mathematics

MDPI

Article

BARRAKUDA: A Hybrid Evolutionary Algorithm for Minimum Capacitated Dominating Set Problem

Pedro Pinacho-Davidson [1] **and Christian Blum** [2,*]

[1] Department of Computer Science, Faculty of Engineering, Universidad de Concepción, Concepción 4070409, Chile; ppinacho@udec.cl

[2] Artificial Intelligence Research Institute (IIIA-CSIC), Campus of the UAB, 08193 Bellaterra, Spain

* Correspondence: christian.blum@iiia.csic.es

Received: 14 September 2020; Accepted: 15 October 2020; Published: 23 October 2020

Abstract: The minimum capacitated dominating set problem is an NP-hard variant of the well-known minimum dominating set problem in undirected graphs. This problem finds applications in the context of clustering and routing in wireless networks. Two algorithms are presented in this work. The first one is an extended version of construct, merge, solve and adapt, while the main contribution is a hybrid between a biased random key genetic algorithm and an exact approach which we labeled BARRAKUDA. Both algorithms are evaluated on a large set of benchmark instances from the literature. In addition, they are tested on a new, more challenging benchmark set of larger problem instances. In the context of the problem instances from the literature, the performance of our algorithms is very similar. Moreover, both algorithms clearly outperform the best approach from the literature. In contrast, BARRAKUDA is clearly the best-performing algorithm for the new, more challenging problem instances.

Keywords: minimum capacitated dominating set problem; hybrid evolutionary algorithm; biased random key genetic algorithm

1. Introduction

Dominating set problems are difficult combinatorial optimization problems from the family of set covering problems. Over the last two decades, they have attracted research interests due to their applications to both clustering and routing in wireless networks [1–3]. The most basic dominating set problem is the classical minimum dominating set (MDS) problem. Given an undirected graph G with vertex set V (where $|V| = n$), a set $D \subseteq V$ is called a *dominating set* if and only if each vertex $v \in V$ either forms part of D or has at least one neighbor v', which is a member of D. In the latter case, we say that v' dominates v. Each dominating set is a feasible solution to the MDS problem. The optimization goal of the MDS problem is to find a smallest dominating set in G. In the case of the MDS problem, given a valid solution D, each vertex $v \in D$ is said to dominate all its neighbors that do not form part of D themselves.

1.1. Literature Review for the CapMDS Problem

In this work, we focus on an NP-hard variant of the MDS problem known as the minimum capacitated dominating set (CapMDS) problem. The difference to the MDS problem is that the CapMDS problem restricts the number of vertices that each vertex from a solution D can dominate. The CapMDS problem is relevant in the field of wireless communications, and for this reason, several approaches have been found in the literature in recent years. Even though the CapMDS problem has been demonstrated to be NP-hard [4], some researchers have focused on the development of exact approaches. Cygan et al. [5] presented an algorithm based on maximum matching that runs in $\mathcal{O}(1.89^n)$

time. The performance of the algorithm was further improved by considering dynamic programming over subsets [6] with a time complexity of $\mathcal{O}(1.8463^n)$. A distributed approximation scheme was developed in [7]. This algorithm achieves a $\mathcal{O}(\log\triangle)$-approximation in $\mathcal{O}(\log^3 n + \log(n)/\epsilon)$ time, where n represents the number of vertices and $\triangle$ denotes their maximal degree. Finally, the CapMDS problem has been subject to a few research studies focused on heuristics and metaheuristic approaches. Potluri and Singh [8] carried out a performance comparison between three greedy heuristics. They showed that the best heuristic is the one that selects, at each step, the vertex that maximizes the minimum between the vertex capacity and the number of uncovered neighbors. The neighboring vertices dominated by this vertex (in case the neighborhood is larger than the capacity of the vertex) are chosen randomly. This greedy heuristic has been used as the basis for the design of two metaheuristic approaches: one based on ant colony optimization (named ACO) and the other one based on genetic algorithms (named SGA) [9]. Lately, Li et al. [10] developed an iterated local search approach labeled LS_PD, showing a significantly better performance than ACO and SGA when applied to general graphs with uniform and variable capacity. LS_PD adopts a penalization strategy in the context of the vertex-scoring scheme. Moreover, it makes use of a two-mode dominated vertex selection strategy taking into account both random and greedy decisions for the choice of the neighbors that a chosen vertex should dominate. This is done for the purpose of achieving a balance between the intensification and the diversification of the search process. Recently, Pinacho-Davidson et al. [11] developed a construct, merge, solve and adapt (CMSA) approach for the CapMDS. CMSA is a hybrid proposal that integrates a heuristic algorithm and an exact solver to accelerate the solution process, especially thought for the application to large-scale problem instances. Moreover, they applied—for the first time—a high-performance integer linear programming (ILP) solver, namely CPLEX, to all problem instances from the literature. They showed that both CMSA and CPLEX outperform LS_PD.

