.2	309080.7	251.6
p500-100-6	**310280.0**	33.8	**310280.0**	20.3	**310280.0**	41.2	310260.0	150.7	310227.7	250.9	310235.5	185.5
p500-100-7	**310063.0**	117.2	**310063.0**	145.8	310045.6	130.7	**310063.0**	42.2	310020.1	213.8	310060.1	204.6
p500-100-8	**303148.0**	100.1	**303148.0**	66.1	**303148.0**	71.2	**303148.0**	84.5	302868.2	164.6	302981.4	242.3
p500-100-9	**305305.0**	12.8	**305305.0**	5.6	**305305.0**	10.3	**305305.0**	25.2	305270.4	163.7	305283.3	216.6
p500-100-10	**314864.0**	14.1	**314864.0**	17.0	**314864.0**	49.2	**314864.0**	15.5	314799.5	146.3	314853.2	159.5
gauss500-100-1	265059.9	122.1	265049.8	103.6	**265070.0**	105.7	265059.9	141.5	264841.0	118.4	264853.3	246.9
gauss500-100-2	269067.9	169.8	**269076.0**	91.7	269067.6	135.3	269061.6	110.4	268845.2	194.0	268928.0	242.5
gauss500-100-3	257619.2	221.9	257667.5	141.8	**257691.5**	134.5	257607.9	176.1	257340.9	163.8	257204.6	166.3
gauss500-100-4	**267683.0**	41.2	**267683.0**	17.2	267678.6	137.4	267674.8	83.7	267608.5	139.2	267312.5	193.9
gauss500-100-5	**271567.0**	12.1	**271567.0**	6.5	**271567.0**	98.2	**271567.0**	32.5	271502.7	262.7	271540.0	233.0
unif700-100-1	**515016.0**	142.5	**515016.0**	207.1	514985.3	303.0	514835.5	412.8	514391.5	351.0	513821.2	485.0
unif700-100-2	**519441.0**	152.6	**519441.0**	110.9	519276.1	341.2	**519441.0**	325.0	518395.1	456.8	518458.1	531.6
unif700-100-3	**512310.4**	349.1	511870.1	514.9	512038.5	246.9	512061.2	441.8	511676.4	505.4	510137.9	491.6
unif700-100-4	**513582.0**	125.5	513440.6	455.1	**513582.0**	52.0	513561.4	280.7	513292.3	277.6	511523.8	489.1
unif700-100-5	**510454.2**	341.0	510402.4	236.8	510361.7	441.1	510325.0	554.9	510068.9	405.7	509533.0	579.4
unif800-100-1	**639675.0**	152.9	639672.8	410.6	639371.1	516.2	639616.0	413.8	639175.2	387.0	638474.1	520.1
unif800-100-2	**630704.0**	209.9	630703.9	348.4	630652.3	446.7	630647.8	405.1	630287.6	510.4	629373.2	546.9
unif800-100-3	**629049.1**	459.4	629002.4	621.5	628740.2	484.6	628863.6	420.5	628495.1	401.8	627353.6	513.5
unif800-100-4	624366.2	368.2	624440.0	452.1	**624499.7**	364.2	624434.8	451.4	624171.4	575.5	622700.0	601.0
unif800-100-5	625846.2	289.3	**625905.0**	233.3	625603.1	280.6	625548.6	483.3	625228.4	449.3	624196.0	618.2
avg	233149.9	95.6	233139.0	125.9	233122.9	142.8	233125.0	144.2	232990.8	198.7	232795.1	248.0

Three modified versions of SACC are proposed here by using various combination of strategies:

- SAiter is the version that employs the iteration of simulated annealing, and does not use the configuration checking strategy and the *DescentSearch* method, i.e., SACC excluding the configuration checking strategy and the descent search method.

- SAdesc is the version that performs *DescentSearch* but does not trigger the configuration checking strategy. State in another way, the configuration checking strategy is disabled in the function *BestCluster*, and thus the algorithm will choose the cluster with the maximum increment.

