A Hybrid Evolutionary Algorithm for the Clique Partitioning Problem

Zhi Lu, Yi Zhou, and Jin-Kao Hao

Accepted to IEEE Transactions on Cybernetics, January 2021

***Abstract*—The Clique Partitioning Problem (CPP) of an edge-weighted complete graph is to partition the vertex set V into k disjoint subsets such that the sum of the edge weights within all cliques induced by the subsets is as large as possible. The problem has a number of practical applications in areas such as data mining, engineering and bio-informatics and is however computationally challenging. To solve this NP-hard problem, we propose the first evolutionary algorithm that combines a dedicated Merge-Divide crossover operator to generate offspring solutions and an effective simulated annealing based local optimization procedure to find high-quality local optima. Extensive experiments on three sets of 94 benchmark instances (including two sets of 63 classical benchmark instances and one new set of 31 large benchmark) show a remarkable performance of the proposed approach compared to the state-of-the-art methods. We analyze the key algorithmic ingredients to shed lights on their impacts on the performance of the algorithm. The algorithm and its available source code can benefit people working on practical problems related to the CPP.**

***Index Terms*—Clique partitioning; Hybrid evolutionary search method; Crossover; Local optimization.**

I. INTRODUCTION

The Clique Partitioning Problem (CPP) [1], [2] is a general model that can naturally formulate many practical applications in various areas including data mining [3], industrial engineering [4], [5], transportation science [6] and bio-informatics [7], [8]. Let $G = (V, E, w)$ be an undirected, complete and edge-weighted graph with a vertex set V, an edge set $E = V \times V$ and an edge weight function $w : E \rightarrow \mathbb{R}$. The CPP involves partitioning V into unrestricted number of disjoint subsets such that the sum of edge weights within the subsets is maximized. The CPP is known to be NP-hard [1], [2] and thus computationally challenging for solution methods. It is worth noting that the name of clique-partitioning is sometimes used to refer to another different problem (the clique cover problem) which is to find the smallest number of cliques in a simple graph such that every vertex in the graph belongs to exactly one clique [9].

For the CPP studied in this work, a number of exact algorithms have been proposed in the last decades. Integer Linear Programming is among the most commonly used approaches for finding optimal solutions. These algorithms are based on various techniques such as cutting planes [1], [2], Lagrangian relaxation [10], branch-and-cut [11] and branch-price-cut [12], [13]. Constraint programming [14] and SAT reasoning [3] were also used to solve the CPP exactly.

However, due to the NP-hardness of the problem, exact algorithms may require a prohibitive time in practice even for problem instances of limited sizes (with up to 500 vertices). To handle large instances, heuristic and metaheuristic algorithms which aim to obtain high quality, but not necessarily optimal solutions were also investigated (For general surveys on metaheuristics, see [15], [16]). For example, De Amorim *et al.* [17] introduced the first tabu search and simulated annealing algorithms using the simple vertex reallocation operator. Dorndorf *et al.* [11] combined ejection chain and Kernighan-Lin heuristic with the same reallocation operator. Charon *et al.* [18], [19] designed the so-called noising method which adds random noises to the candidate solutions being evaluated. Brusco *et al.* [20] integrated the local search algorithms of [17] into a variable neighborhood search algorithm. Palubeckis *et al.* [21] additionally presented an iterated tabu search algorithm (ITS) which repeats a tabu search procedure followed by a random perturbation procedure. This algorithm with fine-tuned parameters reported excellent computational results on a set of benchmark instances. Later, Zhou *et al.* [22] proposed an effective three-phase local search algorithm (CPP-P^3) combining descent search, exploration search and directed perturbation.

In addition to these local search algorithms which solve the problem directly, the problem transformation approach was also investigated, which converts the CPP to another well-studied problem. For example, Kochenberger *et al.* [8] transformed the CPP to the popular Unconstrained Binary Quadratic Programming (UBQP) problem and solved the resulting problem by an existing tabu search algorithm designed for the UBQP. Recently, Brimberg *et al.* [23] recast the CPP to the Maximally Diverse Grouping Problem which was then solved via a variable neighborhood search method (SGVNS).

According to the computational results reported in the above studies, we identify three best performing algorithms: ITS [21], CPP-P^3[22] and SGVNS [23]. However, these studies indicate that existing approaches have difficulties in robustly and consistently producing high-quality solutions for problem instances with more than 500 vertices. On the other hand, to the best of our knowledge, the powerful population-based evolutionary approach has not been investigated for the CPP

Z. Lu and J.K. Hao (corresponding author) are with the Department of Computer Science, LERIA, University of Angers, 2 Boulevard Lavoisier, 49045 Angers 01, France. J.K. Hao is also affiliated with the Institut Universitaire de France. Y. Zhou (corresponding author) is with the School of Computer Science and Engineering, University of Electronic Science and Technology of China, 2006 Xiyuan Avenue, Chengdu 611731, China. (E-mail: zhilusix@gmail.com; zhouyi0922@gmail.com; jin-kao.hao@univ-angers.fr)

yet. In this work, we fill this gap by presenting an effective hybrid evolutionary algorithm called the Merge-Divide Memetic Clique Partitioning Algorithm (MDMCP). The main contributions of this work can be summarized as follows.

First, from the algorithmic perspective, the proposed algorithm is the first evolutionary algorithm with several complementary search components. The dedicated crossover is designed to enable offspring solutions to preserve good properties of parent solutions (based on the notions of vertex "grouping" and "separating"). The simulated annealing based local optimization procedure makes use of a restricted neighborhood that allows the procedure to effectively sample candidate solutions. The pool management relies on both distance and quality information to maintain a healthy population.

Second, from the computational perspective, we report new record results (improved lower bounds) for 12 out of the popular 63 benchmark instances with up to 2000 vertices. We also present for the first time lower bounds on 31 new large benchmark instances with at least 2500 vertices. These new bounds are useful for future studies on the CPP (e.g., as reference values for performance assessment). Moreover, given that the CPP has a number of practical problems in various application domains, the proposed algorithm can be used by researchers and practitioners working in these domains to better solve their problems and the availability of the source code of our algorithm eases such practical applications.

Finally, this study demonstrates the benefit of the population-based approach for solving this challenging problem and the dominance of this approach over all local search and problem transformation approaches. Moreover, the design of the crossover and local optimization procedure of this work could inspire the invention of effective search operators for other partition and grouping problems.

The rest of the paper is organized as follows. Section II introduces necessary notations and definitions. Section III shows the general framework of the proposed algorithm. Sections IV and V introduce two of the most important components of MDMCP, i.e., the dedicated crossover operator and the local optimization procedure. Computational results and analysis are presented in Sections VI and VII, respectively, followed by conclusions in the last section.

II. Notations and definitions

A CPP instance is given by an undirected complete graph $G = (V, E, w)$ where V is the set of vertices, E is the set of edges, and w is the edge weight function $w : E \to \mathbb{R}$.

For the given instance G, a candidate solution is any partition of V into k (k is unknown) non-empty disjoint subsets $\mathcal{P} = \{S_1, ..., S_k\}$ such that $\bigcup_{i=1}^{k} S_i = V$ and for all $1 \leq i \neq j \leq k, S_i \cap S_j = \emptyset$. Then the search space Ω of G includes all possible partitions of V and has a size of $\sum_{j=1}^{n} S(n, j)$ where $S(n, j)$ is the Stirling number of the second kind.

For a candidate solution $\mathcal{P} = \{S_1, ..., S_k\}$, its fitness $f(\mathcal{P})$ is given by the objective value which is the sum of the edge weights within each S_i ($i = 1, 2, \dots, k$), i.e., $f(\mathcal{P}) = \sum_{i=1}^{k} \sum_{u,v \in S_i, u \neq v} w_{uv}$.

To compare two candidate solutions $\mathcal{P}^a$ and $\mathcal{P}^b$, $\mathcal{P}^a$ is better (or worse) than $\mathcal{P}^b$ if $f(\mathcal{P}^a) > f(\mathcal{P}^b)$ (or $f(\mathcal{P}^a) < f(\mathcal{P}^b)$).

To measure the difference between two partitions, we use the following well-known set-theoretic partition distance [24].

Definition 1: Given two partitions $\mathcal{P}^a$ and $\mathcal{P}^b$, the *distance* $Dist(\mathcal{P}^a, \mathcal{P}^b)$ between them is the minimum number of vertices that need to be displaced between subsets of $\mathcal{P}^a$ so that the resulting partition becomes the same as $\mathcal{P}^b$.

Definition 2: Given two partitions $\mathcal{P}^a$ and $\mathcal{P}^b$, if $Dist(\mathcal{P}^a, \mathcal{P}^b) = 0$, then $\mathcal{P}^a$ and $\mathcal{P}^b$ are two *identical* partitions. Otherwise, they are *different* partitions.

$Dist(\mathcal{P}^a, \mathcal{P}^b)$ can be calculated in polynomial time [25].

III. The memetic framework

The Merge-Divide Memetic Clique Partitioning Algorithm (MDMCP) is shown in Algorithm 1. MDMCP relies on the canonical memetic algorithm framework [26], [27], which is a powerful tool for solving difficult optimization problems [28], [29], [30], [31], [32]. Generally, starting with an initial population of at most p ($p \geq 2$) individuals, MDMCP performs a number of generations to evolve the population until the given stopping condition (typically a cutoff time limit) is met. During each generation, two parent solutions $\mathcal{P}^a$ and $\mathcal{P}^b$ are randomly selected from the population and then recombined by the dedicated Merge-Divide crossover operator (Section IV) to generate an offspring solution. The offspring solution is then submitted to the Simulated Annealing based local optimization procedure for quality improvement (Section V). Finally, the improved offspring solution is used to update the population according to the adopted updating rule.

Below, we explain how the population is initialized and updated while the crossover operator and the local optimization procedure are presented in two other sections.

Algorithm 1: Main framework of the MDMCP algorithm

```
Input: Problem instance G = (V, E, w), population size p and
       the other parameters of the algorithm
Output: The best solution found P*
1  begin
2      Pop ← PoolInitialize(p)
3      P* ← arg max_{P∈Pop} f(P)
4      while stopping condition is not met do
5          Randomly select two solutions P^a and P^b from Pop
6          P^o ← MDX_Crossover(P^a, P^b) /* Sect. IV */
7          P^o ← SALO_LocalOptimization(P^o) /* Sect. V */
8          if f(P^o) > f(P*) then
9              P* ← P^o
10         Pop ← UpdatePool(Pop, P^o)
11 end
12 return P*
```

a) Population initialization: The initial population Pop is generated as follows. We first create a trivial partition where each of the n vertices forms a subset of the partition. Then we run p times the local optimization procedure of Section V from this trivial partition and fill Pop with the p improved partitions. Given the stochastic nature of local optimization, the set of p improved solutions may be all different from one

another, or may contain some identical solutions in the sense of Definition 2 (Section II) as well. To avoid identical solutions in Pop, we use the distance measure given in Definition 1 to calculate the pairwise distances of the p solutions and eliminate the duplicated solutions. If this reduces Pop to only one solution (which happens when the p solutions are identical), we apply again the local optimization procedure to the initial trivial partition and stop the process when a different solution of high-quality is reached. Therefore, the initial population contains always 2 to p *different* solutions, which are not only of high quality, but also diverse.

b) Population updating strategy: The population is updated each time an offspring solution is generated and further improved. For this, we first calculate the distance between the offspring and each individual in the population. If the offspring is identical to an individual in the population, the offspring is dropped without changing the population. Otherwise, we update the population by the following rule. If the number of solutions in the population is less than p, then the offspring is inserted into the population. Otherwise, we compare the offspring with the worst solution in the population. If the offspring is better than the worst solution, the offspring replaces the worst one; otherwise, the offspring is dropped.

