A Three-Phased Local Search Approach for the Clique Partitioning Problem

Yi Zhou · Jin-Kao Hao[+] · Adrien Goëffon

Abstract This paper presents a three-phased local search heuristic CPP-P^3 for solving the Clique Partitioning Problem (CPP). CPP-P^3 iterates a descent search, an exploration search and a directed perturbation. We also define the *Top Move* of a vertex, in order to build a restricted and focused neighborhood. The exploration search is ensured by a tabu procedure, while the directed perturbation uses a GRASP-like method. To assess the performance of the proposed approach, we carry out extensive experiments on benchmark instances of the literature as well as newly generated instances. We demonstrate the effectiveness of our approach with respect to the current best performing algorithms both in terms of solution quality and computation efficiency. We present improved best solutions for a number of benchmark instances. Additional analyses are shown to highlight the critical role of the *Top Move*-based neighborhood for the performance of our algorithm and the relation between instance hardness and algorithm behavior.

In memory of Professor Wenqi Huang who has dedicated his life to research on heuristic methods for optimization

Yi Zhou
LERIA, Université d'Angers, 2 Boulevard Lavoisier, 49045 Angers, France
E-mail: zhou@info.univ-angers.fr

Jin-Kao Hao (Corresponding author)
a) LERIA, Université d'Angers, 2 Boulevard Lavoisier, 49045 Angers, France
b) Institut Universitaire de France, Paris, France
E-mail: jin-kao.hao@univ-angers.fr

Adrien Goëffon
LERIA, Université d'Angers, 2 Boulevard Lavoisier, 49045 Angers, France
E-mail: adrien.goeffon@univ-angers.fr
Accepted to Journal of Combinatorial Optimization, Oct 2015

1 Introduction

Let $G = (V, E, W)$ be a complete edge-weighted undirected graph with a vertex set $V = \{v_1, v_2, ...v_n\}$, an edge set $E = \{e_{uv} = \{u, v\} : u, v \in V, u \neq v\}$ and a set of edge weights $W = \{w_{uv} : w_{uv} \in \mathbb{R} \cup \{-\infty\}, e_{uv} \in E, w_{uv} = w_{vu}\}$. The Clique Partitioning Problem (CPP) consists in clustering all the vertices into k (unfixed) mutually disjoint subsets (or groups), such that the sum of the edge weights of all groups is as large as possible [17,18,30]. Formally, let $s = \{G_1, G_2, \ldots, G_k\}$ be such a partition of a given graph G such that $\bigcup_{i=1}^{k} G_i = V$ and $G_i \cap G_j = \emptyset$ ($\forall 1 \leq i < j \leq k$). Let Ω denote the set of all partitions for G. CPP is then to find a partition $s^* \in \Omega$ that maximizes the following function:

$$f(s) = \sum_{p=1}^{k} \sum_{u,v \in G_p} w_{uv} \tag{1}$$

The set of edges connecting vertices of different groups of a partition of G is called a cut of G. In order to find groups as homogeneous as possible, positive edges should appear within groups and negative edges in the cut. Hence, a best partition has a minimal cut weight. If all the edge weights are non-negative or non-positive, the problem can be easily solved. If the graph has both negative and positive edge weights, then the decision problem of CPP is NP-complete [30].

CPP has a number of practical applications such as biology, flexible manufacturing systems, airport logistics, and social sciences. For instance, in qualitative data analysis, CPP can be used to uncover natural groupings, or types of objects, each one being characterized by several attributes. One can bijectively associate these objects with vertices of an edge-weighted graph G; each positive or negative edge weight represents some measure of similarity or dissimilarity of two objects linked by the edge. The associated CPP problem consists in determining an appropriate partition of the vertices. In biology, the classification of animals and plants is based on qualitative and/or quantitative descriptions. As the number of classes is unknown a priori, CPP is well suited for determining such classifications [17]. In transportation, an application of CPP to a flight-gate scheduling problem is presented in [9]. The problem is modeled as a Clique Partitioning Problem and then solved by a heuristic algorithm based on ejection chain algorithm. In manufacturing, CPP is used to determine different groups of products as shown in [27,31].

Given the relevance of CPP, a number of solving procedures have been reported in the literature, including both exact and heuristic methods. Most of the exact methods follow the branch and bound framework [10,20,21]) and cutting plane method [17]. However, exact methods may become prohibitively expensive when they are used to solve large instances. Consequently, various heuristic algorithms have been proposed to find high-quality solutions to large CPP instances with acceptable computation time. In [8], tabu search [16] and simulated annealing [24] were applied to this problem. Improved solutions were reported in [4] by using a reallocation heuristic and an embedded tabu search

routine. In [9,10], an ejection chain heuristic was presented, borrowing the idea of classical Kernighan-Lin algorithm [23]. In [5,6], authors proposed a noising method which adds decreasing levels of noise to neighbor evaluations. More recently, two heuristic algorithms [28,3] were presented. The work of [28] focuses on an iterated tabu search with tuned parameters. The approach of [3] uses a generalized VNS algorithm designed for the *Maximally Diverse Grouping Problem* to solve CPP. These two last CPP algorithms integrate two neighborhood relations: (i) reallocating one vertex and (ii) swapping two vertices. Both of them performs quite well on the instances of [4] as well as large graphs with more than 1000 vertices. In the experimental section of this work, these two algorithms will be used as the main references for comparisons.

In this paper, we introduce a novel heuristic algorithm, denoted by CPP-P^3, which uses a Three Phase Local Search framework combining a descent procedure, a tabu-based exploration search and GRASP-like perturbations. Different from previous local search algorithms, CPP-P^3 introduces a constrained *Top Move* neighborhood to make the search more focused and calls for a heap data structure to ensure a fast neighborhood examination. Computational experiments on four groups of graph instances (from 100 to 2000 vertices) show that CPP-P^3 competes favorably with the current best performing algorithms (including ITS [28] and SGVNS [3]) both in terms of solution quality and computational efficiency. We also carry out additional analysis to gain insights about the behavior of the proposed algorithm.

The remaining part of the paper is organized as follows. Section 2 describes the components of the CPP-P^3 algorithm and presents the concept of the *Top Move* neighborhood. Section 3 is dedicated to computational results and detailed comparisons with state-of-art algorithms. Section 4 investigates the effectiveness of the *Top Move* neighborhood and the hardness of problem instances by means of a landscape analysis. Conclusions are drawn in the last section.

2 General Procedure

The proposed CPP-P^3 method (see Algorithm 1) follows the general scheme of the Three-Phase Search (TPS) introduced in [12]. From a given starting solution, the first phase aims to reach a local optimal solution using a basic descent procedure. From this local optimum, the second phase is applied to discover nearby local optima of better quality within the current regional area of the search space. If no further improvement can be obtained, TPS switches to its third phase to perturb the incumbent solution to displace the search to a more distant area from which a new round of the three phased process is launched.

In this section, we first describe fundamental elements of the proposed procedure: the search space and the evaluation function (Section 2.1), the specific neighborhood induced by the *Top Move* concept (Section 2.2) and the associated heap structure (Section 2.3). Then, we present each subroutine of the

Algorithm 1, i.e. the generation of initial solution and the three phases Descent_Search(), Explore_Local_Optima(), and Diversified_Perturbation(). We also discuss the main differences between CPP-P^3 and the existing heuristic algorithms in Section 2.7.