1.2. Literature Review on Related Problems

Apart from the CapMDS, various other computationally hard variants of the MDS problem can be found in the related literature. One of them is the so-called minimum connected dominating set (MCDS) problem, which has the additional restriction that a solution D must induce a connected sub-graph of the input graph. Recent algorithmic approaches for the MCDS problem include different local search algorithms [12,13] and a hybrid algorithm combining ant colony optimization with reduced variable neighborhood search [14]. Another example of an extension of the classical MDS problem is the minimum independent dominating set problem in which a solution D must be an independent set of the input graph in addition to being a dominating set. Recent algorithmic approaches tailored to this problem include a two-phase removing algorithm [15] and a memetic algorithm [16]. The node-weighted versions of these problems are also often considered. The latest algorithms for the minimum weight (vertex) independent dominating set problem include a local search approach that makes use of a reinforcement learning based repair procedure [17] and a memetic algorithm [18]. Next, it is worth mentioning the so-called minimum total dominating set problem, which was solved by means of a hybrid evolutionary algorithm in [19]. Finally, it is also worth mentioning that high-quality algorithms for the CapMDS problem can also be useful for solving problems, such as the capacitated vertex k-center problem; see [20].

1.3. Literature Review on Similar Techniques

In this paper we develop heuristic algorithms for the CapMDS problem that are based on solving a sequence of reduced sub-instances of the original problem instance. Historically, this general idea was first been exploited in the context of problem decomposition by techniques from the field of Operations Research, such as Dantzig–Wolfe decomposition (column generation) and Benders decomposition (row generation) [21]. However, more recently, these ideas have also been applied in the context of heuristic optimization algorithms that make use of mathematical programming techniques, known as matheuristics [22]. Large neighborhood search (LNS) [23], respectively, very large-scale neighborhood

search [24], are among the most popular matheuristic techniques. These algorithms function as local searches. However, they make use of mathematical programming to find an improving solution of the incumbent solution at each iteration in a large-scale neighborhood of the incumbent solution. Many LNS approaches are based on the principle of ruin-and-recreate [25], also sometimes found as destroy-and-recreate or destroy-and-rebuild . Alternative ways of defining large neighborhoods are local branching [26], the corridor method [27], and POPMUSIC [28].

In this work we present two algorithms from the field of matheuristics for the CapMDS problem. The first one is a version of *construct, merge, solve and adapt* (CMSA) whose general idea was first presented in [29]. In CMSA, the sub-instance that is solved at each iteration by a mathematical programming technique is assembled from a set of heuristically constructed solution. In the second algorithm proposed in this work, BARRAKUDA, the intention is to enhance the idea of CMSA with the learning component present in evolutionary algorithms.

1.4. Organization of The Paper

The remainder of the paper is organized as follows. The considered optimization problem is introduced in Section 2, while the proposed algorithms are described in Section 3. Finally, a comprehensive experimental evaluation is provided in Section 4, and conclusions, as well as indications on future lines of work, are presented in Section 5.

2. The Capmds Problem

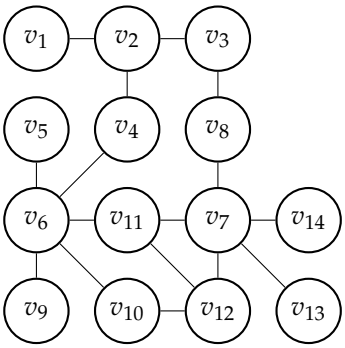
We recall some preliminary definitions before describing the CapMDS problem. Let $G = (V, E)$ be an undirected graph on a set of n vertices $V = \{v_1, v_2, \ldots, v_n\}$ and a set of edges E. We assume that G neither contains edges from a vertex $v \in V$ to itself (loops) nor multi-edges. Two vertices are neighbors (also called adjacent to each other) if and only if there exists an edge between them, that is, $v \in V$ and $u \in V$ are said to be neighbors if and only if $(v, u) \in E$. For a vertex v, let $N(v) := \{u \in V \mid (v, u) \in E\}$ denote the set of neighbors known as the *open neighborhood* (or simply *neighborhood*) of v in G. Furthermore, the *closed neighborhood* of a vertex $v \in V$, denoted by $N[v]$ and contains the vertices adjacent to v in addition to v itself, that is, $N[v] := N(v) \cup \{v\}$. The degree $\deg(v)$ of v is the cardinality of the set of neighbors of v, that is, $\deg(v) = |N(v)|$. As already mentioned above, a dominating set of G is a (sub)set $S \subseteq V$ such that each vertex $v \in V \setminus S$ must be adjacent to at least one vertex in S. Each vertex in S is called a *dominator*. Otherwise, it is called *dominated*. A dominator dominates itself and some or all of its neighbors.