Table 5: Average behavior of comparative algorithms on large-scale instances

	SACC		MDMCP		TMLS_SA		CPP-P^3		SGVNS		ITS	
instance	f_{avg}	runtime	f_{avg}	runtime	f_{avg}	runtime	f_{avg}	runtime	f_{avg}	runtime	f_{avg}	runtime
p1000-1	**885029.5**	721.1	884749.2	801.4	884871.4	766.1	884371.8	1162.2	883910.3	1037.7	881247.7	989.9
p1000-2	**881525.8**	872.7	881480.9	1076.5	880980.5	504.1	880544.2	1045.7	880191.7	990.6	878439.3	1106.6
p1000-3	866227.6	916.3	**866296.2**	1188.8	865866.8	973.1	865420.4	1081.2	865094.5	1058.7	862338.8	916.8
p1000-4	**869374.0**	464.1	868770.2	971.1	869281.1	782.4	868593.1	922.2	868592.3	948.8	864603.5	990.6
p1000-5	**888579.9**	924.2	888508.9	1143.9	888347.5	498.7	888214.0	914.9	888211.2	823.8	885730.0	1120.8
p1500-1	**1619461.9**	1530.4	1619270.3	1840.8	1617088.6	1639.6	1616768.2	1837.7	1615599.2	2446.9	1610283.0	2621.5
p1500-2	**1649333.2**	1539.3	1648963.9	2400.1	1646984.5	1805.9	1646900.7	1989.3	1644636.0	1768.8	1640417.1	2450.9
p1500-3	**1610649.3**	1718.4	1610114.6	2572.7	1607384.6	1797.0	1607018.5	2266.2	1606870.2	2324.0	1599411.2	2221.1
p1500-4	1641724.9	1576.3	**1641854.1**	2188.5	1639167.4	1749.0	1638996.0	2257.2	1637994.2	1716.9	1630857.5	2300.9
p1500-5	**1595627.0**	1309.2	1594744.4	2225.3	1593969.5	1807.4	1592413.7	2347.4	1591863.6	1876.5	1584605.2	2252.9
p2000-1	**2507580.0**	4554.2	2506411.2	7026.0	2503658.6	4906.8	2501897.5	5487.6	2502584.2	5161.7	2490738.0	5636.9
p2000-2	**2494752.5**	4941.9	2494057.1	4565.7	2489496.8	4789.3	2489264.3	5003.3	2488469.9	4746.8	2477793.1	6256.2
p2000-3	**2543164.8**	4569.3	2542872.0	4463.1	2539138.5	3943.9	2538463.2	5915.2	2539021.0	4649.6	2527159.0	5868.6
p2000-4	**2528048.7**	5477.7	2527830.9	5552.3	2522811.8	4577.1	2522707.0	5489.9	2522935.0	5684.2	2512500.2	5810.3
p2000-5	**2512017.8**	4579.1	2510457.3	4876.2	2508388.0	5750.5	2505609.3	5765.8	2508007.0	6140.6	2496996.3	5851.8
b2500.1	**1062794.0**	5848.9	1061252.3	5282.8	1059512.0	4106.8	1057873.1	5453.5	N/A	N/A	1051512.6	6177.7
b2500.2	**1063377.8**	5763.8	1062422.9	6429.2	1060141.5	5170.1	1059020.6	6323.2	N/A	N/A	1052115.2	6925.4
b2500.3	**1082297.1**	5638.5	1081398.3	5744.8	1078579.0	5714.4	1077071.2	6100.5	N/A	N/A	1070640.5	5260.9
b2500.4	**1065600.6**	5552.0	1064954.9	4841.3	1062677.5	4363.5	1061263.6	5392.2	N/A	N/A	1055368.0	5696.4
b2500.5	**1065391.8**	5276.5	1064078.5	5416.8	1062385.8	4168.1	1059844.4	5905.1	N/A	N/A	1053858.1	6589.1
b2500.6	**1066423.0**	5875.5	1065527.1	5761.1	1063387.8	4675.2	1060998.3	6367.8	N/A	N/A	1055080.3	6062.5
b2500.7	**1067665.9**	5829.0	1066829.7	5021.2	1065031.3	5225.7	1063393.7	5986.3	N/A	N/A	1056204.9	5794.1
b2500.8	**1070132.1**	4884.2	1069331.8	5934.6	1066753.4	4528.6	1065768.4	4905.6	N/A	N/A	1057983.6	6463.2
b2500.9	**1070680.9**	6026.6	1070233.3	4073.8	1067694.7	3468.3	1066106.4	6003.0	N/A	N/A	1058917.8	5970.7
b2500.10	**1066642.8**	6269.2	1066198.7	5873.3	1062170.4	4576.9	1060604.2	6662.7	N/A	N/A	1053105.7	6780.5
p3000.1	**3258066.2**	11183.4	3255182.0	12807.0	3249355.1	9767.1	3245084.5	11840.7	N/A	N/A	3224769.1	10887.3
p3000.2	**4100589.8**	7834.2	4097360.2	9271.0	4089067.7	11281.3	4084435.2	12989.9	N/A	N/A	4063505.5	12358.0
p3000.3	**4121662.7**	10118.1	4118719.0	12549.0	4107593.4	9324.4	4104790.5	11256.5	N/A	N/A	4078299.4	13977.0
p3000.4	**4586545.0**	10629.6	4585094.5	11321.7	4572942.2	10953.9	4569794.8	12908.1	N/A	N/A	4539313.9	14110.3
p3000.5	4632508.0	11996.0	**4633187.6**	12553.7	4618095.3	10505.1	4615328.8	12593.8	N/A	N/A	4586673.4	12508.5
p4000.1	**5015852.0**	12193.2	5011392.5	11190.4	4996223.4	9149.6	4988841.6	12144.9	N/A	N/A	4953857.1	14423.1
p4000.2	**6375097.9**	10459.3	6371684.0	11933.8	6353129.4	11243.1	6342676.0	13771.3	N/A	N/A	6301585.4	13316.7
p4000.3	6382777.9	12274.8	**6383783.8**	13528.5	6354013.5	8310.8	6351079.8	15000.3	N/A	N/A	6301181.6	13781.7
p4000.4	**7123288.1**	10903.2	7116888.0	13672.2	7099920.3	9006.3	7088911.9	11429.4	N/A	N/A	7034381.1	15453.2
p4000.5	**7043396.7**	11178.2	7037057.3	12325.8	7018026.2	10249.4	7002004.6	13423.2	N/A	N/A	6948567.1	17632.1
p5000.1	**7005544.5**	10423.0	6998458.6	13932.6	6976279.1	10152.7	6955997.9	12480.3	N/A	N/A	6893235.1	14477.7
p5000.2	**8838727.6**	12713.2	8829379.9	14344.4	8800679.8	9307.2	8783998.3	14357.2	N/A	N/A	8694258.2	16054.0
p5000.3	**8969701.8**	12415.1	8965445.3	13783.6	8929892.9	11598.8	8910271.6	16165.2	N/A	N/A	8824513.8	16180.6
p5000.4	**9944681.4**	11769.9	9933111.9	12401.8	9906679.3	10087.5	9886947.4	12987.6	N/A	N/A	9775197.6	15188.3
p5000.5	**9836746.4**	11898.0	9823943.3	11772.7	9789355.8	10673.6	9775388.4	13935.8	N/A	N/A	9664859.0	15906.7
p6000.1	**9206922.5**	10307.6	9201348.9	10543.6	9164910.2	12163.4	9139816.5	16278.2	N/A	N/A	9024459.5	15073.0
p6000.2	**11719095.2**	12689.4	11706646.8	12360.8	11659342.9	8478.4	11633754.3	16390.6	N/A	N/A	11485554.4	14626.6
p6000.3	**13042774.6**	12847.6	13027470.7	14030.2	12984775.4	12155.0	12943624.1	13369.3	N/A	N/A	12762380.6	16039.8
p7000.1	**11625022.6**	14583.9	11605625.1	12905.6	11563759.6	10738.3	11532784.8	15131.4	N/A	N/A	11350953.1	14583.9
p7000.2	**14683896.6**	15121.2	14671977.6	15121.0	14594817.8	11169.0	14559221.1	16207.6	N/A	N/A	14299194.0	15202.7
p7000.3	**16379753.1**	13676.9	16361097.8	11513.4	16285208.6	14128.6	16239420.1	13211.9	N/A	N/A	15925156.1	14651.8
avg	4557972.9	7388.5	4554205.7	7850.7	4539561.6	6581.1	4531158.6	8488.2	N/A	N/A	4483473.9	9011.9