Algorithm 1 Pseudo-code of CPP-P^3

1: **Input**: A graph instance described by a weight matrix G
2: **Output**: The best solution s_{best} and its objective value f_{best}
3: $s \leftarrow$ Build_Init_Solution()   /∗ Section 2.4 ∗/
4: $f_{best} \leftarrow 0$
5: **while** stop condition is not met **do**
6:     $s \leftarrow$ Descent_Search(s)   /∗ Section 2.5 ∗/
7:     $s \leftarrow$ Explore_Local_Optima(s)   /∗ Section 2.6 ∗/
8:     **if** $f(s) > f_{best}$ **then**
9:         $s_{best} \leftarrow s$
10:         $f_{best} \leftarrow f(s)$
11:     **end if**
12:     $s \leftarrow$ Diversified_Perturbation(s)   /∗ Section 2.7 ∗/
13: **end while**

2.1 Search Space and Evaluation Function

As previously mentioned, a solution s can be expressed as a partition of the set V of vertices into k mutually disjoint *groups* $\{G_1, G_2, ..., G_k\}$, such as $\bigcup_{i=1}^{k} G_i = V$, $G_i \cap G_j = \emptyset, \forall i \neq j$, and $\forall i, |G_i| > 0$. The number of groups $k \in \{1, \ldots, |V|\}$ is not fixed. The search space of a given CPP instance is composed of all such solutions and has a cardinality of $|\Omega| = \sum_{k=1}^{n} S(n, k)$, where $S(n, k)$ denotes the Stirling number of the second kind. Enumerating such a huge search space is obviously impractical, even on reasonably small graphs.

To evaluate the quality of a candidate solution $s \in \Omega$, we simply use the objective function f (see Equation (1)), which sums the weights of the edges whose end-points are inside the same group.

2.2 Top Move and Restricted Neighborhood

CPP-P^3 is a local search algorithm, which requires a neighborhood allowing the search to navigate through a given search space. The most intuitive way to transform one solution to a close solution for CPP is probably to reallocate one vertex from its current group to another group. Compared with other popular neighbor operators such as swapping two vertices or reallocating an edge, the size of the neighborhood of reallocating a vertex is relatively reasonable for complete graphs. Also, it is easy to observe that each solution can lead to any other solution by applying this reallocation operator. This property makes it possible for the algorithm to explore potentially the whole search space, depending on the selection criterion.

We now define the reallocation operator in a more formal way. Given an feasible solution (partition) $s = \{G_1, G_2, ..., G_k\}$, a neighbor of s is a solution s' which can be obtained after moving one vertex u from its original group G_u to another group G_t ($G_t \in s \cup \{\emptyset\} \setminus \{G_u\}$), including a potentially new group. When $G_t = \emptyset$, a new group $G'_t = G_t \cup \{u\} = \{u\}$ will be added to s' as the $(k+1)$-th group. On the other hand, if $G'_u = G_u \setminus \{u\} = \emptyset$, it will not be conserved in s' anymore.

The key concept related to our neighborhood is the *move gain*, which indicates the change of the objective value between solution s and a neighbor s'. In order to calculate the move gain as fast as possible, we firstly define the *potential* of vertex u relative to G_i, ($G_i \in s \cup \{\emptyset\}$):

$$p_{u,G_i} = \sum_{v \in G_i} w_{vu} \tag{2}$$

Given $u \in G_u$ in s, $move(u, G_t)$ the move which reallocates u from group G_u to G_t, the move gain Δ_{u,G_t} (the variation of objective f) of $move(u, G_t)$ can be incrementally computed by:

$$\Delta_{u,G_t} = f(s') - f(s) = p_{u,G_t} - p_{u,G_u} \tag{3}$$

In previous studies on CPP like [4,6,8,28], selecting an appropriate *move* (u, G_t) requires the evaluation of all feasible neighboring solutions, i.e., by considering all reallocations of each vertex to each group including the empty one. Let $s \oplus move(u, G_t)$ designate the neighboring solution s' obtained by applying $move(u, G_t)$ to s, then this complete neighborhood $N(s)$ is defined by:

$$N(s) = \{s' \leftarrow s \oplus move(u, G_t) : \forall u \in V, \forall G_t \in s \cup \{\emptyset\} \setminus \{G_u\}\} \tag{4}$$

In order to make the search more focused and more efficient, our CPP-P^3 algorithm follows the idea of elite candidate list [16] and builds a *restricted neighborhood* which is defined with the notion of *Top Move*.

Definition 1 A $move(u, G_\zeta)$ is called *Top Move* of vertex u ($u \in G_i$) if

$$G_\zeta = \underset{G_t \in s \cup \{\emptyset\} \setminus G_u}{\operatorname{argmax}} (\Delta_{u,G_t})$$

The target group G_ζ is denoted as G_{TM_u}, while the move gain Δ_{u,G_ζ} is denoted as Δ_{TM_u}.

So the *Top Move* of a vertex identifies the destination group with the largest move gain. Then the *restricted neighborhood* induced by the *Top Moves* of all vertices is given by:

$$N'(s) = \{s' \leftarrow s \oplus move(u, G_{TM_u}) : \forall\ u \in V\} \tag{5}$$

In the general case, $|N(s)| = k \times |V|$ for the complete neighborhood[1] while $N'(s) = |V|$ for the restricted neighborhood. Consequently, it is easier to evaluate the candidate solutions in $N'(s)$ than in $N(s)$ (we will show a comparison of using these two neighborhoods in Section 4). We present below a fast method to identify the *Top Move* of each vertex in each iteration of the algorithm.

Notice that in several recent studies [7,32,33], such a restricted neighborhood strategy has been shown to be particularly useful.

2.3 Heap structure

Let s be the incumbent solution. Suppose that $move(u, G_t)$ $(u \in G_u)$ is chosen and executed to obtain the desired neighboring solution s', we update $G'_u = G_u \setminus \{u\}$, $G'_t = G_t \cup \{u\}$ as well as the potentials of each vertex $v \in V$ as follows:

$$\begin{cases} p'_{v,G'_u} = p_{v,G_u} - w_{uv}, \ G'_u = G_u \setminus \{u\} \\ p'_{v,G'_t} = p_{v,G_t} + w_{uv}, \ G'_t = G_t \cup \{u\} \end{cases} \tag{6}$$

To speed up the search and updating procedures, we store the potentials of each vertex in a heap structure. For a vertex u belonging to G_u in the current solution s, the potentials of u to the groups $s \cup \{\emptyset\} \setminus \{G_u\}$ are stored in the heap such that the top element $p_{u,G_{top}}$ of the heap is always the largest potential of vertex u. According to Equation (3), G_{top} will be the target group G_{TM_u}.

To maintain the heap, normally, when $move(u, G_t)$ $(u \in G_u)$ is executed, p'_{v,G'_u} and p'_{v,G'_t} will replace p_{v,G_u} and p_{v,G_t} respectively and their positions in the heap of vertex v will be adjusted. However, to ensure potential $p_{v,\emptyset} = 0$ always be exclusively conserved in the heap, two exceptional situations should be considered. If $G'_u = \emptyset$ in s', p'_{v,G'_u} should be removed from the heap to avoid the repetition of $p_{v,\emptyset}$. If $G'_t = \{u\}$ (i.e., $G_t = \emptyset$), we should add a new potential $p'_{v,G'_t} = p_{v,\{u\}} = p_{v,\emptyset} + w_{uv}$, rather than replace $p_{v,\emptyset}$.

By this method, the *Top Move* of one vertex can be found in $O(1)$ time from the top of the heap. Moreover, updating the whole heap structure costs $O(|V| \cdot \log |V|)$.

2.4 Generation of initial solution

For the Clique Partitioning Problem, any partition of the vertices of G is a possible feasible solution. The question is then to fix the number of groups in the initial solution. In CPP-P^3, we simply assign each vertex to an exclusive group. Considering the initial solution will be immediately improved by the descent search, the quality of the initial solution is not essential in our case.

---

[1] If some groups contain exactly one vertex, then several moves lead to equivalent solutions. Thus, $|N(s)|$ can be slightly inferior to $k \times |V|$.