A problem instance of the CapMDS problem is a tuple (G, Cap) that consists of an undirected graph $G = (V, E)$—without loops nor multi-edges—and a capacity function $Cap : V \to \mathbb{N}$. This function assigns a positive integer value $Cap(v)$ to each vertex $v \in V$ representing the maximum number of adjacent vertices this vertex is allowed to dominate. Any dominating set $S \subseteq V$ is a potential solution to the CapMDS problem. Such a dominating set S is a *valid solution*, if it is possible to identify a set $\{C(v) \mid v \in S\})$ that (1) contains for each dominator $v \in S$ the (sub-)set $C(v) \subseteq N(v) \setminus S$ containing those neighbors of v that are chosen to be dominated by v, and (2) fulfills the following conditions:

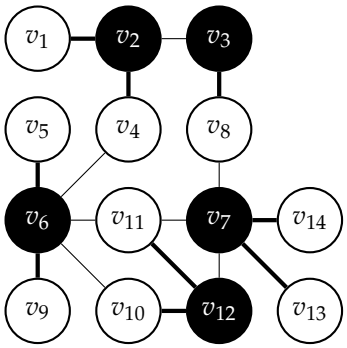
1. $S \cup (\bigcup_{v \in S} C(v)) = V$, that is, all vertices from V are either chosen to be a dominator, or are dominated by at least one dominator.
2. $|C(v)| \leq Cap(v)$ for all $v \in D^S$, that is, all chosen dominators $v \in D^S$ dominate at most $Cap(v)$ of their neighbors.

Finally, the objective function value (to be minimized) is defined as $f(S) := |S|$.

Figure 1 presents an illustrative example of the CapMDS problem. While Figure 1a displays an example graph, Figure 1b shows an optimal solution (black vertices) considering a uniform capacity of 2 for each vertex. Notice that the sets of dominated neighbors of each node in S are indicted by bold edges. Vertices v_5 and v_9, for example, form set $C(v_6)$. More specifically, $S = \{v_2, v_3, v_6, v_7, v_{12}\}$ and $C(v_2) = \{v_1, v_4\}, C(v_3) = \{v_8\}, C(v_6) = \{v_5, v_9\}, C(v_7) = \{v_{14}, v_{13}\}, C(v_{12}) = \{v_{11}, v_{10}\}$.



(**a**) CapMDS problem instance, considering a uniform capacity $Cap(v) = 2, \forall v \in V$.



(**b**) Solution $S = \{v_2, v_3, v_6, v_7, v_{12}\}$.

Figure 1. An example of the CapMDS problem. Note that the bold edges in (**b**) indicate the domination relations. Vertices v_1 and v_4, for example, are dominated by vertex v_2 which forms part of the solution.

An ILp Formulation for the Capmds Problem

The CapMDS can be modeled in terms of the following ILP model. [11]:

$$\textbf{minimize} \sum_{v_i \in V} x_i \tag{1}$$

$$\textbf{subject to} \sum_{v_j \in N(v_i)} y_{ji} \geq 1 - x_i \quad \forall v_i \in V \tag{2}$$

$$\sum_{v_j \in N(v_i)} y_{ij} \leq Cap(v_i) \quad \forall v_i \in V \tag{3}$$

$$y_{ij} \leq x_i \quad \forall v_i \in V, v_j \in N(v_i) \tag{4}$$

$$x_i, y_{ij} \in \{0, 1\} \tag{5}$$

This model can be seen as an improvement of the model originally presented in [10], because it has fewer binary variables. First, a binary variable x_i is associated to each vertex $v_i \in V$ indicating whether or not v_i is chosen to be included in the solution. Secondly, for each edge $(v_i, v_j) \in E$ the model contains binary variables y_{ij} and y_{ji}. Hereby, variable y_{ij} takes values one if vertex v_i dominates vertex v_j. Likewise, variable y_{ji} takes value one if v_j dominates v_i.

In the above formulation, constraints (2) enforce that all vertices that are not part of the solution are dominated by at least one neighbor. Constraints (3) make sure that the total number of vertices dominated by a given particular vertex v_i is limited by $Cap(v_i)$. In this way, a vertex v_i can dominate at most $Cap(v_i)$ vertices from its neighborhood. The constraints in (4) ensure that a vertex v_i can dominate a neighbor v_j if and only if v_i is part of the solution.

3. Proposed Algorithms

In this paper, we present the application of two hybrid algorithms for the CapMDS problem. Both proposals can be seen as algorithms from the category "*Hybrid Algorithms Based on Problem Instance Reduction* [30]". The main goal of this type of hybridization is to allow the application of an exact solver to large problem instances for which the direct use of the exact solver would not be possible or efficient given the instance size. For this purpose, this class of algorithms provides functionalities for the intelligent reduction in instances of a problem in order to obtain smaller sub-instances.

The first algorithm, labeled CMSA++, is a revised and extended version of the *construct, merge, solve and adapt* (CMSA) technique presented in prior work [11]. The second algorithm is a new proposal, called BARRAKUDA, which makes use of the algorithmic framework of a *biased random key genetic algorithm* (BRKGA) [31]. In Figure 2, we present a high-level overview of both techniques, especially for pointing out the two separated *levels of operation*. In the *original*

problem instance level of operation, CMSA++ and BARRAKUDA deal with the full-size problem instance and generate a set of solutions needed for the construction of sub-instances. In this context, CMSA++ performs the generation of independent solutions through a *constructive probabilistic heuristic*. Meanwhile, BARRAKUDA makes use of BRKGA as a generator of solutions. In contrast to CMSA++, BARRAKUDA makes use of learning for the generation of solutions. This is because it uses the natural learning mechanism of BRKGA, based on the selection of good solutions for acting as parents in order to produce offspring for the next iteration. This aspect will be explained in detail in Section 3.2.1. In the *reduced problem instance level of operation*, both algorithms make use of an ILP solver to deal with the incumbent sub-instance. If this sub-instance is small enough, the solver obtains good results providing over time high-quality solutions to the original problem instance.