Table 6: The p-value of Wilcoxon signed-rank test

	MDMCP	TMLS_SA	CPP-P^3	ITS
avg	5.813E-08	3.523E-09	3.523E-09	3.523E-09
best	5.905E-08	1.772E-08	7.616E-09	3.523E-09

- SAdesc&fullcc is the version stipulating that the configuration checking strategy updates timestamps not only after a vertex performs a better move and also after a decreasing move. Therefore, the algorithm does not forbid a vertex to move back to a cluster as long as a new vertex is added to the cluster.

We only list and analyze the experimental results on large-scale instances. Table 7 and 8 list the comparative results, showing the best solutions found and average solutions, respectively. First, from the tables, it can be seen that SAiter can obtain good solutions,

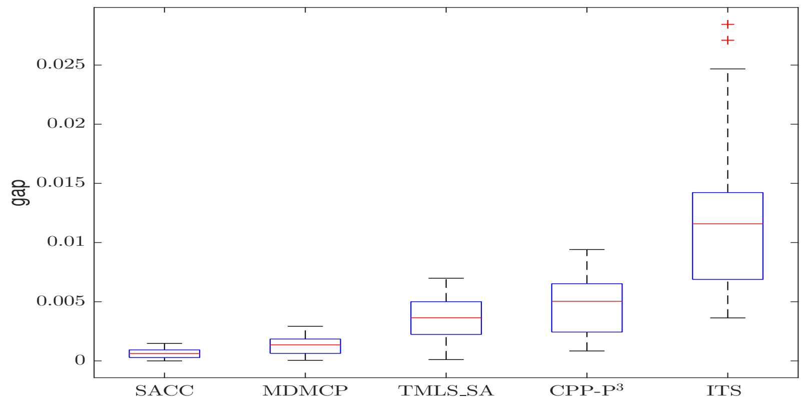


Figure 2: The box plot of gaps between the average solutions found and the best solution of all algorithms.

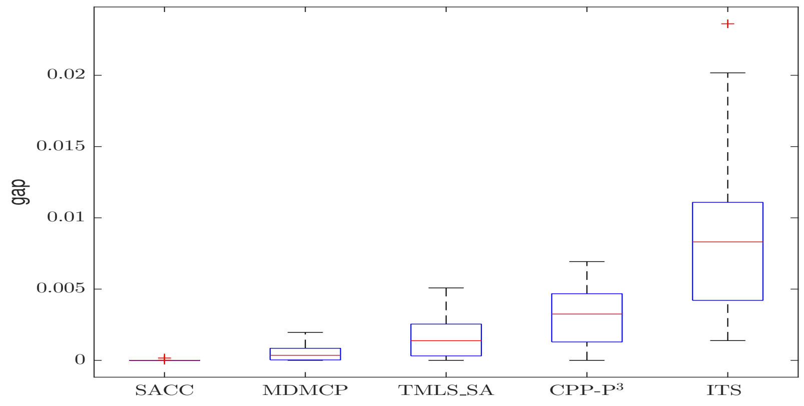


Figure 3: The box plot of gaps between the best solutions produced by an algorithm and the best solution of all algorithms.

and by combining the descent search method the algorithm SAdesc shows a good ability to find better solutions, as it produces more best found solutions over 20 runs compared with SAiter. Moreover, our configuration checking strategy further improves the quality of solutions. It produces the most number of the best found solutions, and more importantly, it can improve average solutions greatly, since we find it performs better on 29 instances compared with SAiter whereas SAiter only has 16 better average solutions. Additionally, we can see the configuration checking strategy has a better performance than the full configuration checking strategy. Though they achieve similar results for the best solutions over 20 runs, SACC has far better average results than SAdesc&fullcc, because SACC has 29 better average solutions and SAdesc&fullcc only has 17 better solutions. As a result, it confirms that configuration checking has essential effects on improving searching behavior in our simulated annealing algorithm. It is noted that the average runtime of finding the best solution and hit time are similar among all versions in comparison here.