2.5 Descent Search phase

The descent search phase aims at finding new solutions of better quality from a given initial solution. For this purpose, it builds a search trajectory by examining the restricted neighborhood defined in Section 2.2. More precisely, the descent iteratively improves the quality of the current solution by searching the given neighborhood and stops when a local optimum is reached, i.e., when no better solution can be found in the neighborhood. At each iteration, vertices are evaluated in a random order. Let u be the vertex under consideration and s be the current solution. If the *Top Move* gain Δ_{TM_u} of vertex u is a positive number (i.e., the Top Move with vertex u leads to an improved solution with respect to the incumbent solution), $move(u, G_{TM_u})$ is executed and the neighboring solution $s \oplus move(u, G_{TM_u})$ replaces s to become the new incumbent solution. Otherwise the evaluation continues with the next vertex (always randomly chosen). If no *Top Move* offers a positive gain during one iteration after examining all the vertices, then the descent search stops and the last solution (i.e., a local optimum) is returned and used as the starting solution of the second search phase (Exploration Search, see next section).

Note that within the restricted neighborhood N', the way the neighboring solutions are examined and selected corresponds to the *random-order first improvement* search strategy. Clearly, the selected neighbor using this strategy is not necessarily the best neighbor within the restricted neighborhood N' nor within the complete neighborhood N.

The computing time of each iteration is bounded by $O(|V| + |V| \cdot \log |V|)$.

2.6 Exploration Search phase

From the local optimum solution returned by the descent search phase, the second search phase explores the nearby areas for better solutions. For this purpose, the Explore_Local_Optima procedure (Algorithm 2) adopts a tabu search method [16] also based on the above restricted neighborhood.

At each iteration, a Top Move $move(u, G_{TM_u})$ is considered to be eligible only if two conditions are satisfied. First, the move must be a non-trivial one (i.e., reallocating a vertex from a group with only one vertex to an empty set is forbidden). Second, the move is not forbidden by the tabu list except when the move leads to a new solution better than any previous visited solution. Among these eligible moves, the one with the largest move gain is chosen and executed.

To avoid short-term cycling, the executed $move(w, G_{TM_w})$ as well as the index of the original group of vertex w are stored in the tabu list. As such, vertex w is prohibited to move back to its original group during the next tt iterations (tt is called the *tabu tenure*). However, if the vertex of the executed move is the only vertex in its original group, then the vertex will be forbidden to be moved into the empty group for the next tt iterations.

Algorithm 2 Tabu-based exploration phase (Explore_Local_Optima).

1: **Input**: current solution s, current best objective value f_{best}
2: **Output**: best solution s^*
3: Initialize the tabu list
4: $iter \leftarrow 0$ /* total number of iterations */
5: **repeat**
6:    Find out vertex w with the maxium Δ_{TM_w} from the vertices which meets the following
      2 conditions:
      1. ($|G_u| = 1$ AND $|G_{TM_u}| \neq \emptyset$) OR ($|G_u| > 1$)
      2. ($move(u, G_{TM_u})$ is not TABU) OR ($\Delta_{TM_u} + f(s) > f_{best}$)
7:    **if** $|G_w| = 1$ **then**
8:       Mark $move(w, \emptyset)$ tabu for the next tt iterations
9:    **else**
10:       Mark $move(w, G_w)$ tabu for the next tt iterations
11:    **end if**
12:    Execute $move(w, G_{TM_w})$ and update data structure
13:    **if** $f(s) > f_{best}$ **then**
14:       $s^* \leftarrow s$, $f_{best} \leftarrow f(s)$
15:    **end if**
16:    $iter \leftarrow iter + 1$
17: **until** f_{best} has not been improved for L iterations

---

It is well known that the performance of a tabu search algorithm depends on the way the tabu tenure is determined [16]. However, no optimal technique is universally available to tune the tabu tenure, which is often fixed empirically in practice. In our case, we adopt the technique proposed in [14] and use a base length *tbase* adjusted with a random value as follows:

$$tt = tbase + random(0, k) \tag{7}$$

where $random(0, k)$ is a random integer in $\{0, \ldots, k\}$ (k being the number of groups of the current solution). After preliminary robustness tests (see Section 3.1), we set *tbase* to 15.

For this exploration search phase, the algorithm is supposed to reach a local optimum if the best solution cannot be improved during L consecutive iterations. In this case, the Explore_Local_Optima phase stops and the best solution found is used as the input solution of the Diversified_Perturbation phase.

2.7 Directed perturbation phase

The perturbation phase (see Algorithm 3) aims to jump out of the current regional search area and displace the search into a distant area. Instead of relying on random perturbation (e.g., randomly move some vertices), CPP-P^3 uses a *directed perturbation* mechanism [2].

The perturbation concerns reallocating a number of specific vertices from a *Restricted Candidate List* (RCL). For each perturbation phase, the number of reallocated vertices, which defines the perturbation strength, is randomly selected from range $\{\alpha \times |V|, \ldots, \beta \times |V|\}$, where α and β ($0 \leq \alpha \leq \beta \leq 1$) are two parameters.

Algorithm 3 Perturbation phase (Diversified_Perturbation).

1: **Input**: current solution s
2: **Output**: perturbated solution s^*
3: $PL \leftarrow random([\alpha \times |V|, \beta \times |V|])$
4: $M \leftarrow \emptyset$
5: **while** $|M| < PL$ **do**
6:     Construct $RCL = \{$The $Maxrcl$ vertices in $V \setminus M$ with the greatest $\Delta_{TM_u}\}$
7:     Randomly choose a vertex w from RCL
8:     Execute $move(w, G_{TM_w})$ and update data structure
9:     $M \leftarrow M \cup \{w\}$
10: **end while**

To determine the vertices that will be reallocated during the perturbation phase, we build an RCL to store the partial best candidates, like with the GRASP method [11]. Here, the RCL contains the $Maxrcl$ first vertices with the largest *Top Move* gain. For each perturbation step, a vertex w is randomly chosen from the RCL and the corresponding *Top Move* is executed. Once a vertex is reallocated, it is removed from RCL during the current perturbation phase.

To implement the RCL efficiently, we employ once again a heap structure to make sure that it always contains the best $Maxrcl$ vertices and ignores the other ones. Thus, the construction of the RCL in each iteration takes $O(V \cdot \log Maxrcl)$ time, while updating potentials consumes $O(V \cdot \log V)$ after one move. The complexity of one iteration in each perturbation phase is bounded by $O(V \cdot \log V)$.

2.8 Singularity of CPP-P^3

We now discuss the major differences between CPP-P^3 and other local search algorithms for CPP. ITS [28] may be the closest approach for solving CPP as it also integrates a descent search, a tabu search and a perturbation procedure. However, there are three main differences between the two algorithms. First, ITS alternates two neighbor operators, i.e. reallocating a vertex and swapping two vertices, while only the first operator is considered in CPP-P^3. Second, ITS evaluates all the neighbor solution induced by the reallocating operators to determine the neighbor solution with the maximum objective gain while CPP-P^3 seeks the best neighbor solution from a restricted solution set associated with *Top Move* definition. Third, the two algorithms employ different strategies for managing the tabu tenure. The value in ITS is more deterministic while CPP-P^3 is more randomized. As we show in Section 3, these differences make the proposed CPP-P^3 more effective than ITS.

CPP-P^3 also shares similarities with Eject Chain algorithm [10] in the sense that both approaches use a heap structure to identify the move. In [10], the move with the largest objective gain is always preferred, while in the tabu phase of CPP-P^3, the vertex associates with the best *Top Move* is selected. Therefore, in CPP-P^3, the moves whose associated *Top Move* is marked tabu

will not be considered, even though such a move may lead to the best gain of objective value. We investigate in Section 4.1 the effect of both strategies and demonstrate the interest of the strategy we adopted in CPP-P^3.