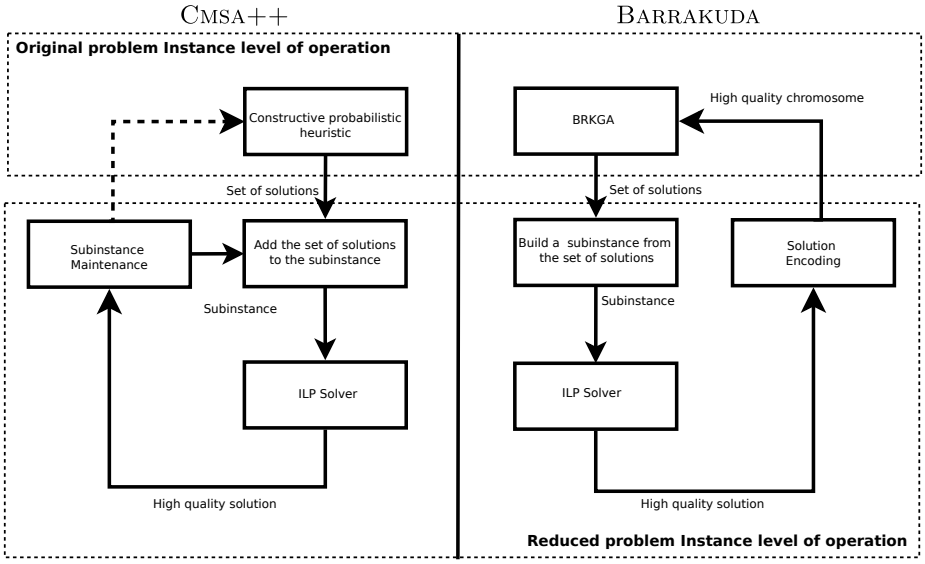


Figure 2. High-level overview of the two proposed hybrid algorithms.

The main difference between both proposals is the strategy for the creation and maintenance of the reduced sub-instances. CMSA++ produces sub-instances by using solution components found in the solutions produced by the constructive probabilistic heuristic, keeping the sub-instance size small enough through a *sub-instance maintenance* process. This process uses information about the utility of solution components, which is used to eliminate seemingly useless solution components from the sub-instances. On the other hand, BARRAKUDA does not make use of such a maintenance mechanism. In contrast, it creates sub-instances from scratch at each iteration. For this purpose, BARRAKUDA uses the solution components from a set of solutions selected from a subset of the chromosomes of the incumbent BRKGA population. After the application of the ILP solver to the sub-instance, BARRAKUDA implements a solution encoding process in order to be able to add the high-quality solution produced by the ILP solver to the incumbent population of the BRKGA.

Finally, the main reason for choosing BRKGA—instead of another evolutionary algorithm from the literature—as the main algorithm framework of BARRAKUDA is the following one. A BRKGA is usually very easy to apply, as it only requires to find a clever way of translating the individuals of BRKGA (which are kept as random keys) into feasible solutions to the tackled problem. All other components, such as mutation and crossover, are the same for any BRKGA. In this way, it is possible to focus all efforts on the hybridization aspect.

3.1. CMSA++*: An Extension of* CMSA

As mentioned above, CMSA++ is a revised and extended version of the CMSA algorithm from [11]. The extensions concern the following aspects:

- Incorporation of a set of new features concerning successful variations used in other CMSA implementations.
- Exploration of different constructive probabilistic heuristics for the probabilistic generation of solutions at each iteration.
- Testing of two different sub-instance maintenance mechanisms, going beyond the original CMSA.
- Making the algorithm more robust concerning the parameter values. Note that, in [11], we had to conduct a very fine-grained tuning process in order to achieve a good algorithm performance for all problem instances.

Nevertheless, keep in mind that the original CMSA from [11] can be obtained by setting the parameter values of CMSA++ in a specific way. In other words, the parameter tuning process may choose this option, if it results better than the newly introduced variations.

A high-level description of CMSA++ for the CapMDS problem is provided in Algorithm 1. The algorithm maintains, at all times, an initially empty subset V' of the set of vertices V of the input graph. This set is henceforth called the *sub-instance*. Vertices are added to sub-instance V' by means of a probabilistic solution construction process, which is implemented in function ProbabilisticSolutionGeneration (d_{rate^*},l_{size}, g_{opt}) (see line 10 of Algorithm 1). In particular, all dominator vertices found in the solutions generated by this function are added to V'—if not already in V'—and their so-called "age value" $age[]$ is initialized to zero (see lines 12 and 13 of Algorithm 1). Moreover, solving the sub-instance V' refers to the application of the ILP solver CPLEX in order to find—if possible within the imposed time limit of t_{solver} seconds—the optimal solution to sub-instance V'; that is, the optimal CapMDS solution in G that is limited to only contain dominators from V'. This is achieved by adding the following set of constraints to the ILP model from the previous section:

$$x_i = 0 \quad \forall x_i \in V \setminus V' \tag{6}$$

The process of solving the sub-instance V' is done in function ApplyExactSolver(V', t_{solver}); see line 16 of Algorithm 1.