IV. The Merge-Divide Crossover

It is generally recognized that a well-designed crossover operator can play a driving role within a memetic algorithm [33]. A suitable crossover operator should pass pertinent genetic information from parents to offspring solutions. In graph problems, such information is often represented by structural properties shared by parent solutions (e.g. [28], [30], [32], [34]). For the CPP, we introduce the concepts of *grouping edges* and *separating edges* of the parents. We then design the dedicated Merge-Divide crossover (MDX) that generates offspring solutions while exploring such information.

A. Grouping and separating edges

Definition 3: Let $\mathcal{P}^a$ and $\mathcal{P}^b$ be two parent solutions. Let $\{u, v\}$ be an edge. If u and v are in the same subset in both $\mathcal{P}^a$ and $\mathcal{P}^b$ (i.e., there exist two subsets $X \in \mathcal{P}^a$ and $Y \in \mathcal{P}^b$ such that $u, v \in X$, $u, v \in Y$), then $\{u, v\}$ is a *grouping edge* of the parents. If u and v are in two different subsets in both parent solutions (i.e. there exist two different sets $X, Y \in \mathcal{P}^a$ and two different sets $Z, W \in \mathcal{P}^b$ such that $u \in X$, $v \in Y$, and $u \in Z$, $v \in W$), then $\{u, v\}$ is a *separating edge* of the parents. In the other cases, $\{u, v\}$ is an *undetermined gene* of the parents.

Definition 4: Given two parent solutions $\mathcal{P}^a$ and $\mathcal{P}^b$, let $\{u, v\}$ be a grouping edge and $\{x, y\}$ be a separating edge of the parents. We say that a third solution $\mathcal{P}^c$ *inherits* the grouping edge $\{u, v\}$ if u and v are in the same subset in $\mathcal{P}^c$. Likewise, we say that $\mathcal{P}^c$ *inherits* the separating edge $\{x, y\}$ if x and y are in different subsets in $\mathcal{P}^c$.

As they are defined, grouping and separating edges can be considered as basic building blocks. Our crossover operator aims then to identify the grouping and separating edges in the parent solutions and build a promising (partial) offspring solution by *inheriting* some selected grouping and separating edges. To streamline the inheritance process from parents to offspring, MDX applies an original technique that simplifies the input complete graph into an *auxiliary* incomplete graph of smaller size while randomly sampling the set of edges. When this phase finishes, MDX ensures that any *clique cover* of the resulting incomplete graph can directly produce a valid offspring solution that inherit the selected edges. By combining the inheritance process and the clique cover process, MDX has the advantage of favoring the creation of offspring solutions of both high quality and diversity.

We explain below how MDX samples edges of interest and reduces the input graph. Then we show a clique cover heuristic to obtain a high-quality offspring solution from the resulting graph.

B. Edge sampling and the Merge-Divide operations

MDX (see Algorithm 2) iteratively samples edges of an auxiliary edge-weighted graph $G_i = (V_i, E_i, w^i)$ while considering two reference solutions $\mathcal{P}_i^a$ and $\mathcal{P}_i^b$, where i represents the number of iterations. Initially, G_0 is set to G, i.e., $V_0 = V$, $E_0 = E$ and $w^0 = w$, $\mathcal{P}_0^a$ and $\mathcal{P}_0^b$ are the same as $\mathcal{P}^a$ and $\mathcal{P}^b$, respectively. Then, in the ith iteration, MDX randomly samples an edge from G_i, say $\{u, v\}$ and transforms the current graph G_i into a coarsened graph G_{i+1} via a *Merge* or *Divide* operation as follows.

If $\{u, v\}$ is a grouping edge of $\mathcal{P}_i^a$ and $\mathcal{P}_i^b$, then *Merge* collapses $u, v \in V_i$ to form a new coarse vertex $y \in V_{i+1}$ and adds w_{uv}^i to the fitness f. For any other vertex $x \in V_i$ adjacent to both u and v in G_i, the *Merge* operator merges edges $\{u, x\}$ and $\{v, x\}$ into a new edge $\{y, x\}$ in E_{i+1} and sets the edge-weight w_{yx}^{i+1} as $w_{ux}^i + w_{vx}^i$. The remaining vertices in G_i incident to u or v are now adjacent to y in G_{i+1}. Therefore, after the Merge operation, the fitness equals the sum of the merged edge-weight of the initial graph G_0 (which is G). Meanwhile, to make sure that $\mathcal{P}_i^a$ and $\mathcal{P}_i^b$ are still a partition of V_i, u and v are replaced by y in $\mathcal{P}_i^a$ and $\mathcal{P}_i^b$.

If $\{u, v\}$ is a separating edge, the *Divide* operator is applied, which just removes the edge $\{u, v\}$ from G_i. If $\{u, v\}$ is an undetermined edge, MDX just ignores this edge and continues its sampling process.

The Merge and Divide operations repeat until either G_i becomes sufficiently small, i.e., the number of vertices in G_i is not larger than $n * \eta$ (η is a predefined parameter in $(0, 1]$) or there are no more grouping or separating edges.

C. Offspring generation with clique cover

Upon the termination of the Merge and Divide operations with auxiliary graph $G_i = (V_i, E_i, w^i)$, we are ready to create the offspring solution by finding a *clique cover* of G_i. Recall first that a clique cover of an undirected graph is a partition of its vertex set such that the vertices in each subset are pairwise adjacent, that is, each subset is a clique. In our case, we use a simple heuristic (GreedyCC, see Algorithm 3) to obtain a clique cover $CC = \{S_1, S_2, \ldots, S_k\}$. To identify a clique S_j ($j = 1, \ldots, k$), GreedyCC starts with a random seeding vertex

Algorithm 2: Merge-Divide crossover (MDX)

Input: Graph instance $G = (V, E, w)$, parent solutions $\mathcal{P}^a$ and $\mathcal{P}^b$, shrink ratio $\eta \in (0, 1]$
Output: Offspring solution P^c
1 **begin**
2 $G_0 \leftarrow G, \mathcal{P}_0^a \leftarrow \mathcal{P}^a$, $\mathcal{P}_0^b \leftarrow \mathcal{P}^b$
3 $f \leftarrow 0, i \leftarrow 0$
4 **repeat**
5 Randomly sample an edge $\{u, v\}$ from E_i
6 **if** $\{u, v\}$ *is a grouping edge of* $\mathcal{P}_i^a$ *and* $\mathcal{P}_i^b$ **then**
7 $G_{i+1} \leftarrow$ a coarsened graph of G_i by merging u and v into y, update the edges incident to u and v /* Merge operation */
8 $\mathcal{P}_{i+1}^a \leftarrow$ replace u and v by y in $\mathcal{P}_i^a$
9 $\mathcal{P}_{i+1}^b \leftarrow$ replace u and v by y in $\mathcal{P}_i^b$
10 $f \leftarrow f + w_{uv}^i$
11 **else if** $\{u, v\}$ *is a separating edge of* $\mathcal{P}_i^a$ *and* $\mathcal{P}_i^b$ **then**
12 $G_{i+1} \leftarrow$ a simplified graph of G_i by removing the edge $\{u, v\}$ /* Divide operation */
13 **else**
14 $G_{i+1} \leftarrow G_i, \mathcal{P}_{i+1}^a \leftarrow \mathcal{P}_i^a, \mathcal{P}_{i+1}^b \leftarrow \mathcal{P}_i^b$
15 $i \leftarrow i + 1$
16 **until** $(i > n * \eta)$ *OR* (*neither grouping nor separating edge exists in* E_i) ;
17 $\mathcal{P}_i^c, \Delta \leftarrow$ GreedyCC(G_i)
18 $\mathcal{P}^c \leftarrow$ Replace each vertex of $\mathcal{P}_i^c$ by the aggregated vertices of G_0
19 **end**
20 **return** $\mathcal{P}^c, f + \Delta$

Algorithm 3: The clique cover procedure (GreedyCC)

Input: An undirected graph $G = (V, E, w)$
Output: Clique cover CC and objective value Δ
1 **begin**
2 $k \leftarrow 1$, $\Delta \leftarrow 0$, $CC \leftarrow \emptyset$
3 **repeat**
4 $S_k \leftarrow$ a random vertex in V
5 $B \leftarrow V$
6 **repeat**
7 Randomly take $u \in \{v \in B : \sum_{y \in S_k} w_{vy} > 0\}$
8 $S_k \leftarrow S_k \cup \{u\}$
9 $B \leftarrow (B \setminus \{u\}) \cap N_G(u)$ /* $N_G(u)$ is the set of neighbors of u */
10 $\Delta \leftarrow \Delta + \sum_{y \in S_k} w_{uy}$
11 **until** *there is no vertex* $v \in B$ *such that* $\sum_{y \in S_k} w_{vy} > 0$;
12 $V \leftarrow V \setminus S_k$
13 $CC \leftarrow CC \cup \{S_k\}$
14 $k \leftarrow k + 1$
15 **until** $V = \emptyset$;
16 **end**
17 **return** *clique cover* CC, Δ

and then expands the clique with other vertices that must be adjacent to all vertices of the clique under construction.

From the clique cover $CC = \{S_1, S_2, \ldots, S_k\}$ of G_i, we now unfold the aggregated vertices of each clique S_i to obtain a set of vertices of the initial graph G_0, which forms a subset of the offspring partition. The final offspring partition $\mathcal{P}^c$ is thus composed of k subsets of vertices thus created. The fitness of $\mathcal{P}^c$ is equal to $f + \Delta$, where f is the sum of edge weights of all grouping edges returned by the Merge and Divide operations and Δ is the objective value of the clique cover returned by the clique cover heuristic.

We now verify that the offspring $\mathcal{P}^c$ inherits the required property (grouping and separating edges) of the parent solutions. For this, let us qualify a grouping edge of $\mathcal{P}^a$ and $\mathcal{P}^b$ as *selected* if its two vertices are merged into one coarse vertex in G_i. Likewise, let us qualify a separating edge of $\mathcal{P}^a$ and $\mathcal{P}^b$ as *selected* if its endpoints are merged into different coarse vertices and meanwhile, no edge exists between the two coarse vertices in G_i. Now, if $\{u, v\}$ is a selected grouping edge, then u and v must be merged into one coarse vertex in G_i by the *Merge* operator, and thus must be in the same set in $\mathcal{P}^c$. If $\{x, y\}$ is a selected separating edge, assume that x is merged to x' and y is merged to y', then the *Divide* operator removes the edge between x' and y'. Thus, x' and y' cannot be in the same subset of a clique cover. As a result, x and y must be in different sets in $\mathcal{P}^c$.