Table 7: Comparative analysis of strategies (best solution)

	SACC		SAiter		SAdesc		SAdesc&fullcc	
instance	f_{best}	hit	f_{best}	hit	f_{best}	hit	f_{best}	hit
p1000-1	**885281**	5	**885281**	5	**885281**	6	**885281**	4
p1000-2	**881751**	14	**881751**	12	**881751**	13	**881751**	14
p1000-3	866488	3	866488	2	866488	1	**866587**	1
p1000-4	**869374**	20	**869374**	20	**869374**	19	**869374**	20
p1000-5	**888960**	2	**888960**	1	**888960**	1	**888960**	2
p1500-1	**1619470**	2	**1619470**	4	**1619470**	6	**1619470**	6
p1500-2	**1649778**	16	**1649778**	17	**1649778**	19	**1649778**	20
p1500-3	**1611197**	11	**1611197**	8	**1611197**	11	**1611197**	14
p1500-4	**1641933**	16	**1641933**	13	**1641933**	14	**1641933**	15
p1500-5	**1595627**	20	**1595627**	17	**1595627**	16	**1595627**	19
p2000-1	**2508005**	3	**2508005**	4	**2508005**	1	**2508005**	1
p2000-2	**2495994**	1	2495099	4	2495099	5	2495522	1
p2000-3	2543724	3	**2544728**	1	**2544728**	1	2543724	2
p2000-4	**2528721**	8	**2528721**	9	**2528721**	11	**2528721**	9
p2000-5	**2514009**	2	**2514009**	1	2512730	3	2512730	3
b2500.1	**1064366**	1	1063663	1	1064063	2	**1064366**	1
b2500.2	**1064428**	2	1064267	1	1064270	1	1064176	1
b2500.3	**1083209**	1	**1083209**	1	1082582	1	1082582	1
b2500.4	**1066258**	1	1066124	3	1066148	1	1066138	1
b2500.5	**1066226**	1	1066207	1	1066169	1	1066174	1
b2500.6	**1067531**	1	1067214	1	1067214	1	1067214	1
b2500.7	1068324	1	1068566	1	1068391	3	**1068576**	1
b2500.8	**1070534**	1	1070151	1	**1070534**	1	1070151	3
b2500.9	1071447	1	1071497	2	**1071646**	1	1071411	1
b2500.10	**1066871**	2	1066818	5	**1066871**	3	**1066871**	1
p3000.1	**3259900**	6	**3259900**	3	**3259900**	5	**3259900**	2
p3000.2	4101652	1	4101652	1	**4102907**	1	4102182	1
p3000.3	**4122814**	6	**4122814**	6	**4122814**	5	**4122814**	4
p3000.4	**4588584**	7	**4588584**	5	**4588584**	3	**4588584**	8
p3000.5	**4639266**	1	4639203	1	**4639266**	3	4639203	2
p4000.1	**5021579**	1	5018049	1	5019636	1	**5021579**	1
p4000.2	6381090	2	**6381289**	1	6381090	1	**6381289**	1
p4000.3	6388024	1	6387931	1	6387703	1	**6388075**	1
p4000.4	7127592	1	7128292	1	7130335	1	**7130397**	1
p4000.5	**7048838**	1	**7048838**	2	**7048838**	3	**7048838**	4
p5000.1	**7011355**	1	7011160	1	7010819	1	7010728	1
p5000.2	8848190	1	8847591	1	8845424	1	**8850743**	1
p5000.3	**8978790**	1	8975219	1	8978405	1	8977584	1
p5000.4	9951747	1	**9957492**	1	9954155	1	9953586	1
p5000.5	9842989	1	9845604	1	**9845791**	1	9842523	1
p6000.1	9217584	1	9211365	1	9215077	1	**9218467**	1
p6000.2	11729985	1	11728425	1	11730979	1	**11733007**	1
p6000.3	**13058463**	1	13049086	1	13055387	1	13056965	1
p7000.1	11638146	1	11636377	1	11638414	1	**11638591**	1
p7000.2	14697515	1	14699831	1	14690280	1	**14699943**	1
p7000.3	16391377	1	16403924	1	**16408045**	1	16398054	1
total/avg	4561630.1	176	4561538.3	168	4561758.2	177	4561942.8	179

Table 8: Comparative analysis of strategies (average values)