3 Computational experiments

This section is dedicated to an experimental assessment of CPP-P^3. For this purpose, we show extensive computational results obtained by CPP-P^3 on a large collection of benchmark instances. We also make a comparison with the current best performing algorithms published in the literature.

3.1 Benchmark instances and parameter settings

As the instances considered in CPP are complete graphs, the differences between instances only concern the value range and distribution of the edge weights. To give a comprehensive evaluation of our algorithm, we not only consider the instances from the aforementioned papers [4, 6, 28], but also introduce additional large instances. In total, we use 63 instances whose characteristics are listed below:

- Group I: a set of 7 instances which constitutes the benchmark reported in the literature in 2006 [6][2]. Instances named "rand100-100","rand300-100","rand 500-100" are generated by choosing a random integer for the edge weights in range [-100, 100], while the weights of "rand300-5" and "zahn300" respectively take value in [-5, 5] and {-1, 1}. To generate "sym300-50", 50 symmetric relations among 300 vertices were established, and the differences between related and unrelated components were used to compute the edge weights. To create the last instance named "regnier300-50", 50 bipartitions of 300 vertices were established and the difference between the number of bipartitions for which each pair of vertices is or is not in the same cluster was used to obtain the edge weights.
- Group II: a set of 6 instances originally proposed in 2009 [4] ("rand200-100", "rand400-100","rand100-5", "rand200-5", "rand400-5", "rand500-5")[3]. For these graphs, (integer) edge weights are uniformly generated in range [-100,100] or [-5, 5].
- Group III: a set of 35 instances reported in 2014 [28][4]. These instances are grouped into 4 categories by the number of vertices. Edge weights are also uniformly distributed in range [-5,5] or [-100, 100].
- Group IV: a set of 15 additional instances specifically generated for this study[5]. We provide a first set of 5 instances with 500 vertices, where edge

---

[2] Available at `http://www.infres.enst.fr/~charon/partition/`

[3] Available at `http://mailer.fsu.edu/~mbrusco/`

[4] Available at `http://www.proin.ktu.lt/~gintaras/`

[5] Available at `http://www.info.univ-angers.fr/pub/hao/cpp.html`

Table 1 Parameter settings

Parameters	§	Description	Values	Range
$tbase$	2.6	Tabu tenure base	15	$\{7, 10, 15, 20\}$
L	2.6	Maxium consecutive non-improving iterations	$|V|$	$\{0.5|V|, |V|, 1.5|V|, 2|V|\}$
α	2.7	Lower limit of perturbation strength	0.2	$\{0.1, 0.2, 0.3\}$
β	2.7	Upper limit of perturbation strength	0.5	$\{0.4, 0.5, 0.6\}$
$Maxrcl$	2.7	Maximum length of RCL	10	$\{7, 10, 15, 20\}$

weights are generated thanks to a Gaussian distribution $\mathcal{N}(0, 5^2)$ (the prefix of the instances name is "gauss"). Moreover, we provide another set of 10 large instances involving graphs of 700 and 800 vertices, while the edge weights are uniformly distributed in range [-5, 5] (the prefix of the instances name is "unif").

CPP-P^3 involves 5 parameters whose settings indicated in Table 1 are used for all our experiments. In order to estimate an appropriate value for these parameters, we first fixed a set of possible values for each parameter as indicated in column *Range*. Experiments with all these values have been made on a preliminary set of instances; values which achieved a good compromise in terms of best and average objective values as well as CPU time have been kept. Of course, fine-tuning these parameters would allow us to obtain better results. However, as we show in this section, the adopted parameter values of Table 1 lead already to highly competitive results.

3.2 Experiments and comparison

The CPP-P^3 algorithm was implemented in C++[6], compiled by GNU g++ and run on 2.82GHz Xeon CPU with 2 GB RAM. The experiments were conducted on a computer with an AMD Opteron 4184 processor (2.8GHz and 2GB RAM) running Ubuntu 12.04. When solving the DIMACS machine benchmarks[7] without compilation optimization flag, the run time on our machine is 0.40, 2.50 and 9.55 seconds respectively for graphs r300.5, r400.5 and r500.5. We used the CPU clock as the stop condition of CPP-P^3. In the following experiments, in order to get relative stable results, we fixed cut-off times which refer to the order (i.e., the number of vertices) of the graphs. Each instance was solved 10 times independently using different random seeds.

As mentioned in the introduction, ITS [28] and SGVNS [3] are the most recent and also the best performing algorithms among all previously developed heuristics. Consequently, we used ITS and SGVNS as the main references for our comparative study. To make a fair comparison between the three algorithms, we ran them under the same conditions, i.e. the same cut-off time for each instances and the same testing platform. The source code of ITS is publicly available from `http://www.proin.ktu.lt/~gintaras/`, and the code of SGVNS was kindly provided by the authors of [3].

---

[6] The source code is available at: `http://www.info.univ-angers.fr/pub/hao/cpp.html`.

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

3.2.1 Computational results on instances of Groups I and II

Table 2 shows the computational results of three algorithms on Group I and Group II instances. Column 1 and 2 provide instance name and best known objective value f_{prev} which are extracted from [4,6]. The same time limits for each algorithm are fixed as follows: 200, 500, and 1000 seconds for graphs with vertices in range [1,200], [201,300], and [301,500] respectively.

For each instance and each algorithm (CPP-P^3, ITS and SGVNS), we report the best objective value attained f_{best} over 10 runs, the average objective value f_{avg}, the frequency *hit* of reaching f_{best} over 10 runs, as well as the average CPU *time* (in seconds) for finding the best value.

In Table 2, one observes that in terms of solution quality, CPP-P^3 is able to hit all the previous best results while both ITS and SGVNS fail on the two largest instances rand500-5 and rand500-100. While comparing the performance robustness over 10 runs, we note that the average objective value of CPP-P^3 is better than the competing algorithms on all instances. Moreover, CPP-P^3 consistently reaches the previously best-known f_{prev} at each of 10 runs on 11 instances over 13, contrary to the ITS (7/13) and SGVNS (7/13). Finally, CPP-P^3 spends less average time than the other two algorithms to hit the best solutions. This experiment indicates that CPP-P^3 dominates the two references algorithms on instances from Groups I and II.

Table 2 Computation results on instances of Groups I and II of our CPP-P³, ITS [28] and SGVNS algorithms [3]

Instance	f_{prev}	CPP-P³				ITS				SGVNS			
		f_{best}	f_{avg}	hit	$time$	f_{best}	f_{avg}	hit	$time$	f_{best}	f_{avg}	hit	$time$
rand100-5	1407	1407	1407.00	10/10	0.05	1407	1407.00	10/10	0.10	1407	1407.00	10/10	0.54
rand100-100	24296	24296	24296.00	10/10	0.74	24296	24296.00	10/10	0.90	24296	24296.00	10/10	1.87
rand200-5	4079	4079	4079.00	10/10	6.57	4079	4079.00	10/10	17.80	4079	4078.80	9/10	31.30
rand200-100	74924	74924	74924.00	10/10	20.33	74924	74924.00	10/10	23.80	74924	74924.00	10/10	94.39
rand300-5	7732	7732	7732.00	10/10	36.01	7732	7730.50	5/10	169.50	7732	7731.40	8/10	231.48
rand300-100	152709	152709	152709.00	10/10	5.18	152709	152709.00	10/10	14.50	152709	152709.00	10/10	39.51
sym300-50	17592	17592	17592.00	10/10	69.53	17592	17590.00	9/10	101.00	17592	17591.40	9/10	438.21
regnier300-50	32164	32164	32164.00	10/10	0.90	32164	32164.00	10/10	1.20	32164	32164.00	10/10	1.56
zahn300	2504	2504	2504.00	10/10	6.21	2504	2504.00	10/10	15.90	2504	2504.00	10/10	32.69
rand400-5	12133	12133	12133.00	10/10	94.33	12133	12130.30	7/10	430.00	12133	12133.00	10/10	677.03
rand400-100	222757	222757	222757.00	10/10	170.32	222757	222690.80	5/10	356.90	222757	222725.20	8/10	522.16
rand500-5	17127	17127	17126.50	9/10	258.50	*17114*	17104.90	2/10	608.70	17127	17124.10	8/10	793.90
rand500-100	309125	309125	308985.50	4/10	205.39	*309007*	308860.90	1/10	406.70	*308984*	308860.30	1/10	1320.87