Fig. 1 illustrates MDX applied to a graph of five vertices. Relative to the given parents $\mathcal{P}^a$ ($f(\mathcal{P}^a) = 4$) and $\mathcal{P}^b$ ($f(\mathcal{P}^b) = 4$), we have one grouping edge $\{C, D\}$, six separating edges $\{A, C\}$, $\{A, D\}$, $\{A, E\}$, $\{B, E\}$, $\{C, E\}$, $\{D, E\}$, and three undetermined edges $\{A, B\}$, $\{B, C\}$, $\{B, D\}$. Steps 1-5 of Fig. 1 show the Merge and Divide operations applied to four randomly sampled edges (one grouping edge and three separating edges). G_5 is the reduced auxiliary graph when the edge sampling process terminates (with the stopping condition $i > n * \eta$ where $i = 5$, $n = 5$ and $\eta = 0.6$). Steps 6-7 in the figure illustrate the clique cover and the resulting offspring solution $f(\mathcal{P}^c) = 5$.

D. Complexity of MDX

We establish the worst-case time complexity of MDX as follows. Given a graph of n vertices, it requires $O(n)$ time to execute Merge and $O(1)$ time to execute Divide. Since there are at most n merge operations, the worst-case time complexity of MDX (without GreedyCC) is $O(n^2)$. Meanwhile, the runtime of GreedyCC is cubic in terms of the input number of vertices, i.e., $O((\eta n)^3)$. Hence, the overall worst-case time complexity of MDX is $O(n^3)$. On the other hand, the space complexity is $O(n^2)$ since we need to keep an auxiliary graph of the same size as G.

V. Simulated Annealing Based Local Optimization

This section is dedicated to the simulated annealing based local optimization (SALO) component of the MDMCP algorithm. We first introduce the neighborhood used by SALO and then present the general algorithm.

A. Neighborhood

Neighborhood is one key ingredient of any local search algorithm and should be carefully designed to favor the sampling of high-quality candidate solutions of the given problem. For the CPP, one popular neighborhood requires that the distance between a solution $\mathcal{P}$ and a neighbor solution $\mathcal{P}'$ equals one, i.e., $Dist(\mathcal{P}, \mathcal{P}') = 1$ according to Definition 1 in Section II.

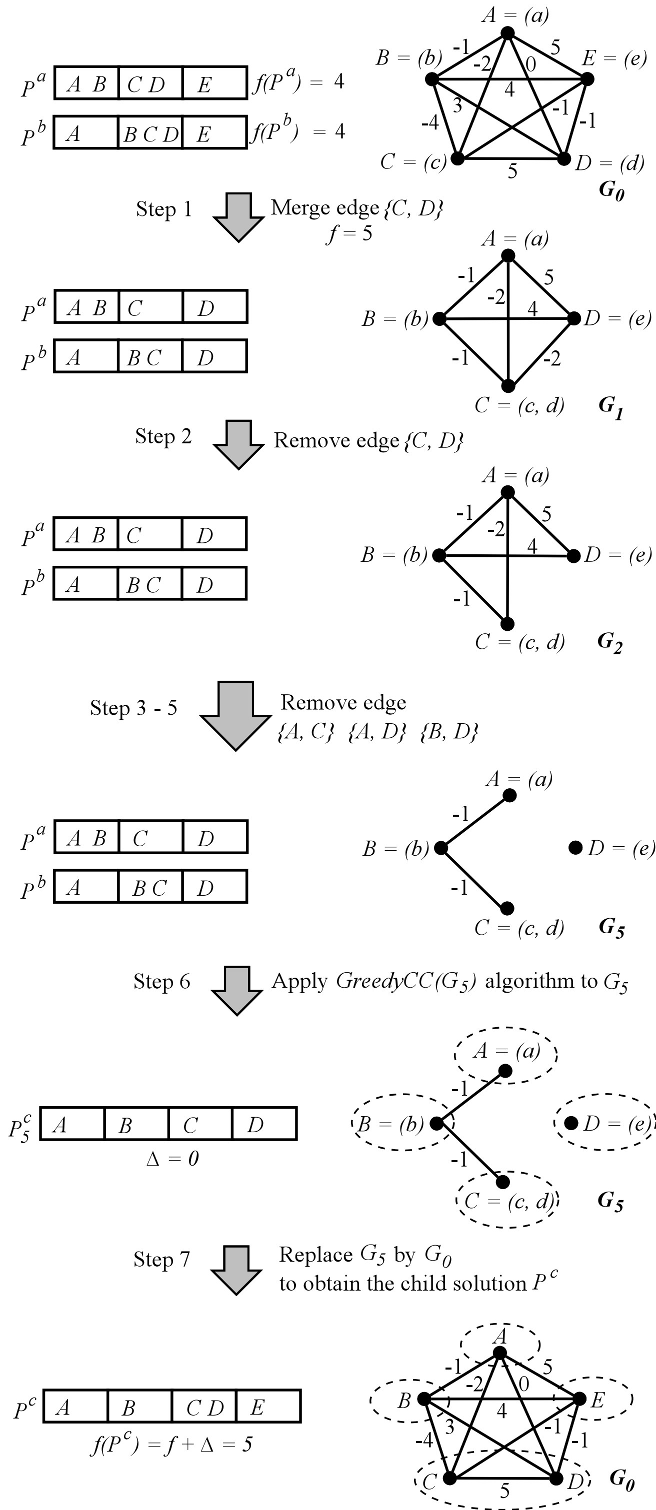


Fig. 1: An illustration of the Merge-Divide crossover (MDX). Given the initial auxiliary graph G_0 and two parent solutions P^a ($f(\mathcal{P}^a) = 4$) and P^b ($f(\mathcal{P}^b) = 4$), the figure shows the steps to generate the offspring solution P^c. Steps 1-5 are the Merge and Divide operations, leading to the objective value $f = 5$; Step 6 applies $GreedyCC(G_5)$ to obtain the partition P_5^c with $\Delta = 0$; Step 7 unfolds graph G_5 to the initial graph G_0 and obtains the offspring solution P^c with the final objective value $f(\mathcal{P}^c) = 5$.

With this neighborhood, a neighbor solution can be built by transferring a random vertex from its current subset to another subset in $\mathcal{P} \cup \{\emptyset\}$. With no restriction on the vertex to be transferred, the algorithm will sample neighbor solutions from a large number of candidate solutions of order $O(n \times k)$.

To improve its search efficiency, our SALO procedure adopts a restricted neighborhood that excludes non-promising candidate solutions. Let $\mathcal{P} = \{S_1, ..., S_k\}$ be the current solution and v a vertex randomly selected in V. Suppose that v belongs to subset S_i in $\mathcal{P}$. We define the best target subset (or simply *best set*) of v to be the subset S^v in $\mathcal{P} \cup \{\emptyset\}$ such that transferring v from S_i to S^v leads to the largest gain of the objective value. Formally, the best set of v is identified by:

$$S^v = \underset{S^v \in \mathcal{P} \cup \{\emptyset\}}{\arg\max} \sum_{u \in S^v} w_{uv} - \sum_{u \in S_i} w_{uv} \quad (1)$$

Given that each vertex has one fixed best set, the size of this restricted neighborhood is bounded by $O(n)$. This neighborhood has the advantages of having a smaller size ($O(n)$ instead of $O(n \times k)$ for the unrestricted neighborhood above) and being more focused (it excludes neighbor solutions with low objective gains).

To generate a neighbor solution, we use the popular REALLOC($\mathcal{P}, v, S^v$) operator, which returns a neighbor $\mathcal{P}'$ of the current solution $\mathcal{P}$ by moving a random vertex v from its current subset S_i to its best set S^v. If S^v is the empty set, REALLOC($\mathcal{P}, v, S^v$) removes v from $\mathcal{P}$ and returns $\mathcal{P} \cup \{\{v\}\}$. Note that, by definition, a solution only consists of non-empty sets, so if S_i becomes empty when v is removed, REALLOC($\mathcal{P}, v, S'$) also deletes this empty set from $\mathcal{P}'$.

As we show in Section VII-A, the SALO procedure using this neighborhood performs very well.

B. General algorithm

The SALO procedure follows the general simulated annealing framework and uses the above neighborhood to effectively explore candidate solutions. As shown in Algorithm 4, SALO performs a number of search rounds (lines 2-13) with different temperature values T (initially set to T_{init}). During each search round, SALO samples $n * k * \theta_{size}$ candidate solutions from the neighborhood to make transitions from the current solution $\mathcal{P}$ to neighbor solutions $\mathcal{P}'$. To generate a neighbor solution, a vertex v is randomly chosen and moved from its current subset to a specially identified target subset (see Section V-A) (lines 5-7). Then a decision is made to decide whether the neighbor $\mathcal{P}'$ is accepted as the new current solution according to the following acceptance probability:

$$Pr\{\mathcal{P} \leftarrow \mathcal{P}'\} = \min(1, e^{\frac{f(\mathcal{P}') - f(\mathcal{P})}{T}}) \quad (2)$$

If the acceptance probability is verified, the neighbor solution $\mathcal{P}'$ becomes the new current solution $\mathcal{P}$; otherwise, the sampled neighbor solution is ignored without changing the current solution during the current iteration (line 8). After each solution transition, the best solution found so far $\mathcal{P}^{best}$ is updated each time a better solution is reached (lines 9-11).

When the current search round reaches its end with the temperature T (i.e., $n * k * \theta_{size}$ candidate solutions are

Algorithm 4: Simulated annealing based local optimization (SALO)

Input: Graph instance $G = (V, E, w)$, input solution $\mathcal{P}$, initial temperature T_{init}, controlling parameters θ_{size}, θ_{cool}, θ_{minper}
Output: The best solution found $\mathcal{P}^{best}$

```
1  begin
2      T ← T_init, P^best ← P
3      repeat
4          for Iter = 1 to |V| * k * θ_size do
5              v ← a random vertex from V
6              S^v ← the best set of v in P ∪ {∅}
7              P' ← REALLOC(P, v, S^v)
8              With probability defined in Equation (2), accept
                   solution P' as new current solution P
9              if f(P) > f(P^best) then
10                 P^best ← P
11                 f^best ← f(P)
12         T ← T * θ_cool /* Temperature cooling down */
13     until Acceptance rate is below θ_minper for 5 consecutive
           rounds ;
14 end
15 return P^best
```

sampled), the current temperature is decreased by a constant factor $\theta_{cool} < 1$ (line 12), which initiates the next search round with the lowered new temperature.