Algorithm 1 CMSA++ for the CapMDS problem

1: **input:** a problem instance (G, Cap)
2: **input:** values for age_{max}, n_{a}, t_{solver}, l_{size} (original CMSA parameters)
3: **input:** values for d_{rate^L} , d_{rate^U}, A_{type}, g_{opt} (additional parameters)
4: $S_{\text{bsf}} := \emptyset$
5: $V' \leftarrow \emptyset$
6: $age[v] := 0$ for all $v \in V$
7: **while** CPU time limit not reached **do**
8: $d_{\text{rate}^*} :=$ DeterminismAdjustment$(d_{\text{rate}^L}, d_{\text{rate}^U})$
9: **for** $i = 1, \ldots, n_{\text{a}}$ **do**
10: $S \leftarrow$ ProbabilisticSolutionGeneration$(d_{\text{rate}^*}, l_{\text{size}}, g_{\text{opt}})$
11: **for** all $v \in S$ **and** $v \notin V'$ **do**
12: $age[v] := 0$
13: $V' \leftarrow V' \cup \{v\}$
14: **end for**
15: **end for**
16: $S'_{\text{opt}} \leftarrow$ ApplyExactSolver(V', t_{solver})
17: **if** $f(S'_{\text{opt}}) < f(S_{\text{bsf}})$ **then** $S_{\text{bsf}} \leftarrow S'_{\text{opt}}$
18: Adapt(V', S'_{opt}, age_{max}, A_{type})
19: **end while**
20: **output:** S_{bsf}

The algorithm takes as input the tackled problem instance (G, Cap), and values for eight required parameters. The first four of them—see line 2 of Algorithm 1—are inherited from the original CMSA proposal, while the remaining four parameters—see line 3 of Algorithm 1—are in the context of the extensions that lead to CMSA++. The eight parameters can be described as follows:

1. age_{max}: establishes the number of iterations that a vertex is allowed to form part of sub-instance V' without forming part of the solution to V' returned by the ILP solver (S'_{opt}).
2. n_{a}: defines the number of solutions generated in the construction phase of the algorithm at each iteration; that is, the number of calls to function ProbabilisticSolutionGeneration().
3. t_{solver}: establishes the time limit (in seconds) used for running the ILP solver at each iteration.
4. l_{size}, d_{rate^L} , d_{rate^U}: these three parameters determine the greediness of the solution construction process in function ProbabilisticSolutionGeneration(), which we described in more detail in the following subsection.
5. g_{opt}: indicates the variant of the solution construction process.
6. A_{type}: chooses between two different behaviors implemented for the sub-instance maintenance mechanism implemented in function Adapt().

As mentioned above, CMSA++ is—like any CMSA algorithm—equipped with a sub-instance maintenance mechanism for discarding seemingly useless solution components from the sub-instance V' at each iteration. The original implementation of this mechanism works as follows. First, the age values of all vertices in V' are incremented. Second, the age values of all vertices in S'_{opt} are re-initialized to zero. Third, the vertices with age values greater than age_{max} are erased from V'. In particular, this original implementation is used in case $A_{\text{type}} = 0$. In the other case—that is, when $A_{\text{type}} = 1$—a probability $\text{age}[v]/\text{age}_{\text{max}}$ for being removed from V' is assigned to each vertex $v \in V'$ with $\text{age}_{\text{max}} > 0$. Note that these probabilities linearly increase until reaching probability 1 for all vertices with an age value greater than age_{max}. Both mechanisms are implemented in function Adapt(V', S'_{opt}, age_{max}, A_{type}) (see line 18 of Algorithm 1). Their aim is to maintain V' small enough in order to be able to solve the sub-instance (if possible) optimally. If this is not possible in the allotted computation time, the output S'_{opt} of function ApplyExactSolver(V', t_{solver}) is simply the best solution found by CPLEX within the given time.

One of the new features of CMSA++ (in comparison to the original version from [11]) is a dynamic mechanism for adjusting the determinism rate (d_{rate^*}) used during the solution construction at each iteration. The mechanism is the same as that described in [32]. In the original CMSA, this determinism rate was a fixed value between 0 and 1, determined by algorithm tuning. In contrast, CMSA++ chooses a value for d_{rate^*} at each iteration in function DeterminismAdjustment(d_{rate^L}, d_{rate^U}) from the interval $[d_{\text{rate}^L}, d_{\text{rate}^U}]$ where $0 \leq d_{\text{rate}^L} \leq d_{\text{rate}^U} \leq 1$. This is done under the hypothesis that CMSA++ can escape from local optima, increasing the randomness of the algorithm. Whenever an iteration improves S_{bsf}, the value of d_{rate^*} is set back to its upper bound (d_{rate^U}). Otherwise, the value of d_{rate^*} is decreased by a factor determined by subtracting the lower bound vale from the upper bound value (d_{rate^U}) and dividing the result by 3.0. Finally, whenever the value of d_{rate^*} falls below the lower bound, it is set back to the upper bound value. Adequate values for d_{rate^L} and d_{rate^U} must be determined by algorithm tuning. Note that the behavior of the original CMSA algorithm is obtained by setting $d_{\text{rate}^U} = d_{\text{rate}^L}$.