	SACC		SAiter		SAdesc		SAdesc&fullcc	
instance	f_{avg}	runtime	f_{avg}	runtime	f_{avg}	runtime	f_{avg}	runtime
p1000-1	885029.5	721.1	**885055.9**	818.4	884965.1	782.1	885008.6	692.2
p1000-2	**881525.8**	872.7	881396.9	897.9	881480.3	1002.5	881513.8	1035.4
p1000-3	866227.6	916.3	**866341.9**	1031.1	866332.2	989.8	866226.7	884.8
p1000-4	**869374.0**	464.1	**869374.0**	796.6	869315.8	418.6	**869374.0**	543.7
p1000-5	888579.9	924.2	**888644.5**	1079.1	888626.0	726.6	888590.2	759.2
p1500-1	**1619461.9**	1530.4	1619438.1	1534.9	1619443.9	1408.5	1619448.9	1847.3
p1500-2	1649333.2	1539.3	1649408.0	1126.9	1649669.5	1155.4	**1649778.0**	1125.6
p1500-3	1610649.3	1718.4	1610266.9	1649.9	1610524.6	1647.5	**1610712.5**	1994.8
p1500-4	1641724.9	1576.3	**1641869.3**	2118.3	1641796.7	1511.3	1641865.0	1696.0
p1500-5	**1595627.0**	1309.2	1595449.5	1099.3	1595442.1	1245.3	1595575.2	967.6
p2000-1	**2507580.0**	4554.2	2507249.6	4252.7	2506798.7	3244.8	2507173.3	4848.2
p2000-2	**2494752.5**	4941.9	2494610.7	3729.0	2494673.0	5446.7	2494638.2	5108.2
p2000-3	2543164.8	4569.3	**2543227.5**	5659.5	2543106.8	3748.8	2543223.2	4862.8
p2000-4	2528048.7	5477.7	**2528291.5**	4358.8	2528236.5	5047.8	2528029.5	4167.0
p2000-5	2512017.8	4579.1	**2512122.5**	4788.6	2511870.5	4684.4	2511555.4	4900.5
b2500.1	1062794.0	5848.9	1062549.2	5421.1	1062515.3	5129.4	**1063028.9**	4064.4
b2500.2	**1063377.8**	5763.8	1063138.9	6135.9	1063103.0	5203.4	1063367.8	4932.4
b2500.3	1082297.1	5638.5	**1082363.8**	5213.2	1082226.1	4896.4	1082264.8	6504.4
b2500.4	**1065600.6**	5552.0	1065546.7	5696.7	1065574.8	4494.4	1065575.8	5544.6
b2500.5	**1065391.8**	5276.5	1065368.3	6392.6	1065262.5	5656.9	1065178.1	5243.6
b2500.6	**1066423.0**	5875.5	1066228.5	4701.0	1066315.4	4554.5	1066238.7	5403.1
b2500.7	1067665.9	5829.0	1067590.1	6001.7	**1067789.1**	4447.8	1067610.6	4898.3
b2500.8	**1070132.1**	4884.2	1069864.7	6176.7	1070088.4	4162.9	1069995.2	5801.5
b2500.9	1070680.9	6026.6	**1070885.3**	5460.4	1070885.2	6020.5	1070745.5	4911.9
b2500.10	1066642.8	6269.2	1066350.6	5370.4	1066710.8	4855.3	**1066725.5**	4705.0
p3000.1	3258066.2	11183.4	3257578.8	10780.1	**3258301.8**	9738.9	3257704.6	11949.7
p3000.2	**4100589.8**	7834.2	4100194.4	9594.5	4099885.4	10448.9	4100413.3	10330.7
p3000.3	**4121662.7**	10118.1	4121518.4	10408.9	4121490.1	8718.1	4121658.0	9592.7
p3000.4	4586545.0	10629.6	4585722.7	11987.0	4586788.4	10171.0	**4588239.3**	8315.9
p3000.5	4632508.0	11996.0	4632387.5	9999.0	**4635045.8**	11352.1	4634180.2	10044.8
p4000.1	5015852.0	12193.2	5014831.5	12145.0	5014662.1	10701.4	**5016110.9**	14694.6
p4000.2	6375097.9	10459.3	**6376106.8**	12181.7	6375746.2	13047.0	6376087.5	11352.7
p4000.3	6382777.9	12274.8	6382529.3	11754.3	6383128.0	8323.4	**6384337.0**	10219.7
p4000.4	7123288.1	10903.2	7122700.1	10689.2	**7124126.6**	11907.2	7123640.9	8934.3
p4000.5	7043396.7	11178.2	7043585.1	10176.5	7042997.2	8361.2	**7043758.4**	10454.7
p5000.1	7005544.5	10423.0	**7005712.1**	10131.7	7005114.9	13943.7	7005480.1	11934.0
p5000.2	**8838727.6**	12713.2	8835880.0	11345.6	8838666.3	12922.0	8836915.0	11259.1
p5000.3	8969701.8	12415.1	8966558.3	9486.2	8967739.7	11716.2	**8969847.7**	11109.8
p5000.4	9944681.4	11769.9	**9946227.4**	9546.3	9944565.7	11772.4	9944450.3	11764.1
p5000.5	**9836746.4**	11898.0	9835513.1	13162.7	9835994.0	11695.6	9835772.1	11533.2
p6000.1	9206922.5	10307.6	9203278.2	12758.4	**9207502.4**	14146.5	9207097.3	14443.4
p6000.2	11719095.2	12689.4	**11720868.7**	11560.1	11718643.1	11517.4	11717781.7	13593.7
p6000.3	13042774.6	12847.6	13039451.5	13040.6	**13042969.2**	12337.8	13042665.5	12789.2
p7000.1	**11625022.6**	14583.9	11622482.3	14468.7	11621696.1	13920.6	11622668.0	16010.5
p7000.2	**14683896.6**	15121.2	14682257.6	14211.8	14679237.0	13992.4	14682901.8	14375.5
p7000.3	16379753.1	13676.9	**16381407.0**	12976.0	16381195.3	12923.2	16377952.3	13279.9
total	4557972.9	7388.5	4557596.1	7259.0	4557788.0	7090.0	4557893.6	7291.8

5.4 Parameter Analysis

In the SACC algorithm, there is a parameter α_{temp}. It controls the speed of convergence and determines iteration behavior. In the following, we analyze whether the SACC algorithm is robust under the control parameter. Twelve instances are selected for parameter analysis: rand300-5, rand500-100, p500-5-3, gauss500-100-3, unif700-100-2, unif800-100-4, p1000-1, p1500-3, p2000-1, p2000-4, b2500.3, and p4000.1. The control parameter α_{temp} specifies the decreasing speed of initial temperature in each call of simulated annealing, so the following experiments are performed to show whether the control parameter α_{temp} impacts on effectiveness of SACC. We set α_{temp} to 6 values ranging from 1.0 to 0.90 by decrement at 0.2, and run SACC to solve the 12 instances over 20 runs. The accumulated objective value of the 12 instances for each run is calculated. Figure 4 illustrates the box plot. In fact, α_{temp} is not sensitive when it is set to 0.98, 0.96, 0.94, 0.92, 0.90, but without the method of the shrinking mechanism that declines initial temperature each call of simulated annealing, the algorithm cannot get comparable performance, as we can see when we set α_{temp} to 1.0 the results are clearly worse than the results with $\alpha_{temp} = 0.98$. When varying α_{temp} from 0.98 to 0.90, there is a slight change on the results, and we choose 0.98 as it has the best average value (15364344.15), and $\alpha_{temp} = 0.92$ gets the second best value (15364182.6).