One notices that each iteration of the SALO procedure does not necessarily lead to a solution transition. For this reason, we keep, in addition to the iteration counter $Iter$, a move counter $Move$ and define $Move/Iter$ as the solution acceptance rate. Then the whole SALO procedure terminates and returns the best-recorded solution $\mathcal{P}^{best}$ when the acceptance rate remains below a threshold θ_{minper} for 5 consecutive search rounds. Finally, like many simulated annealing algorithms, SALO's performance is sensitive to its initial temperature T_{init}. We use a simple automated binary search to find a suitable T_{init} value in the range $[1, 2000]$ for a given instance. For this, we run SALO with T_{init} set to the middle value in the initial range $[1, 2000]$. If this leads to a solution acceptance rate of 50%, then T_{init} is set definitively to this value. Otherwise, we re-run SALO with the first or second half-sized range according to whether the acceptance rate is lower or higher than 50%. The tuning of the other parameters is discussed in Section VII-B.

C. Complexity of the local optimization procedure

Assume that the current solution $\mathcal{P}$ has k non-empty subsets. Clearly, k is bounded by n, the number of vertices in G. In our implementation, for a vertex v and each set $S_i \in \mathcal{P} \cup \{\emptyset\}$, we create a table δ to maintain the sum of edge-weights between v and all vertices in S_i. With this table, finding the best set of any vertex only requires $O(n)$. Meanwhile, the REALLOC operation updates the table in $O(n)$ if a neighbor solution is accepted. Therefore, one iteration of the SALO procedure takes time $O(n^2)$ while the space complexity is $O(n^2)$. Finally, if table δ is organized by using a heap data structure, the time complexity can be further reduced to $O(n \log n)$ without changing the space complexity.

VI. Computational experiments

A. Experimental setting

The proposed MDMCP algorithm was implemented in C++[1], and complied using GNU g++ 5.4.0 with '-O3' option, and all experiments were carried out on a computer with an AMD Opteron 4184 processor (2.82GHz) with 2GB RAM, running the Linux operating system.

Table I shows the parameter setting of MDMCP that was consistently used in all experiments reported in this section. This setting can also be considered as MDMCP's default setting. We explain the tuning procedure in Section VII-B.

For the comparative assessment, we adopt, as our reference algorithms, the best performing algorithms: CPP-P^3 [22], SGVNS [23], and ITS [21]. As indicated in the introduction section, these algorithms represent the state-of-the-art for solving the CPP and together hold the best-known results for the available benchmark instances. It is worth mentioning that the source codes of these algorithms are available to us, making it possible to perform a meaningful and fair comparative study.

Table II shows the cutoff times with respect to the number of vertices of the benchmark instances. These cutoff times are used by the four compared algorithms. Due to the stochastic nature of the tested algorithms, each instance was solved 20 times independently by each algorithm. To run each reference algorithm, we use their default parameter setting given in the corresponding papers.

B. Benchmark instances

Our computational assessments are based on three sets of 94 benchmark instances with 100 to 7000 vertices.

- **Small Set (38 instances)**. The instances of this set have up to 500 vertices. The edge-weights are generated randomly by different distributions. The first 13 instances were provided by Charon and Hudry in [19] and Brusco and Köhn in [20]. The next 20 instances prefixed by *p500-5* or *p500-100* were originated from [21] while the last 5 instances with the prefix *gauss* were proposed in [22].
- **Medium set (25 instances)**. These medium instances were collected from [21] and [22] and the number of vertices varies in {700, 800, 1000, 1500, 2000}. The edge-weights of the first 10 graphs are uniformly distributed in the range $[-5, 5]$ and the last 15 graphs distributed in range $[-100, 100]$.
- **Large set (31 instances)**. To assess the scalability of the algorithms, we additionally adopt 31 large graphs with at least 2500 vertices. The first 10 graphs (*b2500.1* to *b2500.10*) are recast from the well-known Unconstrained Binary Quadratic Programming (UBQP) instances in the OR-Lib[2]. The remaining instances are from [21].

[1]The code of our algorithm and the benchmark instances used in this paper will be publicly available at: http://www.info.univ-angers.fr/pub/hao/MDMCP.html

[2]An UBQP instance is a symmetric matrix which can be treated as the adjacent matrix of an undirected edge-weighted complete graph for the CPP. The diagonal elements in the matrix are ignored.

TABLE I: Parameter setting.

Parameter	Section	Description	Range	Value
p	§III	size of population	$[5,\ 10,\ \ldots,\ 45,\ 50]$	10
η	§IV	shrink ratio in MDX	$[0.40,\ 0.45,\ \ldots,\ 0.80,\ 0.85]$	0.60
θ_{size}	§V	max iteration coefficient per temperature of SALO	$[1,\ 2,\ \ldots,\ 256,\ 512]$	8
θ_{cool}	§V	cool ratio of SALO	$[0.90,\ 0.91,\ \ldots,\ 0.98,\ 0.99]$	0.96
θ_{minper}	§V	max acceptance rate for determining frozen state of SALO	$[0.5\%,\ 1.0\%,\ \ldots,\ 4.5\%,\ 5.0\%]$	1%

TABLE II: The cutoff times (in seconds) for different number of vertices.

Number of vertices (n)	Cutoff times (s)
100-300	200
400-500	500
700-800	1000
1000	2000
1500	4000
2000-2500	10000
Others	20000

C. Computational results

Tables III, IV and V show the computational results of the proposed MDMCP algorithm along with the results of the reference algorithms (CPP-P^3 [22], SGVNS [23] and ITS [21]) on the three sets of benchmark instances respectively. Note that the results of SGVNS on the large instances are missing in Table V because SGVNS fails to run on these large instances due to memory leaks.

In each table, columns *Instance* and f_{prev} show the name of the input instance and the best-known objective value ever reported in the literature. For each algorithm, we report its best objective value f_{best} among 20 runs, the average objective value f_{avg}, the frequency hit of reaching f_{best} and the average time $t(s)$ in seconds to reach f_{best} over 20 runs. Note that it is not really meaningful to compare the run times of the algorithms if they don't report the same f_{best}. As a result, the timing information is provided only for indicative purposes.

Additionally, in the last two rows, we show the number of instances that the corresponding algorithm produces a better, equal and worse objective value compared to f_{prev} and the p-value from the non-parametric Wilcoxon signed-rank test with a confidence level of 99% applied to the best and average objective values of MDMCP and each reference algorithm. In all tables, the f_{best} entries that are superior or inferior to f_{prev} are marked in bold or italic. An asterisk ($*$) indicates a strictly best f_{best} among the compared algorithms, which also corresponds to a new best-known result.

1) Comparison on small instances (Table III): We observe that MDMCP and CPP-P^3 reach the previous best-known results on all 38 instances whereas SGVNS and ITS fail on 5 and 4 instances, respectively. The statistical tests confirm that MDMCP performs marginally better than CPP-P^3 in terms of the average results. On the other hand, MDMCP dominates the two other reference algorithms (SGVNS and ITS) by yielding better average objective values (confirmed by the p-values) and slightly better best objective values. Finally, MDMCP also performs the best in terms of hitting the best objective values.

2) Comparison on medium instances (Table IV): The results indicate that MDMCP dominates all reference algorithms in terms of best and average objective values. Remarkably, MDMCP improves the best-known objective value for 12 out of the 25 instances (CPP-P^3 and SGVNS also find improved best-known result for 1 and 2 instances respectively). The statistical significance of this comparison between MDMCP and each reference algorithm is confirmed by the small p-values. Moreover, even though all algorithms show a relatively low frequency of reaching their respective best results, MDMCP hits the improved results more often than the reference algorithms. We also checked how often MDMCP can reach the previous best values f_{prev} (not shown here) and observed that MDMCP hits f_{prev} more often while requiring shorter run times than the reference algorithms.

3) Comparison on large instances (Table V): One observes that MDMCP performs remarkably well on these 31 large instances and shows its superiority over the reference algorithms. For all 31 instances, MDMCP reports the best f_{best} and f_{avg} values among the compared algorithms. Even the average results are better than the best results of the reference algorithms. These outcomes indicate that MDMCP scales very well on large instances. The statistical significance of this comparison is confirmed by the small p-values from the Wilcoxon tests. However, all the algorithms show a low frequency of hitting their respective f_{best}, implying that these large instances are particularly difficult and there is still room for further improvement.

4) A global comparison with performance profiles: We create *performance profiles* [35] which provide a global performance evaluation of all algorithms. To compare a set of algorithms $\mathcal{A}$ over a set of problem instances $\mathcal{I}$, we define the performance ratio by $r_{a,i} = f_{a,i}/min\{f_{a,i} : a \in \mathcal{A}, i \in \mathcal{I}\}$. If an algorithm a does not solve a problem instance i, then we simply set $r_{a,i} = +\infty$. Thus, the performance function of an algorithm a is given by $\mathcal{I}_a(\tau) = |\{i \in \mathcal{I}_a : r_{a,i} \leq \tau\}|/|\mathcal{I}_a|$. The value $\mathcal{I}_a(\tau)$ computes the fraction of problem instances that algorithm a can solve with at most τ many times the cost of the best algorithm. $\mathcal{I}_a(1)$ corresponds to the number of instances that algorithm a solved faster than, or as fast as the other algorithms in $\mathcal{A}$ do. The value $\mathcal{I}_a(r_f)$, for a large enough r_f, corresponds to the maximum number of instances that algorithm a solved. The quantities $\mathcal{I}_a(1)$ and $\mathcal{I}_a(r_f)$ are called efficiency and robustness of a respectively. We draw the performance profiles of the compared algorithms by using the software '*perprof-py*' [36] and the results are shown in Fig. 2 (for the first two sets of 63 small and medium instances) and

TABLE III: Computational results on 38 small instances of the proposed MDMCP algorithm, and three reference algorithms CPP-P^3 [22], SGVNS [23], and ITS [21].