3.2.2 Computational results on instances of Group III

Table 3 shows comparative results among CPP-P^3, ITS and SGVNS on the instances of Group III. The same information as in Table 2 is provided. The f_{best} entries which are superior to f_{prev} and inferior to f_{prev} are marked respectively in bold and italic. Moreover, a star indicates a stricly f_{best} value among the three algorithms. Larger instances with more than 1000 vertices are included. The time limit was set to 1000, 2000, 4000 and 10000 seconds for instances with 500, 1000, 1500 and 2000 vertices respectively. Note that these time limits are shorter than those used in [28].

Table 3 discloses that CPP-P^3 outperforms ITS and SGVNS on instances of type 'p500' (i.e., with $|V| = 500$) both in terms of solution quality and computation time. We observe that CPP-P^3 reaches every f_{prev} while ITS and SGVNS fail respectively on 1 and 5 instances. Note that the best-known objective value of two instances (p500-5-3 and p500-100-2) is improved by CPP-P^3 and ITS. Another noticeable point is that CPP-P^3 achieves a 100% successful rate (*hit*) on 12 instances over 20, while considering ITS and SGVNS the corresponding robustness indicator is often low and never maximal on any instance. One also observes that the average time of CPP-P^3 is always shorter except slightly longer on p500-100-4 comparing to SGVNS.

Larger instances (with $|V| \geq 1000$) are unsurprisingly much difficult to tackle in a reasonable time limit, as confirmed by the less robust results: each algorithm hits the respective best solution only one or two times over 10 runs under the given time limit. The essential point is that all f_{prev} are improved by CPP-P^3 and SGVNS. for these two algorithms even their average scores f_{avg} are better than the previous best-known values. ITS also improves f_{prev} on three instances (p1000-3, p1000-4 and p2000-3), but systematically less than CPP-P^3 and SGVNS. Comparing the f_{best} (resp. f_{avg}) entries, CPP-P^3 reached better solutions on 7 (resp. 10) instances, and SGVNS on 8 (resp. 5). Contrary to the previous sets of instances where CPP-P^3 was the most powerful algorithm, it appears that on this particular set, CPP-P^3 and SGVNS perform similarly.

Table 3 Computation results on instances of Group III of our CPP-P³, ITS [28] and SGVNS algorithms [3]

Instance	f_{prev}	CPP-P³				ITS				SGVNS			
		f_{best}	f_{avg}	hit	time	f_{best}	f_{avg}	hit	time	f_{best}	f_{avg}	hit	time
p500-5-1	17691	17691	17690.10	9/10	140.32	17691	17683.30	2/10	864.08	17680	17673.40	4/10	562.20
p500-5-2	17169	17169	17168.10	7/10	300.22	17169	17152.90	1/10	406.80	17166	17154.70	1/10	705.73
p500-5-3	16815	**16816**	16815.20	3/10	326.05	**16816**	16812.50	1/10	623.80	16810	16805.10	1/10	1845.85
p500-5-4	16808	16808	16808.00	10/10	141.59	16808	16790.90	2/10	473.00	16808	16800.60	6/10	598.65
p500-5-5	16957	16957	16957.00	10/10	99.00	16957	16948.50	6/10	595.00	16957	16943.80	2/10	446.62
p500-5-6	16615	16615	16615.00	10/10	185.04	16615	16606.80	3/10	482.20	16615	16609.00	3/10	481.48
p500-5-7	16649	16649	16648.80	9/10	285.55	16649	16632.70	1/10	547.80	*16647*	16632.20	1/10	314.10
p500-5-8	16756	16756	16756.00	10/10	156.56	16756	16752.30	4/10	376.90	16756	16748.30	9/10	590.98
p500-5-9	16629	16629	16629.00	10/10	161.42	16619	16607.60	2/10	526.50	16629	16623.80	5/10	549.73
p500-5-10	17360	17360	17360.00	10/10	51.52	17360	17356.80	8/10	547.40	17360	17359.80	8/10	430.33
p500-100-1	308896	308896	308896.00	10/10	317.40	308896	308810.90	2/10	481.90	308896	308890.60	4/10	741.65
p500-100-2	310163	**310241**	310225.40	8/10	367.47	**310241**	309843.40	1/10	530.50	**310241**	310170.80	1/10	449.71
p500-100-3	310477	310477	310471.10	9/10	267.23	310477	310028.60	2/10	555.10	*310418*	310259.20	3/10	690.50
p500-100-4	309567	309567	309528.60	9/10	376.93	*309494*	309193.80	1/10	412.00	309567	309411.60	1/10	359.16
p500-100-5	309135	309135	309116.90	9/10	286.63	309135	308898.60	2/10	639.20	309135	309116.90	9/10	897.59
p500-100-6	310280	310280	310280.00	10/10	251.93	310280	309973.70	6/10	482.60	310280	310218.50	3/10	353.39
p500-100-7	310063	310063	310063.00	10/10	153.16	310063	310029.40	5/10	479.90	310063	309992.70	3/10	342.03
p500-100-8	303148	303148	303148.00	10/10	284.58	303148	302830.10	5/10	580.00	303148	302774.30	5/10	737.08
p500-100-9	305305	305305	305305.00	10/10	81.30	305305	305236.40	8/10	497.50	305305	305299.80	9/10	623.31
p500-100-10	314864	314864	314864.00	10/10	43.99	314864	314815.30	5/10	392.90	314864	314818.20	5/10	492.15
p1000-1	883359	*884970	884348.68	1/10	1221.80	*881855*	879382.10	1/10	1215.10	*885016	884126.10	2/10	2125.72
p1000-2	879792	881149	880131.87	1/10	1212.22	*878854*	875849.90	1/10	1218.00	*881751	880189.30	1/10	3424.15
p1000-3	862969	*866415	864718.62	1/10	1058.54	**864309**	860987.90	1/10	1376.00	**866041**	864874.20	1/10	809.87
p1000-4	865754	868573	867295.37	1/10	885.94	**865866**	862984.70	1/10	1207.20	*869374	867669.00	2/10	1347.60
p1000-5	887314	888678	887997.00	1/10	1257.44	*887066*	883768.50	1/10	876.50	*888720	887783.00	1/10	3487.01
p1500-1	1614791	1616999	1614847.50	1/10	2711.04	*1608761*	1604753.00	1/10	2489.00	*1618281	1615019.10	1/10	3300.02
p1500-2	1642442	*1648800	1646916.12	1/10	2313.66	*1636856*	1631070.60	1/10	2643.40	**1647557**	1644372.50	1/10	4769.19
p1500-3	1600857	*1609854	1607426.12	1/10	2991.33	*1600451*	1595268.00	1/10	2977.80	**1608877**	1605009.60	1/10	2328.56
p1500-4	1633081	1639528	1637375.12	1/10	1867.77	*1628349*	1626165.90	1/10	2552.20	*1640643	1636222.90	1/10	1091.37
p1500-5	1585484	1592732	1590291.25	1/10	1791.04	*1580965*	1575690.70	1/10	2585.80	*1593518	1591043.60	1/10	3567.00
p2000-1	2489880	*2505359	2500050.00	1/10	5238.93	*2487048*	2480720.40	1/10	7710.90	*2501962	2498417.40	1/10	3127.15
p2000-2	2479127	**2490104**	2488016.75	1/10	5997.60	*2473691*	2464960.90	1/10	6750.60	*2492662	2485936.90	1/10	4925.60
p2000-3	2527119	*2540063	2536091.50	1/10	6990.83	**2528777**	2517090.50	1/10	7639.30	**2538196**	2535414.30	1/10	1731.34
p2000-4	2523884	*2525903	2521449.00	1/10	5992.97	*2509080*	2502059.40	1/10	6607.00	**2522156**	2516375.90	1/10	2730.68
p2000-5	2499690	*2508729	2504678.00	1/10	5022.83	*2496036*	2487416.50	1/10	7239.00	**2506784**	2503396.90	1/10	1945.23