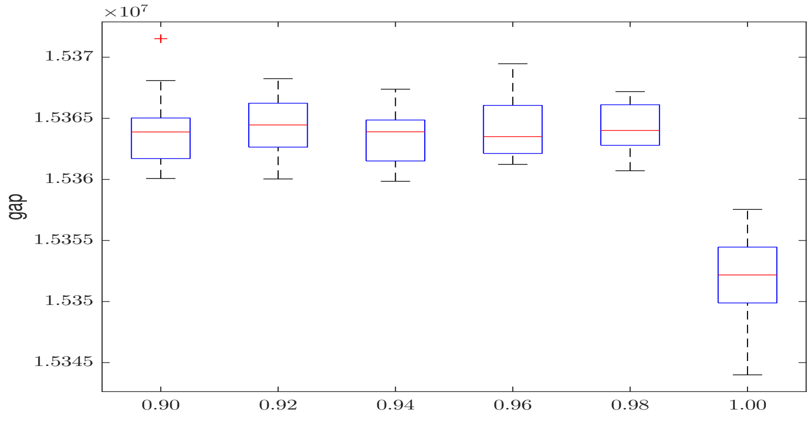
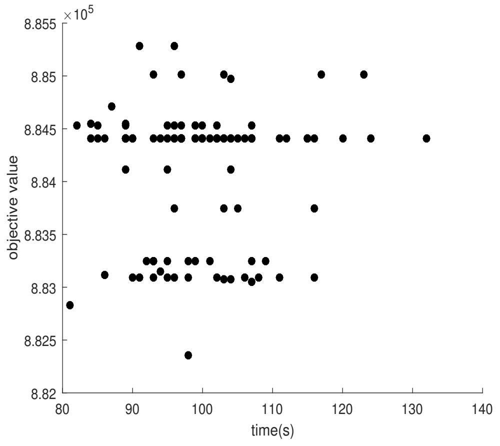


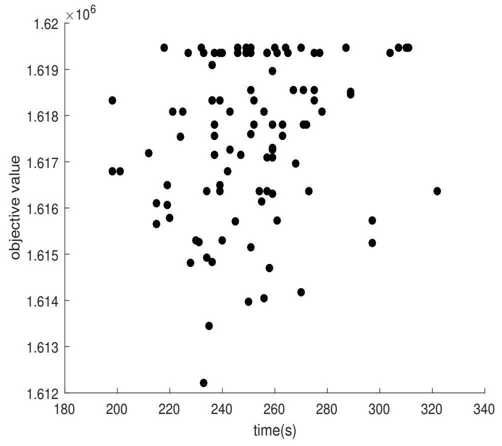
Figure 4: The box plot of the accumulated objective values of 12 instances by varying values of parameter α_{temp}

5.5 Analysis on the Restart Mechanism

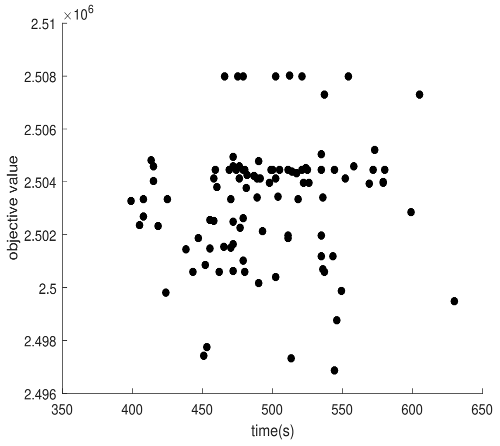
Finally, we analyze the restart mechanism in our algorithm. We selected 4 instances (p1000-1, p1500-1, p2000-1 and p4000.1) to show how often restart is performed and the efforts of the restart mechanism. Here we call the procedure of lines 3-9 in Algorithm 3 a run of iterated simulated annealing. We then count the runtime and the objective value for each run of iterated simulated annealing as two variables, 100 runs for each instance, and draw scatter plots to show the relationship between the variables. The results are illustrated in Figure 5, where the x-axis is the runtime and the y-axis is the objective value. Each point represents a run of iterated simulated annealing without a restart. Four sub-figures show the distributions of the points for the 4 instances, respectively. Clearly, the points are scattered all over the graph, so those sub-figures show that there is no linear positive relation between runtimes and the objective values, and there is no correlation between the two variables. Moreover, without the restart mechanism, the algorithm will converge early. For example, in Figure 5 (a), it takes about 80-130s to complete a run of iterated simulated annealing, so more than 10 times restarts will be performed for SACC when solving p1000-1. However, for the instance p4000.1 (Figure 5 (d)), it needs more time to complete search until the temperature T decreases to 0, since a single run always requires more than 3000s at most. Without the restart mechanism, SACC sometimes cannot get satisfactory results, and thus the restart mechanism is necessary for our algorithm.