Instance	f_{prev}	MDMCP				CPP-P^3 [22]				SGVNS [23]				ITS [21]			
		f_{best}	f_{avg}	hit	$t(s)$	f_{best}	f_{avg}	hit	$t(s)$	f_{best}	f_{avg}	hit	$t(s)$	f_{best}	f_{avg}	hit	$t(s)$
rand100-5	1407	1407	1407.00	20/20	0.70	1407	1407.00	20/20	0.15	1407	1407.00	20/20	0.44	1407	1407.00	20/20	0.05
rand100-100	24296	24296	24296.00	20/20	1.82	24296	24296.00	20/20	1.03	24296	24296.00	20/20	3.24	24296	24296.00	20/20	1.00
rand200-5	4079	4079	4079.00	20/20	26.14	4079	4079.00	20/20	9.56	4079	4078.95	19/20	23.28	4079	4079.00	20/20	23.25
rand200-100	74924	74924	74924.00	20/20	56.62	74924	74924.00	20/20	19.39	74924	74840.90	14/20	56.32	74924	74924.00	20/20	25.80
rand300-5	7732	7732	7732.00	20/20	201.68	7732	7732.00	20/20	45.20	7732	7731.45	17/20	129.83	7732	7730.50	10/20	156.75
rand300-100	152709	152709	152709.00	20/20	2.54	152709	152709.00	20/20	5.23	152709	152709.00	20/20	7.78	152709	152709.00	20/20	17.85
sym300-50	17592	17592	17592.00	20/20	100.54	17592	17592.00	20/20	53.84	17592	17589.40	14/20	149.37	17592	17589.10	15/20	162.55
regnier300-50	32164	32164	32164.00	20/20	2.21	32164	32164.00	20/20	0.83	32164	32164.00	20/20	1.32	32164	32164.00	20/20	1.90
zahn300	2504	2504	2504.00	20/20	7.68	2504	2504.00	20/20	6.58	2504	2502.90	18/20	16.14	2504	2504.00	20/20	29.15
rand400-5	12133	12133	12133.00	20/20	142.52	12133	12133.00	20/20	155.43	12133	12128.85	14/20	214.04	12133	12130.20	15/20	418.50
rand400-100	222757	222757	222757.00	20/20	88.53	222757	222757.00	20/20	198.31	222757	222681.95	13/20	242.06	222757	222678.30	10/20	490.45
rand500-5	17127	17127	17125.45	14/20	451.24	17127	17127.00	20/20	278.94	17127	17117.90	9/20	496.82	17127	17108.05	4/20	581.80
rand500-100	309125	309125	308901.80	2/20	953.10	309125	308920.45	3/20	328.41	309125	308839.40	3/20	808.37	*309007*	308851.55	1/20	516.85
p500-5-1	17691	17691	17691.00	20/20	188.76	17691	17691.00	20/20	350.59	17691	17680.30	4/20	137.74	*17684*	17671.40	1/20	601.70
p500-5-2	17169	17169	17167.65	11/20	476.10	17169	17168.85	19/20	355.19	*17166*	17153.80	3/20	79.13	17169	17152.50	2/20	500.60
p500-5-3	16816	16816	16815.35	9/20	447.93	16816	16815.05	4/20	579.30	*16815*	16803.60	1/20	782.32	16816	16807.35	2/20	536.95
p500-5-4	16808	16808	16808.00	20/20	192.45	16808	16808.00	20/20	155.67	16808	16797.35	7/20	618.47	16808	16790.00	5/20	577.45
p500-5-5	16957	16957	16957.00	20/20	161.17	16957	16957.00	20/20	98.89	16957	16939.85	3/20	335.00	16957	16945.80	10/20	563.15
p500-5-6	16615	16615	16614.75	19/20	289.60	16615	16614.85	19/20	350.22	16615	16604.40	6/20	559.36	16615	16603.90	6/20	444.15
p500-5-7	16649	16649	16648.55	16/20	447.10	16649	16647.30	12/20	351.07	*16639*	16624.15	5/20	396.57	16649	16629.80	1/20	571.30
p500-5-8	16756	16756	16755.55	17/20	326.41	16756	16756.00	20/20	174.15	16756	16744.25	14/20	361.43	16756	16739.75	5/20	547.40
p500-5-9	16629	16629	16628.60	19/20	322.48	16629	16629.00	20/20	350.85	16629	16609.65	2/20	455.10	*16619*	16600.35	1/20	548.50
p500-5-10	17360	17360	17360.00	20/20	25.75	17360	17360.00	20/20	69.87	17360	17347.00	11/20	226.46	17360	17353.75	16/20	470.45
p500-100-1	308896	308896	308892.40	12/20	420.82	308896	308895.65	19/20	319.17	308896	308855.15	5/20	83.45	308896	308832.35	7/20	467.05
p500-100-2	310241	310241	310174.70	3/20	724.81	310241	310217.00	13/20	447.77	310241	310046.15	2/20	329.90	310241	309809.20	2/20	543.60
p500-100-3	310477	310477	310465.20	16/20	550.08	310477	310474.05	19/20	266.29	*310418*	309943.55	4/20	358.81	310477	309883.45	3/20	513.90
p500-100-4	309567	309567	309555.50	16/20	509.23	309567	309412.00	7/20	411.82	*309494*	309311.00	2/20	311.80	*309494*	309163.55	1/20	431.75
p500-100-5	309135	309135	309135.00	20/20	196.45	309135	309106.95	16/20	473.37	309135	308888.75	11/20	454.52	309135	308934.30	2/20	473.85
p500-100-6	310280	310280	310280.00	20/20	83.06	310280	310280.00	20/20	300.59	310280	310210.70	15/20	426.52	310280	309903.65	11/20	454.80
p500-100-7	310063	310063	310057.20	18/20	403.36	310063	310063.00	20/20	186.26	310063	309984.35	6/20	347.18	310063	310013.30	8/20	490.20
p500-100-8	303148	303148	303148.00	20/20	301.58	303148	303114.25	19/20	324.59	303148	302756.05	10/20	380.30	303148	302601.70	6/20	505.50
p500-100-9	305305	305305	305305.00	20/20	35.52	305305	305305.00	20/20	138.16	305305	305056.70	10/20	526.67	305305	305184.45	10/20	500.70
p500-100-10	314864	314864	314864.00	20/20	78.04	314864	314864.00	20/20	52.24	314864	314771.00	4/20	269.08	314864	314815.90	11/20	334.20
gauss500-100-1	265070	265070	265049.80	18/20	429.43	265070	265029.25	16/20	468.48	265070	264788.55	2/20	165.69	265070	264770.15	2/20	567.85
gauss500-100-2	269076	269076	269076.00	20/20	320.45	269076	269028.05	12/20	409.84	269076	268648.10	3/20	338.24	269076	268734.05	4/20	374.80
gauss500-100-3	257700	257700	257590.25	12/20	504.67	257700	257401.00	2/20	598.59	257700	257296.20	4/20	511.24	257700	257029.80	1/20	611.35
gauss500-100-4	267683	267683	267683.00	20/20	114.68	267683	267615.20	16/20	343.28	267683	267549.15	9/20	497.22	267683	266876.15	5/20	446.15
gauss500-100-5	271567	271567	271567.00	20/20	38.68	271567	271567.00	20/20	174.08	271567	271493.00	9/20	379.52	271567	271471.65	7/20	461.85
#Best/Equal/Worst		0/38/0				0/38/0				0/33/5				0/34/4			
p-value						1.00e-00	3.76e-01			6.25e-02	3.65e-07			1.25e-01	1.17e-06		

Fig. 3 (for the third set of 31 large instances).

Fig. 2 and 3 show that MDMCP has an excellent performance, surpassing the reference algorithms in terms of the objective value (f_{best} and f_{avg}). MDMCP reports the highest value of $\mathcal{I}_a(1)$, indicating that MDMCP can quickly find the highest objective values for the tested instances. MDMCP has a good robustness by arriving at $\mathcal{I}_a(r_f)$ first, implying that it can consistently solve all the instances. Moreover, according to Fig. 3, the dominance of MDMCP over the reference algorithms is even more evident on the set of large instances.

5) A time-to-target comparison: We show a *time-to-target (TTT) plots* [37] to compare the running time distributions of all algorithms on representative (hard) instances. A TTT plot is a useful tool to display, on the Y-axis, the probability that an algorithm will find a solution at least as good as a given target value within a given running time shown on the X-axis. It is generated as follows. We perform E_x independent runs of each algorithm per instance. The running time to arrive at a given target objective value is recorded per run. Then, for each instance, the running times are sorted in ascending order. We associate with the i-th sorted running time t_i a probability $p_i = (i-1/2)/E_x$, and plot the points (t_i, p_i), for $i = 1, \ldots, E_x$.

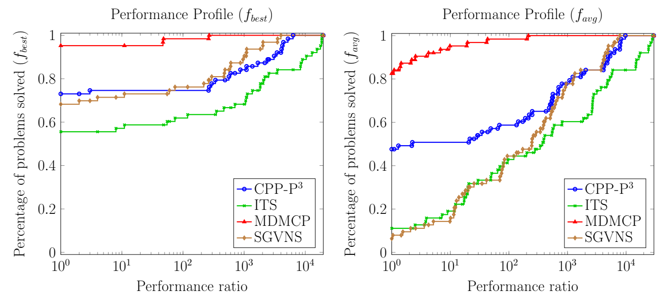


Fig. 2: Performance profiles of the MDMCP algorithm and reference algorithms CPP-P^3 [22], SGVNS [23], and ITS [21] on the sets of 63 small and medium instances.

In our case, we randomly choose 4 hard instances: {*unif800-100-5*, *p1000-5*, *b2500.6*, *p3000.1*}, and 200 independent runs were executed per instance: $E_x = 200$. In order to enable all compared algorithms to get the target objective values in each run, we set the target objective value which is 0.5% smaller than the best objective value found by ITS [21].

Fig. 4 shows the probability distribution plots of three algorithms for the four instances to obtain given target objective

TABLE IV: Computational results on 25 medium instances of the proposed MDMCP algorithm, and three reference algorithms CPP-P[3] [22], SGVNS [23], and ITS [21].