3.2.3 Computational results on instances of Group IV

As Group IV instances are newly created, no previous results are available. To obtain reasonable reference values f_{prev}, each instance is first solved by CPP-P^3 for 8 hours and the objective value of the best solution found is set as f_{prev}. Then, we run CPP-P^3, ITS and SGVNS on all the instances of this group under a cutoff limit of 1000 seconds. The computational results are given in Table 4.

We first examine the *uniform* graphs ('unif700' and 'unif800'). Recall that 1000 seconds are sufficient for CPP-P^3 to reach a stable objective value for random instances of size 500 (Group III, see table 3). Here, one can note that *hit* rates are decreasing when the number of vertices grows to 700 and 800. Nevertheless, CPP-P^3 and SGVNS are able to hit f_{prev} at least once for these 10 instances while ITS fails to match this performance.

The edge weights of the last five instances (of type "gauss500") are generated according to a Gaussian distribution. Comparing the results with the instances of Group III (p500-100-1 to p500-100-10), we observe that the *hit* and *time* indicators, for each algorithm, are similar. This may imply that the distribution has no significant influence on the performance of these local search based algorithms.

Table 4 Computation results on new instances of our CPP-P³, ITS [28] and SGVNS algorithms [3]

Instance	f_{prev}	CPP-P³				ITS				SGVNS			
		f_{best}	f_{avg}	hit	time	f_{best}	f_{avg}	hit	time	f_{best}	f_{avg}	hit	time
unif700-100-1	515016	515016	514768.40	6/10	768.90	514689	513368.50	1/10	965.40	515016	514291.50	2/10	322.68
unif700-100-2	519441	519441	518815.20	6/10	571.39	519141	517377.70	1/10	1055.60	519441	518265.40	3/10	2673.35
unif700-100-3	512351	512351	511457.70	3/10	749.91	512351	510002.50	1/10	1094.30	512351	511876.90	7/10	2033.99
unif700-100-4	513582	513582	513540.80	9/10	755.93	511772	510861.60	1/10	859.80	513582	513279.30	8/10	1328.62
unif700-100-5	510387	510387	510244.50	1/10	759.50	510234	509162.70	1/10	963.80	510367	509985.10	3/10	1026.23
unif800-100-1	639675	639675	639605.50	4/10	1033.06	639307	637927.40	1/10	879.90	639675	639373.70	3/10	1101.74
unif800-100-2	630704	630704	630488.00	2/10	1248.86	630088	628776.30	1/10	1203.50	630704	630034.00	2/10	882.15
unif800-100-3	629108	629108	628819.60	2/10	821.97	628130	627045.40	1/10	1267.70	629108	628740.50	1/10	3055.54
unif800-100-4	624728	624728	624153.20	1/10	1140.46	623905	622172.00	1/10	1130.60	624728	624165.50	2/10	2717.19
unif800-100-5	625905	625905	625378.40	1/10	1097.96	625066	623177.00	1/10	1336.60	625905	625355.90	2/10	639.25
gauss500-100-1	265070	265070	265049.10	9/10	410.31	265070	264741.50	1/10	554.10	265070	264882.40	2/10	964.81
gauss500-100-2	269076	269076	269039.20	7/10	469.05	269076	268815.20	2/10	512.60	269076	268953.10	4/10	967.99
gauss500-100-3	257700	257700	257467.40	3/10	343.48	257700	257205.30	1/10	485.20	257700	257444.50	2/10	871.46
gauss500-100-4	267683	267683	267633.70	9/10	251.65	267683	267266.90	4/10	424.10	267683	267589.50	4/10	458.60
gauss500-100-5	271567	271567	271567.00	10/10	94.08	271567	271485.40	5/10	576.80	271567	271539.60	7/10	638.76

Table 5 Comparison results of CPP-P^3-X and CPP-P^3

Instance	CPP-P^3-X					CPP-P^3				
	f_{best}	$f_{avg}(\sigma)$	hit	time	$iter_{avg}$	f_{best}	$f_{avg}(\sigma)$	hit	time	$iter_{avg}$
rand300-5	**7732**	**7732.0**(0.0)	10/10	64.29	26321	**7732**	**7732.0**(0.0)	10/10	33.82	33812
rand500-100	309007	308891.3(38.8)	1/10	131.60	15226	**309125**	**308935.6**(98.4)	2/10	259.65	21499
p500-5-3	**16815**	16814.2(0.4)	2/10	274.24	17011	**16815**	**16815.0**(0.0)	10/10	373.79	23100
gauss500-100-3	**257700**	257264.4(229.6)	1/10	500.59	13851	**257700**	**257558.6**(148.0)	3/10	245.04	21417
unif700-100-2	**519441**	518482.6(1026.3)	5/10	728.18	13919	**519441**	**519441.0**(0.0)	10/10	907.01	21289
unif800-100-4	624646	623951.8(383.2)	2/10	932.25	12099	**624728**	**624164.3**(372.0)	2/10	1001.97	15067
p1000-1	**884861**	883068.0(1133.7)	1/10	1111.76	6504	884709	**883565.0**(854.1)	1/10	921.53	9970
p1500-3	1607199	1604345.2(2026.0)	1/10	1595.41	4234	**1608549**	**1604854.2**(1568.4)	1/10	2549.95	4051
p2000-1	2504261	2489626.0(6067.3)	1/10	6593.32	1536	**2506312**	**2500859.4**(3420.5)	1/10	6256.35	9875
p2000-4	2517390	2510779.3(4151.5)	1/10	6405.45	1363	**2522665**	**2519964.9**(1835.7)	1/10	5761.96	4008

4 Analysis

4.1 The effectiveness of Top Move based neighborhood

One of the most crucial features of a local search algorithm is the definition of its neighborhood and the selection criterion used. Contrary to the previous algorithms like [4,6,8,28], CPP-P^3 is based on a different neighborhood definition. Indeed, while other methods consider all the possible moves for each vertex (see $N(s)$ in Section 2.2), CPP-P^3 only considers a restricted neighborhood defined by the *Top Move* of each vertex (see $N'(s)$ in Section 2.2).

To evaluate the most accurate strategy, we defined another algorithm, CPP-P^3-X which replaces $N'(s)$ in CPP-P^3 by the complete neighborhood $N(s)$ and keeps the other elements of CPP-P^3 unchanged. Then, we selected 10 instances of different sizes from the benchmarks (see Table 5) and ran CPP-P^3-X as well as CPP-P^3 under the same conditions as described in Section 3.2. To give a comprehensive comparison between the two methods, we report in Table 5, for each instance, the best objective value f_{best} obtained over 10 runs, the average objective value f_{avg} with the standard deviation σ, and the average *time* needed to attain a best objective value. We define one pass of three phases of the algorithms as one iteration, and the average number of iterations over 10 runs is given in column $iter_{avg}$.