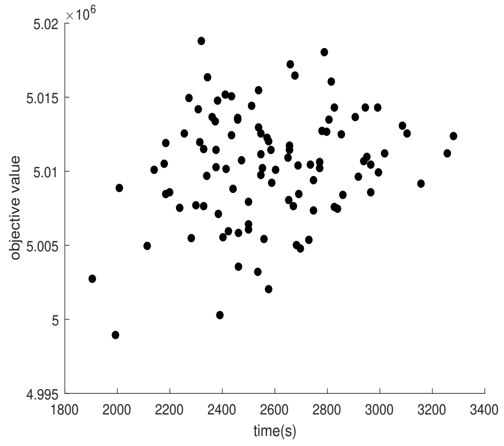
(a) p1000-1



(b) p1500-1



(c) p2000-1



(d) p4000.1

Figure 5: Scatter plots for 4 instances: p1000-1, p1500-1, p2000-1 and p4000.1

Overall, with the above comparative experiments, it is clear that our algorithm has a better performance than all the comparative algorithms, which have been proposed recently. On the other hand, the comparative analysis of strategies employed in our algorithm confirms that our newly proposed strategies are effective, and they can improve the performance of our algorithm.

6. Conclusion

The clique partitioning problem is a classic combinatorial optimization problem and is used to model many practical problems from real-life application, such as clustering, social network analysis and manufacturing system. In the last three decades, many heuristics and local search algorithms, as well as exact algorithms, have been presented to solve this problem. In this paper, we propose an iterated simulated annealing algorithm to solve the clique partitioning problem maximizing the total weights within clusters. In the algorithm, one of the most important components is the simulated annealing algorithm combined with configuration checking. We propose a new configuration check strategy that avoids simulated annealing trapping in search cycling, and thus it enhances the ability of global searching and prevents simulated annealing from revisiting some states that have been visited recently. Furthermore, another component of our algorithm is the descent search method. It is used to search a large neighborhood with the aim of finding a better solution near the solution found by simulated annealing so far, and thus it compensates for the local search ability of simulated annealing, which may miss some better solutions. In addition, we integrate a restart mechanism into our algorithm.

We carried out extensive experiments to evaluate our algorithm, and compared our algorithm with the state-of-the-art algorithms for the CPP. We employ benchmark instances tested by the previous algorithms. There are 94 instances in total with 7000 vertices at maximum. With the same limited time as set in (Lu et al., 2021), each instance is solved 20 times, and the best and average objective value, and runtimes to find the best solution were counted, respectively. From the comparative results, we can conclude that our local search algorithm has the best performance on most instances among all existing algorithms, because it can produce the best solutions for most large-scale instances, and average results of our algorithm are also better than others. In addition, we analyze our strategies in the algorithm. We show that the configuration checking strategy is effective and the descent search method improves solution quality. Finally, we analyze parameter settings and the efforts of the restart strategy. Future works will include studies on combining other meta-heuristics to solve the CPP more efficiently.

Acknowledgment

We would like to thank anonymous reviewers for useful suggestions. We thank Dr. Zhi Lu, Dr. Yi Zhou and Dr. Shuli Hu for their source codes, and Prof. Dragan Urošević for the binary file of SGVNS. The work in this paper was supported by the National Natural Science Foundation of China under Grant Nos. 61972063 and 61972384.

References

Abramé, A., Habet, D., & Toumi, D. (2017). Improving configuration checking for satisfiable random k-SAT instances. *Ann. Math. Artif. Intell.*, *79*(1-3), 5–24.

Alduaiji, N., Datta, A., & Li, J. (2018). Influence propagation model for clique-based community detection in social networks. *IEEE Trans. Comput. Soc. Syst.*, *5*(2), 563–575.

Bilandi, N., Verma, H. K., & Dhir, R. (2021). hPSO-SA: hybrid particle swarm optimization-simulated annealing algorithm for relay node selection in wireless body area networks. *Appl. Intell.*, *51*(3), 1410–1438.

Brimberg, J., Janićijević, S., Mladenović, N., & Urošević, D. (2017). Solving the clique partitioning problem as a maximally diverse grouping problem. *Optimization Letters*, *11*(6), 1123–1135.

Brusco, M. J., & Köhn, H.-F. (2009). Clustering qualitative data based on binary equivalence relations: Neighborhood search heuristics for the clique partitioning problem. *Psychometrika*, *74*(4), 685–703.

Cai, S., & Su, K. (2013). Local search for boolean satisfiability with configuration checking and subscore. *Artif. Intell.*, *204*, 75–98.

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

Charon, I., & Hudry, O. (2006). Noising methods for a clique partitioning problem. *Discret. Appl. Math.*, *154*(5), 754–769.

Chen, P., Wan, H., Cai, S., Li, J., & Chen, H. (2020). Local search with dynamic-threshold configuration checking and incremental neighborhood updating for maximum k-plex problem. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, 2020, New York, NY, USA, February 7-12, 2020*, pp. 2343–2350. AAAI Press.

Chu, Y., Luo, C., Huang, W., You, H., & Fan, D. (2017). Hard neighboring variables based configuration checking in stochastic local search for weighted partial maximum satisfiability. In *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA*, pp. 139–146. IEEE Computer Society.

De Amorim, S. G., Barthélemy, J.-P., & Ribeiro, C. C. (1992). Clustering and clique partitioning: simulated annealing and tabu search approaches. *Journal of Classification*, *9*(1), 17–41.

Dorndorf, U., Jaehn, F., & Pesch, E. (2008). Modelling robust flight-gate scheduling as a clique partitioning problem. *Transportation Science*, *42*(3), 292–301.

Dorndorf, U., & Pesch, E. (1994). Fast clustering algorithms. *INFORMS J. Comput.*, *6*(2), 141–153.