Instance	f_{prev}	MDMCP				CPP-P[3] [22]				SGVNS [23]				ITS [21]			
		f_{best}	f_{avg}	hit	$t(s)$	f_{best}	f_{avg}	hit	$t(s)$	f_{best}	f_{avg}	hit	$t(s)$	f_{best}	f_{avg}	hit	$t(s)$
unif700-100-1	515016	515016	514787.45	10/20	550.73	515016	513984.60	4/20	589.30	515016	513938.95	3/20	538.06	*514550*	512057.90	1/20	635.80
unif700-100-2	519441	519441	519065.40	15/20	344.76	519441	518371.95	7/20	164.75	519441	517829.00	2/20	471.64	*519387*	516371.80	1/20	568.20
unif700-100-3	512351	512351	511206.20	4/20	575.41	512351	510720.20	1/20	503.58	512351	510860.95	4/20	694.50	512351	509332.95	1/20	565.45
unif700-100-4	513582	513582	512826.85	7/20	677.96	513582	513040.60	6/20	541.69	513582	512490.75	8/20	586.79	*512500*	510433.00	1/20	626.30
unif700-100-5	510387	***510585**	510282.20	1/20	395.83	***510585**	510033.50	1/20	741.04	*510367*	509665.45	2/20	409.75	*510234*	508051.80	1/20	664.45
unif800-100-1	639675	639675	639410.95	6/20	423.72	639675	638873.55	1/20	589.78	639675	638895.45	3/20	633.39	*638968*	636748.60	1/20	636.00
unif800-100-2	630704	630704	630570.75	4/20	400.22	*630702*	629865.55	4/20	530.72	630704	629620.80	2/20	633.55	*629581*	627768.65	1/20	535.80
unif800-100-3	629108	629108	628551.30	2/20	776.37	629108	628330.65	2/20	331.94	***629375**	627866.45	1/20	999.96	*628011*	625896.55	1/20	586.50
unif800-100-4	624728	624728	624090.15	1/20	826.25	*624127*	623584.50	1/20	985.17	*624127*	623448.75	2/20	218.19	*622191*	620910.65	1/20	720.45
unif800-100-5	625905	625905	625664.15	6/20	428.21	625905	625101.50	1/20	891.13	*625611*	625273.60	6/20	485.82	*624846*	621927.20	1/20	696.60
p1000-1	885016	*884970*	884403.60	2/20	968.40	885016	883320.35	1/20	337.05	*884411*	882727.50	1/20	1296.32	*882949*	879069.05	1/20	1379.15
p1000-2	881751	881751	880801.55	5/20	1000.54	*880883*	879574.00	1/20	1569.00	881751	879880.85	1/20	1081.22	*879916*	876054.10	1/20	1227.45
p1000-3	866415	**866441**	865869.25	2/20	1454.97	*866150*	864135.85	1/20	1644.96	***866488**	864564.15	1/20	1030.11	*864309*	860142.60	1/20	1207.00
p1000-4	869374	869374	868684.25	9/20	1287.82	*868279*	866636.20	1/20	808.18	*869374*	867847.70	7/20	1002.64	*865866*	862794.10	1/20	1316.30
p1000-5	888720	***888960**	888383.15	1/20	1356.64	*888666*	887833.80	1/20	1252.13	*888474*	887048.80	1/20	213.89	*887121*	883537.30	1/20	1124.85
p1500-1	1618281	***1619362**	1618310.50	4/20	3126.51	*1616930*	1613490.05	1/20	3135.99	**1619101**	1614210.75	1/20	3649.94	*1608949*	1603665.05	1/20	2676.40
p1500-2	1648800	***1649778**	1647891.20	1/20	2248.48	*1645674*	1643714.80	1/20	3203.94	*1647741*	1643426.70	1/20	3256.79	*1641257*	1631271.35	1/20	2688.25
p1500-3	1609854	***1611197**	1608917.85	1/20	3599.09	*1607746*	1603981.65	1/20	1667.45	*1607668*	1604935.75	1/20	3850.99	*1602226*	1593898.55	1/20	2890.60
p1500-4	1640643	***1641933**	1640887.35	3/20	3067.75	**1641370**	1635524.50	1/20	1919.09	**1640842**	1636631.25	1/20	3465.88	*1628976*	1626086.45	1/20	2757.05
p1500-5	1593518	***1595627**	1594238.60	4/20	2897.65	**1593948**	1589983.70	1/20	3212.44	**1594621**	1590461.50	1/20	2914.42	*1587248*	1576895.15	1/20	2947.60
p2000-1	2505359	***2507892**	2504899.15	1/20	9054.82	*2504797*	2498412.60	1/20	9496.83	**2506791**	2497686.20	1/20	4681.56	*2490112*	2478567.95	1/20	7006.05
p2000-2	2492662	***2494840**	2493592.70	1/20	7786.81	*2489622*	2483856.35	1/20	8741.36	**2493883**	2485781.70	1/20	4266.66	*2476233*	2465157.55	1/20	7259.80
p2000-3	2540063	***2544334**	2541430.45	1/20	8492.93	*2538005*	2533434.40	1/20	9593.49	**2543859**	2536439.15	1/20	3367.03	*2528777*	2517154.75	1/20	7380.35
p2000-4	2525903	***2528684**	2526603.55	1/20	7119.96	*2524452*	2518389.35	1/20	6890.70	**2526658**	2520347.60	1/20	5258.30	*2509885*	2498626.35	1/20	6951.25
p2000-5	2508729	***2513199**	2509993.65	1/20	7308.74	**2508806**	2502155.05	1/20	6118.08	**2509198**	2505138.10	1/20	3397.66	*2494600*	2485841.65	1/20	6721.90
#Best/Equal/Worst		13/11/1				4/8/13				10/8/7				0/1/24			
p-value						4.21e-04	1.39e-05			7.13e-04	1.23e-05			1.82e-05	1.23e-05		

TABLE V: Computational results on 31 large instances of the proposed MDMCP algorithm, and two reference algorithms CPP-P[3] [22], and ITS [21].

Instance	MDMCP				CPP-P[3] [22]				ITS [21]			
	f_{best}	f_{avg}	hit	$t(s)$	f_{best}	f_{avg}	hit	$t(s)$	f_{best}	f_{avg}	hit	$t(s)$
b2500.1	***1063447**	1061285.90	1/20	9603.47	*1058161*	1054223.95	1/20	8268.97	*1051544*	1045699.75	1/20	7018.05
b2500.2	***1063517**	1061926.35	1/20	7465.33	*1058488*	1055570.00	1/20	6647.19	*1051339*	1046204.45	1/20	6445.65
b2500.3	***1082275**	1080759.70	1/20	7991.01	*1076937*	1073714.95	1/20	9934.26	*1070093*	1065769.10	1/20	7111.15
b2500.4	***1065977**	1064729.95	1/20	9871.94	*1060114*	1057526.80	1/20	9344.38	*1055566*	1050087.60	1/20	7414.10
b2500.5	***1066387**	1063602.25	1/20	6777.81	*1061872*	1058104.35	1/20	7249.99	*1054226*	1048259.40	1/20	6301.80
b2500.6	***1066847**	1065194.10	1/20	4908.76	*1060603*	1058830.60	1/20	7445.48	*1054109*	1048448.90	1/20	6401.90
b2500.7	***1068161**	1066540.75	1/20	4400.32	*1063245*	1060809.05	1/20	8734.46	*1053798*	1050445.10	1/20	7457.90
b2500.8	***1069934**	1068766.85	1/20	5681.42	*1064928*	1061280.95	1/20	6635.79	*1056671*	1052615.10	1/20	7314.50
b2500.9	***1071272**	1069488.00	1/20	9088.52	*1066143*	1062813.70	1/20	6121.51	*1056994*	1053589.65	1/20	6646.05
b2500.10	***1066735**	1065303.15	1/20	8823.48	*1061429*	1058383.50	1/20	9931.12	*1052154*	1046592.85	1/20	7041.00
p3000.1	***3257061**	3253637.60	1/20	14031.83	*3243562*	3235638.25	1/20	15910.03	*3218903*	3210395.70	1/20	14750.80
p3000.2	***4099540**	4095924.60	1/20	10344.62	*4086433*	4076967.75	1/20	15866.52	*4065638*	4043362.40	1/20	14511.85
p3000.3	***4121651**	4116510.05	1/20	14861.91	*4106083*	4097060.30	1/20	17460.15	*4076558*	4057087.80	1/20	14403.50
p3000.4	***4586819**	4582002.80	1/20	19683.10	*4574395*	4556865.75	1/20	18944.28	*4522639*	4508185.60	1/20	13856.05
p3000.5	***4638416**	4628915.50	1/20	8037.94	*4611832*	4600197.60	1/20	19886.69	*4576868*	4560847.75	1/20	16027.70
p4000.1	***5013660**	5008649.80	1/20	19087.02	*4991071*	4977208.30	1/20	14350.22	*4935821*	4911988.30	1/20	14613.75
p4000.2	***6376823**	6367326.50	1/20	15685.50	*6336160*	6321660.80	1/20	19480.23	*6274852*	6236934.30	1/20	16549.75
p4000.3	***6386325**	6373723.90	1/20	18755.28	*6346142*	6328661.35	1/20	13341.48	*6289871*	6241976.05	1/20	15998.65
p4000.4	***7121377**	7114980.55	1/20	17620.49	*7100738*	7072143.30	1/20	18782.95	*6995198*	6963301.60	1/20	16557.15
p4000.5	***7045223**	7033485.25	1/20	11596.08	*7002534*	6985122.55	1/20	15541.30	*6924546*	6865919.65	1/20	15003.10
p5000.1	***7009948**	6990929.85	1/20	17047.25	*6949681*	6930495.40	1/20	18712.82	*6846645*	6812578.45	1/20	16920.20
p5000.2	***8827548**	8817131.05	1/20	17491.37	*8776994*	8747892.55	1/20	17078.41	*8644244*	8582114.20	1/20	16228.45
p5000.3	***8965743**	8949408.10	1/20	19998.82	*8895249*	8872251.55	1/20	19979.24	*8767725*	8702682.85	1/20	15863.65
p5000.4	***9936590**	9925708.30	1/20	13058.07	*9870058*	9852803.35	1/20	19525.51	*9691848*	9647205.35	1/20	17097.70
p5000.5	***9822306**	9809770.55	1/20	12928.89	*9757089*	9733352.75	1/20	16660.47	*9594230*	9544658.80	1/20	17109.90
p6000.1	***9188495**	9172648.55	1/20	13235.67	*9119860*	9102598.10	1/20	19692.94	*8912507*	8879127.85	1/20	17827.85
p6000.2	***11695305**	11671648.95	1/20	20385.28	*11609371*	11578413.95	1/20	15881.51	*11383676*	11292837.15	1/20	17172.55
p6000.3	***13021696**	12992334.65	1/20	18872.06	*12917515*	12890377.50	1/20	16937.16	*12626976*	12544219.20	1/20	17273.50
p7000.1	***11587271**	11575532.90	1/20	21024.23	*11505139*	11477864.70	1/20	19788.91	*11208096*	11122645.05	1/20	17797.15
p7000.2	***14630551**	14612414.95	1/20	16824.76	*14522640*	14486184.80	1/20	16774.32	*14110703*	13959415.60	1/20	17642.60
p7000.3	***16342018**	16310296.40	1/20	14620.71	*16226164*	16157042.80	1/20	15459.21	*15671620*	15520258.70	1/20	17532.30
#Best/Equal/Worst	31/0/0				0/0/31				0/0/31			
p-value					1.17e-06	1.17e-06			1.17e-06	1.17e-06		

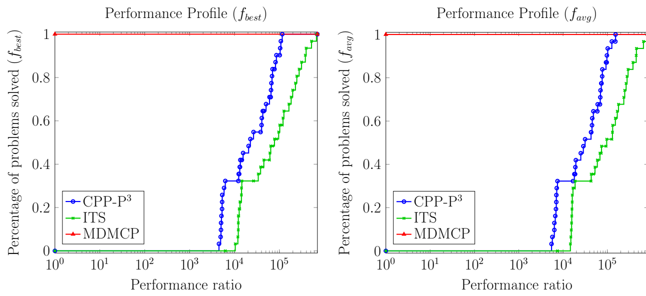
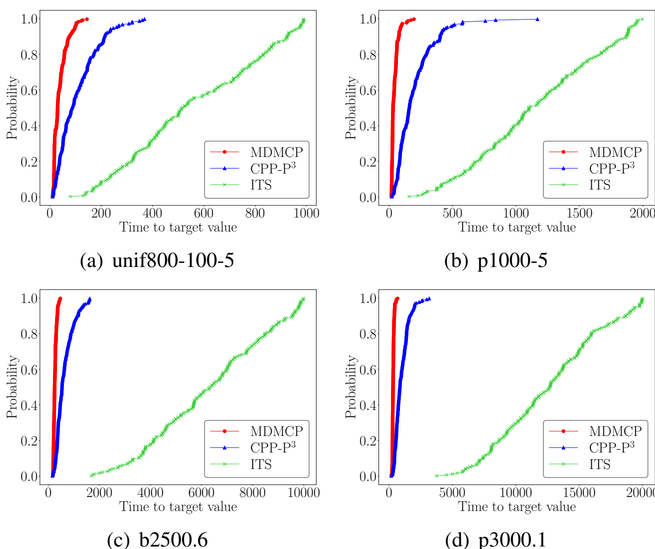


Fig. 3: Performance profiles of the MDMCP algorithm and reference algorithms CPP-P^3 [22] and ITS [21] on the set of 31 large instances.