Finally, note that CMSA++ iterates while a predefined CPU time limit is not reached. Moreover, it provides the best solution found during the search, S_{bsf}, as output.

Probabilistic Solution Generation

The function used for generating solutions in a probabilistic way (in line 10 of Algorithm 1) is presented in Algorithm 2. Note that there are three ways of generating solutions that are pseudo-coded in this algorithm. They are indicated in terms of three options: opt_1, opt_2, and opt_3. Line 18 of Algorithm 2, for example, is only executed for options opt_1 and opt_3. Note that option opt_1 implements the solution construction mechanism of the original CMSA algorithm for the CapMDS problem

from [11]. In the following section, first, the general solution construction procedure is described. Subsequently, we outline the differences between the three options mentioned above.

Algorithm 2 Function ProbabilisticSolutionGeneration(d_{rate^*}, l_{size}, g_{opt})

input: a problem instance (G, Cap)
input: parameter values for l_{size}, d_{rate^*}, g_{opt}
$S := \emptyset$
$W := V$
$V_{\text{uncov}} := V$
ComputeHeuristicValues(W)
while $V_{\text{uncov}} \neq \emptyset$ **do**
  Choose a random number $\delta \in [0, 1]$
  if $\delta \leq d_{\text{rate}^*}$ **then**
    Choose $v^* \in V$ such as $h(v^*) \geq h(v)$ for all $v \in W$
  else
    Let $L \subseteq W$ contain the $\min\{l_{\text{size}}, |W|\}$ vertices from W with the highest heuristic values
    Choose v^* uniformly at random from L
  end if
  $S := S \cup \{v^*\}$
  $V_{\text{uncov}} := V_{\text{uncov}} \setminus (C(v^*) \cup \{v^*\})$
  if $g_{\text{opt}} = \text{opt}_1$ **or** $g_{\text{opt}} = \text{opt}_3$ **then**
    $W := W \setminus (C(v^*) \cup \{v^*\})$
  else
    $W := W \setminus \{v^*\}$
  end if
  if $g_{\text{opt}} = \text{opt}_1$ **or** $g_{\text{opt}} = \text{opt}_2$ **then**
    ComputeHeuristicValues(W)
  end if
end while
output: S

The procedure takes as input the problem instance (G, Cap), as well as values for the parameters l_{size}, d_{rate^*} and g_{opt}. The two first parameters are used for controlling the degree of determinism of the solution construction process. Meanwhile, g_{opt} indicates one of the three options. The process starts with an empty partial solution $S := \emptyset$. At each step, exactly one vertex from a set $W \subseteq V$ is added to S, until the set of uncovered vertices (V_{uncov}) is empty. Note that both W and V_{uncov} are initially set to V. Next we explain how to choose a vertex v^* from W at each step of the procedure. First of all, vertices from W are evaluated in function ComputeHeuristicValues(W) by a dynamic greedy function $h()$, which depends on the current partial solution S. More specifically:

$$h(v) := 1 + \min\{Cap(v), \deg_S(v)\} \quad \forall v \in W \tag{7}$$

Hereby, $Cap(v)$ refers to the capacity of v and $\deg_S(v) := |N_{\text{uncov}}(v)|$, where $N_{\text{uncov}}(v) := N(v) \cap V_{\text{uncov}}$ is the set of currently uncovered neighbors of v. The $h()$-value of a vertex is henceforth called its heuristic value. The choice of a vertex v^* is then done as follows. First, a value $\delta \in [0, 1]$ is chosen uniformly at random. In case $\delta \leq d_{\text{rate}^*}$, v^* is chosen as the vertex that has the highest heuristic value among all vertices in W. Otherwise, a candidate list L containing the $\min\{l_{\text{size}}, |W|\}$ vertices from W with the highest heuristic values is generated, and v^* is chosen from L uniformly at random. Thus, the greediness of the solution construction procedure depends on the values of the determinism rate (d_{rate^*}) and the candidate list size (l_{size}). Note that, when $\deg_S(v^*)$ is greater than $Cap(v^*)$, we have to choose which vertices from the $N(v^*) \cap V_{\text{uncov}}$ vertex v^* is going to

cover. In our current implementation, the set of $Cap(v^*)$ vertices is simply chosen from $N(v^*) \cap V_{\text{uncov}}$ uniformly at random and stored in $C(v^*)$. Finally, W and V_{uncov} are updated at the end of each step.