Experimental results show that CPP-P^3 globally outperforms CPP-P^3-X. One can observe that CPP-P^3 completes more iterations than CPP-P^3-X for the same time limit, which means that CPP-P^3 spends shorter times to find an appropriate move during one iteration. If we compare the results of CPP-P^3-X with the results of the ITS algorithm reported in the last section, one finds that even CPP-P^3-X is more efficient than ITS in reaching better solutions on large instances. This highlights the usefulness of our three phase approach.

To illustrate the convergence rate of both algorithms, we ran CPP-P^3 and CPP-P^3-X on two instances and record the best objective value f_{best} after every 10^4 reallocations. The comparative convergence can be visualized in Figure 1. One observes that using the same computation times, CPP-P^3-X finds a better solution than CPP-P^3 at first, but CPP-P^3 finally exceeds CPP-P^3-X and remains superior to CPP-P^3-X in the following steps. This indicates
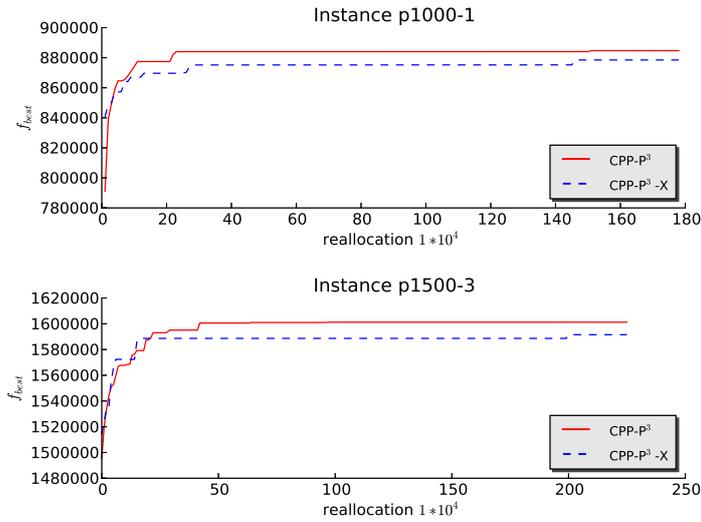
Fig. 1 Running profiles of CPP-P^3 and CPP-P^3-X

that CPP-P^3 has a stronger ability to reach better solutions after experiencing the same number of reallocations.

4.2 Landscape analysis

In order to obtain some information about the difficulty of the CPP instances, we carried out a landscape analysis based on the fitness distance correlation (FDC) [22]. Such an analysis could shed lights on the behavior of the experimented algorithms. FDC estimates how closely the fitness and distance to the nearest global optimum are related. For a maximization problem, if the fitness improves when the distance to the optimum decreases, then the search is expected to be effective as there is a "path" to the optimum via solutions with increasing fitness. The correlation coefficient $\rho_{pdf} \in [-1, 1]$ measures the correlation strength, and the perfect ρ_{pdf} value will be -1 for maximization problems, while for minimization problems, the ideal ρ_{pdf} will be 1.

For this study, we investigated several representative instances: sym300-50, regnier300-50, rand500-100, p500-5-3, p500-5-5, p500-100-6, gauss500-100-3, gauss500-100-4. For each graph, we ran CPP-P^3 and collect 5000 high quality local optimum solutions. The distances between these local optima to the global optimum (in our case, the best local optimum) are computed according to the following definition.

Definition 2 Let $s_1 = \{G_1, G_2, ..., G_k\}$, $s_2 = \{H_1, H_2, .., H_l\}$ be two solutions (partitions) of graph $G = \{V, E, W\}$. The Rand Index [29] computes a distance

between s_1, s_2:

$$d(s_1, s_2) = \frac{\sum_{e \in E} d_e(s_1, s_2)}{|E|} \tag{8}$$

while $d_e(s_1, s_2)$ of edge e_{uv} is defined by:

$$d_e(s_1, s_2) = \begin{cases} 1, if \ \exists G_i \in s_1, \exists H_j \in s_2, and \ e \in G_i, e \in H_j \\ \quad or \ if \ \forall G_i \in s_1, \neg(e \in G_i) \ and \ \forall H_j \in s_2, \neg(e \in H_j) \\ 0, otherwise \end{cases} \tag{9}$$

The correlation between fitness (objective function value) and distance to the reference solution can be visualized in Figure 2. One observes that the instances on the left side of the figure have weak FD correlations as indicated by ρ_{pdf} values close to 0. On the other hand, the instances on the right side have stronger FD correlations. It is interesting to see that the correlation strength is proportional to the efforts of our algorithm to reach the best optimum, i.e., CPP-P^3 needs more time to solve weakly correlated instances contrary to strongly related instances. For example, Table 2 indicates that CPP-P^3 only needs 0.90s to reach the best local optimum on instance regnier300-50 (with a strong correlation $\rho_{pdf} = -0.89$) on average while 69.53s on sym300-50 (with a weak correlation $\rho_{pdf} = -0.12$). Although we did not include fitness-distance plots for all the instances, this observation suggests that ρ_{pdf} helps us estimate the performance of CPP-P^3 on a particular instance.

4.3 Impact of the descent search phase of CPP-P^3

As shown in Section 2, the first phase of CPP-P^3 employs a descent procedure to locate a first local optimum from a given starting solution (which is typically generated by the perturbation phase). Since the descent phase is followed by a tabu-based exploration phase, one may wonder if the descent phase is necessary. To clarify this, we investigated a CPP-P^3 variant where the descent search phase was disabled (i.e., line 6 of Algorithm 1 was removed) and the other components were kept unchanged. We then used the variant to solve all the benchmark instances under the same experimental condition as that used in Section 3.2. Without bothering to give a detailed tabulation of the detailed results, we mention that, between CPP-P^3 and its variant, no strict dominance is observed according to the main performance indicators (best and average objective values, hit and CPU time). In fact, CPP-P^3 performs better than its variant on a number of instances while the reverse is true for other instances. This experiment indicates that it would be more appropriate to consider the descent search phase as an option of the CPP-P^3 which can be switched on or off to solve a given problem instance.
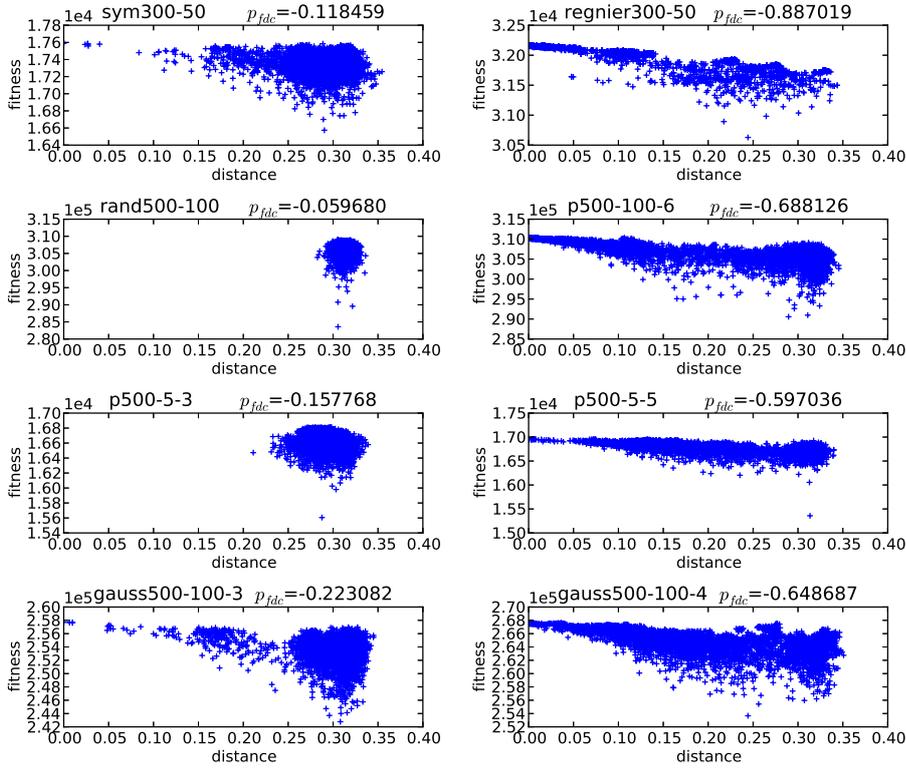
Fig. 2 FD correlation plots with respect to the solution fitness and distance to the optimum for 8 graphs.