(a) unif800-100-5 (b) p1000-5 (c) b2500.6 (d) p3000.1

Fig. 4: Probability distribution of the time (in seconds) needed to attain a given target objective value.

values (Note that the results of SGVNS are missing because we only have the executable code of SGVNS). These plots clearly indicate that our MDMCP algorithm always attains solutions of given target values much more faster than the reference algorithms. For example, in the case of *unif800-100-5* and *p1000-5*, the probability that MDMCP finds a target objective value in at most $100s$ is about 90%, while to reach the same result, CPP-P^3 and ITS require $200s$ and $1000s$ respectively. This experiment demonstrates that our MDMCP algorithm is more time efficient than the reference algorithms.

To sum up, we make the following observations. First, MDMCP and the state-of-the-art algorithms perform similarly on the small instances. Second, MDMCP begins to outperform all competitors on the set of medium size instances and its advantage becomes much more evident on the largest instances. We conclude that our population-based memetic approach is really powerful, especially for solving large instances.

VII. Analysis and Discussions

In this section, we perform additional experiments to get useful insights into the impacts of the main search components of the MDMCP algorithm and its parameters.

A. Assessment of memetic framework and local optimization

We assess the usefulness of 1) the population-based memetic search framework and 2) the SA-based local optimization procedure. For this purpose, we create two MDMCP variants: MDMCP^{Descent} and SALO^{Restart}. In MDMCP^{Descent}, we replace the SALO procedure by a pure descent procedure and keep the other MDMCP ingredients unchanged. Specifically, MDMCP^{Descent} uses the same neighborhood as the SALO procedure, but always chooses the best neighboring solution (ties broken at random) to replace the current solution. This variant allows us to verify the role of the SALO procedure within the MDMCP algorithm. In SALO^{Restart}, we keep SALO alone and disable the other ingredients of the MDMCP algorithm. This variant allows us to verify the importance of the population-based memetic framework. To avoid penalizing the SALO procedure, we restart SALO if SALO converges earlier than the allowed cutoff time. For this experiment, we focus on the 31 large instances and run MDMCP^{Descent} and SALO^{Restart} 20 times to solve each instance under the experimental protocol given in Section VI. The comparative results are reported in Table VI with the same statistics as in previous tables.

From Table VI, we can make the following observations. First, MDMCP^{Descent} reports the worst results in terms of best and average objective values, which are significantly worse than the results of MDMCP (confirmed by the small p-values). This indicates that disabling the simulated annealing based local optimization procedure greatly impacts (negatively) the performance of the MDMCP algorithm and the SALO procedure is one driving search component to ensure MDMCP's high performance. Second, SALO^{Restart} reports better results than MDMCP^{Descent}, but its results are still significantly worse than the results of MDMCP (confirmed by the small p-values). This indicates that removing the memetic ingredients (i.e., population, crossover) greatly degrades the performance of the MDMCP algorithm and the memetic framework positively contributes to the high performance of the MDMCP algorithm. Third, if we compare Tables V and VI, we observe that the SALO procedure alone competes favorably with the two other advanced local search algorithms (CPP-P^3 [22] and ITS [21]). This justifies the choice of SALO, instead of the pure descent-based local search, as the local optimization procedure of the MDMCP algorithm.

B. Impact of the parameter settings

The MDMCP algorithm requires 5 parameters (p, η, θ_{size}, θ_{cool}, and θ_{minper}) (see Table I). Among them, p is the population size, η is the shrinking percentage of the Merge-Divide crossover operator, while θ_{size}, θ_{cool}, θ_{minper} are required by the simulated annealing procedure. To investigate these parameters, we first performed a 2-level full factorial experiment [38] to check the interaction effects between the parameters. For this experiment, we used 10 instances of reasonable size and difficulty: {*rand300-5*, *rand500-100*, *p500-5-3*, *gauss500-100-3*, *unif700-100-2*, *unif800-100-4*, *p1000-1*, *p1500-3*, *p2000-1*, *p2000-4*}. This experiment (detailed results omitted) indicated that the parameters didn't show

TABLE VI: Comparisons of the MDMCP algorithm against a MDMCP variant where SALO is replaced by the descent (denoted by MDMCP^{Descent}) and a multi-start simulated annealing algorithm (denoted by SALO^{Restart}) on the 31 large instances.

Instance	**MDMCP**				MDMCP^{Descent}				SALO^{Restart}			
	f_{best}	f_{avg}	hit	$t(s)$	f_{best}	f_{avg}	hit	$t(s)$	f_{best}	f_{avg}	hit	$t(s)$
b2500.1	**1063447**	1061285.90	1/20	9603.47	1035376	1023423.55	1/20	7528.45	1061857	1059071.45	1/20	1562.25
b2500.2	**1063517**	1061926.35	1/20	7465.33	1038139	1024168.50	1/20	9179.00	1061436	1059209.70	1/20	9599.45
b2500.3	**1082275**	1080759.70	1/20	7991.01	1049939	1043955.30	1/20	9030.92	1078976	1077995.30	1/20	7183.11
b2500.4	**1065977**	1064729.95	1/20	9871.94	1036360	1027606.60	1/20	6571.26	1063708	1062156.90	1/20	7493.55
b2500.5	**1066387**	1063602.25	1/20	6777.81	1035516	1026547.25	1/20	4959.38	1063368	1061635.15	1/20	4103.49
b2500.6	**1066847**	1065194.10	1/20	4908.76	1038276	1027792.70	1/20	8030.66	1064224	1062578.70	1/20	6147.02
b2500.7	**1068161**	1066540.75	1/20	4400.32	1038707	1028545.85	1/20	3742.93	1066325	1064590.80	1/20	9587.66
b2500.8	**1069934**	1068766.85	1/20	5681.42	1042562	1032446.70	1/20	9756.89	1068220	1066249.85	1/20	6890.85
b2500.9	**1071272**	1069488.00	1/20	9088.52	1039552	1030482.65	1/20	8303.01	1068984	1067145.35	1/20	4861.28
b2500.10	**1066735**	1065303.15	1/20	8823.48	1035002	1026529.65	1/20	4854.38	1064969	1062393.15	1/20	1629.67
p3000.1	**3257061**	3253637.60	1/20	14031.83	3179189	3157880.50	1/20	11317.57	3249711	3245383.50	1/20	10765.46
p3000.2	**4099540**	4095924.60	1/20	10344.62	3989090	3969276.25	1/20	19591.90	4092839	4086648.90	1/20	911.81
p3000.3	**4121651**	4116510.05	1/20	14861.91	4019441	3993245.30	1/20	5506.45	4107522	4102876.20	1/20	10288.91
p3000.4	**4586819**	4582002.80	1/20	19683.10	4479556	4434224.75	1/20	17814.79	4574507	4568475.75	1/20	10930.15
p3000.5	**4638416**	4628915.50	1/20	8037.94	4531334	4489278.05	1/20	15408.93	4620217	4614552.60	1/20	7218.19
p4000.1	**5013660**	5008649.80	1/20	19087.02	4884964	4857029.25	1/20	15950.41	5005084	4997783.85	1/20	1575.19
p4000.2	**6376823**	6367326.50	1/20	15685.50	6203953	6164335.30	1/20	18019.52	6364505	6350604.60	1/20	14000.52
p4000.3	**6386325**	6373723.90	1/20	18755.28	6201018	6175411.85	1/20	8566.06	6374324	6354520.80	1/20	4282.66
p4000.4	**7121377**	7114980.55	1/20	17620.49	6970604	6909068.75	1/20	19118.89	7102379	7095909.55	1/20	15192.19
p4000.5	**7045223**	7033485.25	1/20	11596.08	6863026	6823538.35	1/20	19974.46	7019098	7012308.20	1/20	12446.18
p5000.1	**7009948**	6990929.85	1/20	17047.25	6810549	6776700.30	1/20	14837.36	6988837	6973262.50	1/20	17845.95
p5000.2	**8827548**	8817131.05	1/20	17491.37	8597361	8563585.90	1/20	16372.30	8804596	8793274.75	1/20	18345.06
p5000.3	**8965743**	8949408.10	1/20	19998.82	8754076	8682592.55	1/20	15066.93	8943630	8920049.80	1/20	3610.66
p5000.4	**9936590**	9925708.30	1/20	13058.07	9678251	9627255.85	1/20	17022.34	9920825	9903909.25	1/20	13068.94
p5000.5	**9822306**	9809770.55	1/20	12928.89	9601866	9526260.35	1/20	17139.90	9808196	9780004.05	1/20	8057.36
p6000.1	**9188495**	9172648.55	1/20	13235.67	8955011	8911414.35	1/20	18841.14	9174154	9162432.75	1/20	18147.17
p6000.2	**11695305**	11671648.95	1/20	20385.28	11398593	11346418.10	1/20	16085.46	11672843	11653217.35	1/20	5475.32
p6000.3	**13021696**	12992334.65	1/20	18872.06	12707909	12628831.05	1/20	15599.26	13008275	12980911.00	1/20	7291.35
p7000.1	**11587271**	11575532.90	1/20	21024.23	11319801	11268011.35	1/20	19638.84	11586258	11569685.80	1/20	18561.02
p7000.2	**14630551**	14612414.95	1/20	16824.76	14311736	14228451.10	1/20	17494.08	14627372	14611674.90	1/20	12032.58
p7000.3	**16342018**	16310296.40	1/20	14620.71	15942790	15861301.35	1/20	12572.60	16335446	16302131.85	1/20	21255.67
#Best/Equal/Worse	31/0/0				0/0/31				0/0/31			
p-value					1.17e-06	1.17e-06			1.17e-06	1.17e-06		

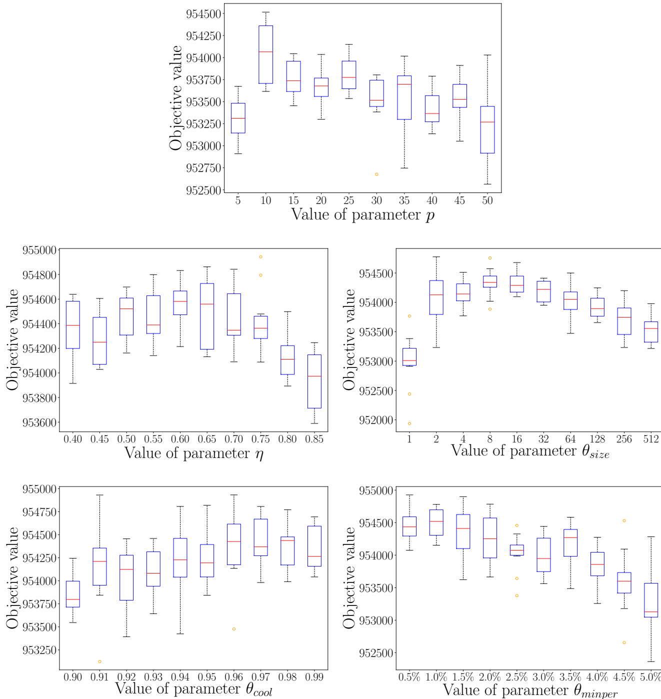


Fig. 5: Analysis of the parameters (p, η, θ_{size}, θ_{cool}, θ_{minper}) on the performance of the MDMCP algorithm.