When $g_{\text{opt}} = \text{opt}_1$, the solution construction procedure is the same as that used in the original CMSA algorithm for the CapMDS problem. In particular, W is updated after each choice of a vertex v^* in line 18 by removing the newly covered vertices. This concerns the chosen vertex v^* and the vertices from $C(v^*)$ that were chosen to be covered by v^*. Moreover, the greedy function values are always updated according to the current partial solution S (see line 23 of Algorithm 2). In contrast, when $g_{\text{opt}} = \text{opt}_2$ the update of W (see line 20) is done by only removing the chosen vertex v^* from the options. The rationale behind this is as follows. Consider the example in Figure 3. In case $g_{\text{opt}} = \text{opt}_1$, nodes v_1 and $C_{v_1} = \{v_2, v_3, v_4, v_5\}$ are removed from W for the next construction step. This removes node v_2 from the set of selectable vertices, although choosing v_2 in the next step would lead to the construction of the optimal solution in this example.

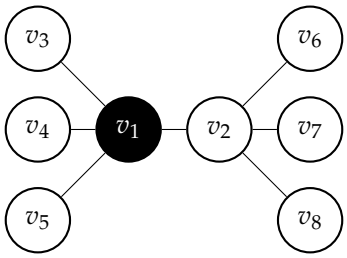


Figure 3. Example problem instance after the first step of generating a solution. Let us assume that the capacity of each node is 4. Vertex v_1 was chosen in the first step. As the capacity of v_1 is 4, all its neighbors are placed in $C(v_1)$ and covered by v_1.

Finally, with $g_{\text{opt}} = \text{opt}_3$ the solution construction is the same as with $g_{\text{opt}} = \text{opt}_1$, just that the greedy function values are not recalculated in line 23. The rationale behind this way of constructing solutions is to gain speed. On the negative side, this proposal possibly overestimates the heuristic values of the vertices and generally favors the hubs (vertices with a high degree) in the input graph.

3.2. BARRAKUDA*: A Hybrid of Brkga with an ILp Solver*

As mentioned before, the main algorithm proposed in this work is a hybrid between the well-known *biased random key genetic algorithm* (BRKGA) [31] and an ILP solver. Roughly, the solution components of (some of the solutions in the) population of the BRKGA are joined at the end of each iteration and the resulting sub-instance is passed to the ILP solver. Afterwards, the solution returned by the ILP solver is fed back into the population of BRKGA. As in CMSA, the main motivation for this proposal is to take profit from the ILP solver even in the context of problem instances that are too large in order to apply the ILP solver directly. The advantage of BARRAKUDA over CMSA is the learning component of BRKGA, which is not present in CMSA.

3.2.1. Main Algorithmic Framework

The main algorithm framework of BARRAKUDA, which is pseudo-coded in Algorithm 3, is that of the BRKGA. In fact, the lines that turn the BRKGA into BARRAKUDA are lines 12–15, which are marked by a different background colors. In the following section, we first outline all the BRKGA components of our algorithm before describing the additional BARRAKUDA components.

BRKGA is a steady-state genetic algorithm which consists of a problem-independent part and of a problem-dependent part. It takes as input the tackled problem instance (G, Cap), and the values of four parameters (p_{size}, p_e, p_m and $prob_{\text{elite}}$) that are found in any BRKGA. The algorithm starts with the initialization of a population P composed of p_{size} random individuals (line 4 of Algorithm 3). Each individual $\pi \in P$ is a vector of length established by $2 * |V|$, where V is the set of vertices of G. In order to generate a random individual, for each position $\pi(i)$ of π ($i = 1, \ldots, 2|V|$) a random

value from $[0,1]$ is generated. In this context, note that values $\pi(i)$ and $\pi(i+|V|)$ are associated to vertex v_i of the input graph G. Then, the elements of the population are evaluated (see line 5 of Algorithm 3). This operation, which translates each individual $\pi \in P$ into a solution S_π (set of vertices) to the CapMDS problem, corresponds to the problem-dependent part described in the next subsection. Note that, after evaluation, each individual $\pi \in P$ has assigned the objective function value $f(S_\pi)$ of corresponding CapMDS solution. Henceforth, $f(\pi)$ will refer to $f(S_\pi)$ and vice versa.

Algorithm 3 BARRAKUDA for the CapMDS problem

1: **input:** a problem instance (G, Cap)
2: **input:** values for p_{size}, p_e, p_m and $prob_{\text{elite}}$ (BRKGA parameters)
3: **input:** values for n_{a}, ex_{mode}, t_{solver} (additional BARRAKUDA parameters)
4: $P \leftarrow \mathsf{GenerateInitialPopulation}(p_{\text{size}})$
5: $\mathsf{Evaluate}(P)$
6: **while** computation time limit not reached **do**
7: $\quad P_e \leftarrow \mathsf{EliteSolutions}(P, p_e)$
8: $\quad P_m \leftarrow \mathsf{Mutants}(P, p_m)$
9: $\quad P_c \leftarrow \mathsf{Crossover}(P, P_e, prob_{\text{elite}})$
10: $\quad \mathsf{Evaluate}(P_m \cup P_c)$ {NOTE: P_e is already evaluated}
11: $\quad P := P_e \cup P_m \cup P_c$
12: $\quad P_{\text{ilp}} \leftarrow \mathsf{ChooseSolutions}(P, ex_{\text{mode}}, n_{\text{a}})$
13: $\quad V' := \bigcup_{S \in P_{\text{ilp}}} S$
14: $\quad S'_{\text{opt}} \leftarrow \mathsf{ApplyExactSolver}(V', t_{\text{solver}})$
15: $\quad P \leftarrow \mathsf{IntegrateSolverSolution}(P, S'_{\text{opt}})$
16: **end while**
17: **output:** Best solution in P