Fox, J., Roughgarden, T., Seshadhri, C., Wei, F., & Wein, N. (2020). Finding cliques in social networks: A new distribution-free model. *SIAM J. Comput.*, *49*(2), 448–464.

Glover, F. W. (1989). Tabu search - part I. *INFORMS J. Comput.*, *1*(3), 190–206.

Glover, F. W. (1990). Tabu search - part II. *INFORMS J. Comput.*, *2*(1), 4–32.

Grötschel, M., & Wakabayashi, Y. (1989). A cutting plane algorithm for a clustering problem. *Math. Program.*, *45*(1-3), 59–96.

Hu, S., Wu, X., Liu, H., Li, R., & Yin, M. (2021). A novel two-model local search algorithm with a self-adaptive parameter for clique partitioning problem. *Neural Comput. Appl.*, *33*(10), 4929–4944.

Hudry, O. (2019). Application of the "descent with mutations" metaheuristic to a clique partitioning problem. *RAIRO Oper. Res.*, *53*(3), 1083–1095.

Jaehn, F., & Pesch, E. (2013). New bounds and constraint propagation techniques for the clique partitioning problem. *Discret. Appl. Math.*, *161*(13-14), 2025–2037.

Ji, X., & Mitchell, J. E. (2007). Branch-and-price-and-cut on the clique partitioning problem with minimum clique size requirement. *Discret. Optim.*, *4*(1), 87–102.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, *220*(4598), 671–680.

Kochenberger, G. A., Hao, J., Glover, F. W., Lewis, M. W., Lü, Z., Wang, H., & Wang, Y. (2014). The unconstrained binary quadratic programming problem: a survey. *J. Comb. Optim.*, *28*(1), 58–81.

Lu, Z., Hao, J., & Wu, Q. (2020). A hybrid evolutionary algorithm for finding low conductance of large graphs. *Future Gener. Comput. Syst.*, *106*, 105–120.

Lu, Z., Hao, J., & Zhou, Y. (2019). Stagnation-aware breakout tabu search for the minimum conductance graph partitioning problem. *Comput. Oper. Res.*, *111*, 43–57.

Lu, Z., Zhou, Y., & Hao, J.-K. (2021). A hybrid evolutionary algorithm for the clique partitioning problem. *IEEE Transactions on Cybernetics*, 1–13.

Luo, C., Cai, S., Wu, W., Jie, Z., & Su, K. (2015). CCLS: an efficient local search algorithm for weighted maximum satisfiability. *IEEE Transactions on Computers*, *64*(7), 1830–1843.

Luo, C., Cai, S., Wu, W., & Su, K. (2013). Focused random walk with configuration checking and break minimum for satisfiability. In *Proceedings of International Conference on Principles and Practice of Constraint Programming*, pp. 481–496.

Matsunaga, T., Yonemori, C., Tomita, E., & Muramatsu, M. (2009). Clique-based data mining for related genes in a biomedical database. *BMC Bioinform.*, *10*.

Michiels, W., Korst, J., & Aarts, E. (2007). *Theoretical Aspects of Local Search*. Springer.

Oosten, M., Rutten, J. H. G. C., & Spieksma, F. C. R. (2001). The clique partitioning problem: Facets and patching facets. *Networks*, *38*(4), 209–226.

Ouyang, G., Dey, D. K., & Zhang, P. (2020). Clique-based method for social network clustering. *J. Classif.*, *37*(1), 254–274.

Palubeckis, G., Ostreika, A., & Tomkevičius, A. (2014). An iterated tabu search approach for the clique partitioning problem. *The Scientific World Journal,2014,(2014-3-4)*, *2014*(2014), 353101.

Skiscim, C. C., & Golden, B. L. (1983). Optimization by simulated annealing: A preliminary computational study for the TSP. In Roberts, S. D., Banks, J., & Schmeiser, B. W. (Eds.), *Proceedings of the 15th conference on Winter simulation, WSC 1983, Arlington, VA, USA*, pp. 523–535. ACM.

Sundar, S., & Singh, A. (2017). Two grouping-based metaheuristics for clique partitioning problem. *Appl. Intell.*, *47*(2), 430–442.

Wakabayashi, Y. (1986). *Aggregation of binary relations: algorithmic and polyhedral investigations.* Ph.D. thesis, University of Augsburg, Germany.

Wang, H., Alidaee, B., Glover, F., & Kochenberger, G. (2006). Solving group technology problems via clique partitioning. *International Journal of Flexible Manufacturing Systems*, *18*(2), 77–97.

Wang, H., Alidaee, B., & Kochenberger, G. A. (2004). Evaluating a clique partitioning problem model for clustering high-dimensional data mining. In *10th Americas Conference on Information Systems, AMCIS 2004, New York, NY, USA, August 6-8, 2004*, p. 234. Association for Information Systems.

Wang, Y., Cai, S., & Yin, M. (2017). Local search for minimum weight dominating set with two-level configuration checking and frequency based scoring function. *J. Artif. Intell. Res.*, *58*, 267–295.

Wang, Y., Pan, S., Al-Shihabi, S., Zhou, J., Yang, N., & Yin, M. (2021). An improved configuration checking-based algorithm for the unicost set covering problem. *Eur. J. Oper. Res.*, *294*(2), 476–491.

Zhang, X., Li, B., Cai, S., & Wang, Y. (2021). Efficient local search based on dynamic connectivity maintenance for minimum connected dominating set. *J. Artif. Intell. Res.*, *71*, 89–119.

Zhou, Y., Hao, J., & Goëffon, A. (2016). A three-phased local search approach for the clique partitioning problem. *J. Comb. Optim.*, *32*(2), 469–491.