significant interactions. Then we performed a one-at-a-time sensitivity analysis [39] to calibrate the parameters as follows. Based on preliminary experiments, we empirically identified a reasonable range of values for each parameter: $p \in [5,\ 10, \ldots, 45,\ 50]$, $\eta \in [0.40,\ 0.45, \ldots, 0.80,\ 0.85]$, $\theta_{size} \in [1,\ 2, \ldots, 256,\ 512]$, $\theta_{cool} \in [0.90,\ 0.91, \ldots, 0.98,\ 0.99]$, and $\theta_{minper} \in [0.5\%,\ 1.0\%, \ldots, 4.5\%,\ 5.0\%]$. Then we tested the values of each parameter independently while fixing the other parameters according to our preliminary experiments (including the values of Table I). We ran the MDMCP algorithm 10 times with each parameter value to solve each of the 10 instances above with the same cutoff time as before. Fig. 5 shows the box and whisker plots of the results, where the X-axis and Y-axis indicate the parameter values and the accumulated best objective values over the 10 test instances, respectively.

Fig. 5 indicates that the performance of MDMCP varies according to the values of these parameters. We adopt the best parameter values of this experiment to define the default setting of Table I, which proves to be quite robust for solving all benchmark instances tested in this work. In practice, when a particular problem is considered, it would be worth fine-tuning these parameters to achieve the best possible results.

VIII. Conclusion

We introduced the first hybrid evolutionary algorithm (MDMCP) dedicated to the challenging Clique Partitioning

Problem. Based on the general memetic framework, the algorithm combines a specialized crossover for solution recombination and an effective simulated annealing optimizer. Extensive computational evaluations of the algorithm on three sets of 94 benchmark instances demonstrated its competitiveness compared to the state-of-the-art methods. In particular, the algorithm established new lower bounds for 12 out of the 63 instances commonly used in the literature and the best lower bounds for the 31 new large instances.

Since the clique partitioning problem can formulate various real-world applications (see examples in Section I), the proposed algorithm can help to better solve these practical problems. The availability of the source code of our algorithm certainly facilitates such applications.

Finally, given the interest of the hybrid evolutionary approach for this partitioning problem, it would be interesting to investigate the benefits of the underlying ideas of the proposed algorithm for solving other related problems such as k-way graph partitioning, clique cover and graph coloring.

Acknowledgment

We are grateful to the reviewers for their useful comments and suggestions which helped us to significantly improve the paper. We thank the authors of [23] for sharing the code of their SGVNS algorithm and the author of [21] to make the code of the ITS algorithm publicly available.

References

[1] M. Grötschel and Y. Wakabayashi, "A cutting plane algorithm for a clustering problem," *Mathematical Programming*, vol. 45, no. 1-3, pp. 59–96, 1989.

[2] M. Grötschel and Y. Wakabayashi, "Facets of the clique partitioning polytope," *Mathematical Programming*, vol. 47, no. 1-3, pp. 367–387, 1990.

[3] A. Miyauchi, T. Sonobe, and N. Sukegawa, "Exact clustering via integer programming and maximum satisfiability," in *Thirty-Second AAAI Conference on Artificial Intelligence, February 2-7, 2018, New Orleans, Louisiana, USA*, 2018, pp. 1387–1394.

[4] M. Oosten, J. H. Rutten, and F. C. Spieksma, "The clique partitioning problem: facets and patching facets," *Networks: An International Journal*, vol. 38, no. 4, pp. 209–226, 2001.

[5] H. Wang, B. Alidaee, F. Glover, and G. Kochenberger, "Solving group technology problems via clique partitioning," *International Journal of Flexible Manufacturing Systems*, vol. 18, no. 2, pp. 77–97, 2006.

[6] U. Dorndorf, F. Jaehn, and E. Pesch, "Modelling robust flight-gate scheduling as a clique partitioning problem," *Transportation Science*, vol. 42, no. 3, pp. 292–301, 2008.

[7] D. Aloise, S. Cafieri, G. Caporossi, P. Hansen, S. Perron, and L. Liberti, "Column generation algorithms for exact modularity maximization in networks," *Physical Review E*, vol. 82, no. 4, p. 046112, 2010.

[8] G. Kochenberger, F. Glover, B. Alidaee, and H. Wang, "Clustering of microarray data via clique partitioning," *Journal of Combinatorial Optimization*, vol. 10, no. 1, pp. 77–92, 2005.

[9] J. Bhasker and T. Samad, "The clique-partitioning problem," *Computers & Mathematics with Applications*, vol. 22, no. 6, pp. 1–11, 1991.

[10] N. Sukegawa, Y. Yamamoto, and L. Zhang, "Lagrangian relaxation and pegging test for the clique partitioning problem," *Advances in Data Analysis and Classification*, vol. 7, no. 4, pp. 363–391, 2013.

[11] U. Dorndorf and E. Pesch, "Fast clustering algorithms," *ORSA Journal on Computing*, vol. 6, no. 2, pp. 141–153, 1994.

[12] X. Ji and J. E. Mitchell, "Branch-and-price-and-cut on the clique partitioning problem with minimum clique size requirement," *Discrete Optimization*, vol. 4, no. 1, pp. 87–102, 2007.

[13] A. Mehrotra and M. A. Trick, "Cliques and clustering: A combinatorial approach," *Operations Research Letters*, vol. 22, no. 1, pp. 1–12, 1998.

[14] F. Jaehn and E. Pesch, "New bounds and constraint propagation techniques for the clique partitioning problem," *Discrete Applied Mathematics*, vol. 161, no. 13-14, pp. 2025–2037, 2013.

[15] I. Boussaïd, J. Lepagnot, and P. Siarry, "A survey on optimization metaheuristics," *Information Sciences*, vol. 237, pp. 82–117, 2013.

[16] T. Dokeroglu, E. Sevinc, T. Kucukyilmaz, and A. Cosar, "A survey on new generation metaheuristic algorithms," *Computers & Industrial Engineering*, vol. 137, p. 106040, 2019.

[17] S. G. De Amorim, J. P. Barthélemy, and C. C. Ribeiro, "Clustering and clique partitioning: simulated annealing and tabu search approaches," *Journal of Classification*, vol. 9, no. 1, pp. 17–41, 1992.

[18] I. Charon and O. Hudry, "The noising methods: A generalization of some metaheuristics," *European Journal of Operational Research*, vol. 135, no. 1, pp. 86–101, 2001.

[19] I. Charon and O. Hudry, "Noising methods for a clique partitioning problem," *Discrete Applied Mathematics*, vol. 154, no. 5, pp. 754–769, 2006.

[20] M. J. Brusco and H. F. Köhn, "Clustering qualitative data based on binary equivalence relations: neighborhood search heuristics for the clique partitioning problem," *Psychometrika*, vol. 74, no. 4, pp. 685–703, 2009.

[21] G. Palubeckis, A. Ostreika, and A. Tomkevičius, "An iterated tabu search approach for the clique partitioning problem," *The Scientific World Journal*, vol. 2014, p. 353101, 2014.

[22] Y. Zhou, J. K. Hao, and A. Goëffon, "A three-phased local search approach for the clique partitioning problem," *Journal of Combinatorial Optimization*, vol. 32, no. 2, pp. 469–491, 2016.

[23] J. Brimberg, S. Janićijević, N. Mladenović, and D. Urošević, "Solving the clique partitioning problem as a maximally diverse grouping problem," *Optimization Letters*, vol. 11, no. 6, pp. 1123–1135, 2017.

[24] S. Régnier, "Sur quelques aspects mathématiques des problèmes de classification automatique," *Mathématiques et Sciences humaines*, vol. 82, pp. 13–29, 1983.

[25] D. C. Porumbel, J. K. Hao, and P. Kuntz, "An efficient algorithm for computing the distance between close partitions," *Discrete Applied Mathematics*, vol. 159, no. 1, pp. 53–59, 2011.

[26] P. Moscato and C. Cotta, "A gentle introduction to memetic algorithms," in *Handbook of Metaheuristics*. Springer, 2003, pp. 105–144.

[27] F. Neri, C. Cotta, and P. Moscato, *Handbook of memetic algorithms*. Springer, 2011, vol. 379.

[28] U. Benlic and J. K. Hao, "A multilevel memetic approach for improving graph k-partitions," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 5, pp. 624–642, 2011.

[29] C. Liao and C. Ting, "A novel integer-coded memetic algorithm for the set k-cover problem in wireless sensor networks," *IEEE Transactions on Cybernetics*, vol. 48, no. 8, pp. 2245–2258, 2018.

[30] S. Wang, J. Liu, and Y. Jin, "Finding influential nodes in multiplex networks using a memetic algorithm," *IEEE Transactions on Cybernetics*, pp. 1–13, 2019.

[31] J. Wu, X. Shen, and K. Jiao, "Game-based memetic algorithm to the vertex cover of networks," *IEEE Transactions on Cybernetics*, vol. 49, no. 3, pp. 974–988, 2019.

[32] Y. Zhou, J. K. Hao, and F. Glover, "Memetic search for identifying critical nodes in sparse graphs," *IEEE Transactions on Cybernetics*, vol. 49, no. 10, pp. 3699–3712, 2018.

[33] J. K. Hao, "Memetic algorithms in discrete optimization," in *Handbook of Memetic Algorithms*. Springer, 2012, pp. 73–94.

[34] Y. Jin and J. Hao, "Solving the latin square completion problem by memetic graph coloring," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 6, pp. 1015–1028, Dec 2019.

[35] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical Programming*, vol. 91, no. 2, pp. 201–213, 2002.

[36] A. S. Siqueira, R. C. da Silva, and L. R. Santos, "Perprof-py: A python package for performance profile of mathematical optimization software," *Journal of Open Research Software*, vol. 4, no. 1, 2016.

[37] R. M. Aiex, M. G. Resende, and C. C. Ribeiro, "TTT plots: a perl program to create time-to-target plots," *Optimization Letters*, vol. 1, no. 4, pp. 355–366, 2007.

[38] D. C. Montgomery, *Design and analysis of experiments*. John Wiley & Sons, 2017.

[39] D. M. Hamby, "A review of techniques for parameter sensitivity analysis of environmental models," *Environmental Monitoring and Assessment*, vol. 32, no. 2, pp. 135–154, 1994.