5 Conclusion

In this paper, we proposed an effective heuristic algorithm, CPP-P^3, to solve the clique partitioning problem. The algorithm is composed of three iterated search phases: a descent search, an exploration search and a directed perturbation. The descent search quickly converges from a starting solution to a local optimum. The exploration search uses a tabu procedure to explore nearby optimum solutions. The directed perturbation creates an effective diversification with a mechanism similar to a GRASP construction process. The originality of the neighborhood search comes from the concept of *Top Move*, which allows the algorithm to reduce drastically the number of considered neighbors.

To verify the effectiveness of our CPP-P^3 algorithm, we evaluated it on a large number of CPP benchmark instances from the literature as well as large random instances specifically generated for this study. We also made a

comprehensive comparison with the most recent and the best performing CPP algorithms available in the literature (ITS and SGVNS). Experimental results showed that CPP-P^3 dominates both ITS and SGVNS in terms of solution quality, computational time and robustness on a large set of graphs. On large instances (i.e., graphs with 1000 vertices and more), CPP-P^3 and SGVNS obtain comparable results (and dominate ITS) although improvements are clearly possible since we observe a loss of robustness for all algorithms on these difficult instances. Note that best-known solutions of some large instances have been improved by our algorithm.

We also provided an analysis of the *Top Move* neighborhood to assess its key role to the performance of CPP-P^3. Moreover, we presented a landscape analysis using the fitness distance correlation to shed lights on the instance characteristics and hardness.

Although the proposed CPP-P^3 algorithm performs well on problem instances with up to 500 vertices, larger instances (say with more than 1000 vertices) are really challenging for CPP-P^3 as well as other existing CPP methods. To obtain improved results on these large instances, it would be useful to investigate other local search operators and hybrid paradigms. One promising direction concerns the population-based memetic framework [19,26] which has proved to be quite successful for solving some challenging graph partition and grouping problems [1,13,14,15]. For this purpose, it would be particularly interesting to design meaningful crossover operators able to recombine the building blocks of the clique partitioning problem.

Acknowledgments

References

1. U. Benlic and J.K. Hao. A multilevel memetic approach for improving graph k-partitions. *IEEE Transactions on Evolutionary Compututation*, 15(5):624–642, 2011.
2. U. Benlic and J.K. Hao. Breakout local search for the quadratic assignment problem. *Applied Mathematics and Computation*, 219(9):4800–4815, 2013.
3. J. Brimberg, S. Janićijević, and N. Mladenović and D. Urošević. Solving the clique partitioning problem as a maximally diverse grouping problem. *Optimization Letters*, DOI:10.1007/s11590-015-0869-4, 2015.
4. M.J. Brusco and H.F. Köhn. Clustering qualitative data based on binary equivalence relations: Neighborhood search heuristics for the clique partitioning problem. *Psychometrika*, 74(4):685–703, 2009.

5. I. Charon and O. Hudry. The noising methods: A generalization of some metaheuristics. *European Journal of Operational Research*, 135(1):86–101, 2001.
6. I. Charon and O. Hudry. Noising methods for a clique partitioning problem. *Discrete Applied Mathematics*, 154(5):754–769, 2006.
7. Y. Chen and J.K. Hao. Iterated responsive threshold search for the quadratic multiple knapsack problem. *Annals of Operations Research*, 226(1):101-131, 2015.
8. S.G. De Amorim, J.P. Barthélemy, and C.C. Ribeiro. Clustering and clique partitioning: Simulated annealing and tabu search approaches. *Journal of Classification*, 9(1):17–41, 1992.
9. U. Dorndorf, F. Jaehn, and E. Pesch. Modelling robust flight-gate scheduling as a clique partitioning problem. *Transportation Science*, 42(3):292–301, 2008.
10. U. Dorndorf and E. Pesch. Fast clustering algorithms. *ORSA Journal on Computing*, 6(2):141–153, 1994.
11. T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, 1995.
12. Z.H. Fu and J.K. Hao. A three-phase search approach for the quadratic minimum spanning tree problem. *Engineering Applications of Artificial Intelligence*, 46: 113-130, 2015.
13. Y. Jin, J.K. Hao, and J.P. Hamiez. A memetic algorithm for the minimum sum coloring problem. *Computers & Operations Research*, 43(3): 318–327, 2014.
14. P. Galinier and J.K. Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379–397, 1999.
15. P. Galinier, Z. Boujbel, and M.C. Fernandes. An efficient memetic algorithm for the graph partitioning problem. *Annals of Operations Research*, 191(1):1-22, 2011.
16. F. Glover and M. Laguna. *Tabu Search*. Springer, 1997.
17. M. Grötschel and Y. Wakabayashi. A cutting plane algorithm for a clustering problem. *Mathematical Programming*, 45(1-3):59–96, 1989.
18. M. Grötschel and Y. Wakabayashi. Facets of the clique partitioning polytope. *Mathematical Programming*, 47(1-3):367–387, 1990.
19. J.K. Hao. Memetic algorithms in discrete optimization. In F. Neri, C. Cotta and P. Moscato (Eds.) Handbook of Memetic Algorithms. Studies in Computational Intelligence 379, Chapter 6, pages 7394, 2012.
20. F. Jaehn and E. Pesch. New bounds and constraint propagation techniques for the clique partitioning problem. *Discrete Applied Mathematics*, 161(13-14):2025–2037, 2013.
21. X. Ji and J.E. Mitchell. Branch-and-price-and-cut on the clique partitioning problem with minimum clique size requirement. *Discrete Optimization*, 4(1):87–102, 2007.
22. T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 184–192. Morgan Kaufmann Publishers Inc., 1995.
23. B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.
24. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
25. H.R. Lourenço, O.C. Martin, and T. Stützle. Iterated local search. *Handbook of Metaheuristics* 57:320–353. Kluwer Academic Publishers, 2003.
26. P. Moscato and C. Cotta. A Gentle Introduction to memetic algorithms. In F. Glover and G. A. Kochenberger (Eds.), Handbook of Metaheuristic. Kluwer, Norwell, Massachusetts, USA, 2003.
27. M. Oosten, J.H.G.C. Rutten, and F.C.R. Spieksma. The clique partitioning problem: facets and patching facets. *Networks*, 38(4):209–226, 2001.
28. G. Palubeckis, A. Ostreika, and A. Tomkevičius. An iterated tabu search approach for the clique partitioning problem. *The Scientific World Journal*, 2014:353101, 2014.
29. W.M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.
30. Y. Wakabayashi. *Aggregation of Binary Relations: Algorithmic and Polyhedral Investigations*. PhD thesis, Universität Ausburg, 1986.
31. H. Wang, B. Alidaee, F. Glover, and G. Kochenberger. Solving group technology problems via clique partitioning. *International Journal of Flexible Manufacturing Systems*, 18(2):77–97, 2006.

32. Q. Wu and J.K. Hao. An adaptive multistart tabu search approach to solve the maximum clique problem. *Journal of Combinatorial Optimization*, 26(1):86-108, 2013.
33. Q. Wu and J.K. Hao. A hybrid metaheuristic method for the maximum diversity problem. *European Journal of Operational Research*, 231(2):452-464, 2013.