Then, at each iteration of the algorithm, a set of genetic operators are executed in order to produce the population of the next iteration (see lines 7–11 in Algorithm 3). First, the best $\max\{\lfloor p_e \cdot p_{\text{size}} \rfloor, 1\}$ individuals are copied from P to P_e in function $\mathsf{EliteSolutions}(P', p_e)$. Then, a set of $\max\{\lfloor p_m \cdot p_{\text{size}} \rfloor, 1\}$ mutant individuals are generated and stored in P_m. These mutants are produced in the same way as individuals from the initial population. Next, a set of $p_{\text{size}} - |P_e| - |P_m|$ individuals are produced by crossover in function $\mathsf{Crossover}(P', P_e, prob_{\text{elite}})$ and stored in P_c. Each such individual is generated as follows: First, an elite parent π_1 is chosen uniformly at random from P_e. Then, a second parent π_2 is chosen uniformly at random from $P \setminus P_e$. Finally, an offspring individual π_{off} is produced on the basis of π_1 and π_2 and stored in P_c. In the context of the crossover operator, value $\pi_{\text{off}}(i)$ is set to $\pi_1(i)$ with probability $prob_{\text{elite}}$, and to $\pi_2(i)$ otherwise. After generating all new offspring in P_m and P_c, these new individuals are evaluated in line 10. A standard BRKGA operation would now end after assembling the population of the next iteration (line 11).

3.2.2. Evaluation of an Individual

The problem-dependent part of the BRKGA concerns the evaluation of an individual, which consists of translating the individual into a valid CapMDS solution and calculating its objective function value. For this purpose, the solution generation procedure from Algorithm 2 is applied with option $g_{\text{opt}} = \text{opt}_2$. The only difference is in the choice of vertex $v^* \in W$ at each step (lines 8–14 of Algorithm 2), and the choice of the set of neighbors $C(v^*)$ to be covered by v^*. Both aspects are outlined in the following section. Moreover, the resulting pseudo-code is shown in Algorithm 4.

Instead of choosing a vertex $v^* \in W$ at each iteration of the solution construction process in a semi-probabilistic way as shown in lines 8–14 of Algorithm 2, the choice is done deterministically

Algorithm 4 Evaluation of an individual in BARRAKUDA

1: **input:** an individual π
2: $S_\pi := \emptyset$
3: $W := V$
4: $V_{\text{uncov}} := V$
5: ComputeHeuristicValues(W)
6: **while** $V_{\text{uncov}} \neq \emptyset$ **do**
7: $v^* := \text{argmax}\{h(v_i) \cdot \pi(i) \mid v_i \in W\}$
8: $S_\pi := S_\pi \cup \{v^*\}$
9: $C(v^*) = \emptyset$
10: $N := N_{\text{uncov}}(v^*)$
11: $r := \min\{|N_{\text{uncov}}(v^*)|, Cap(v^*)\}$
12: **while** $|C(v^*)| < r$ **and** $\exists v_i \in N$ s.t. $\pi(i + |V|) > 0$ **do**
13: $v' := \text{argmax}\{|N_{\text{uncov}}(v_i)| \cdot \pi(i + |V|) \mid v_i \in N\}$
14: $C(v^*) := C(v^*) \cup \{v'\}$
15: $N := N \setminus \{v'\}$
16: **end while**
17: $V_{\text{uncov}} := V_{\text{uncov}} \setminus (C(v^*) \cup \{v^*\})$
18: $W := W \setminus \{v^*\}$
19: ComputeHeuristicValues(W)
20: **end while**
21: **output:** S

based on combining the greedy function value $h(v_i)$ with its first corresponding value in individual $\pi(i)$. (Remember that function $h()$ is defined in Equation (7).) More specifically,

$$v^* := \text{argmax}\{h(v_i) \cdot \pi(i) \mid v_i \in W\} \tag{8}$$

This is also shown in line 7 of Algorithm 4. In this way, the BRKGA algorithm will produce—over time—individuals with high values $\pi(i)$ for vertices v_i that should form part of a solution, and vice versa. The greedy function $h()$ is only important for providing an initial bias towards an area in the search space that presumably contains good solutions. Finally, note that the first $|V|$ values of an individual (corresponding, in the given order, to vertices $v_1, \ldots, v_n$}) are used for deciding which vertices are chosen for the solution.

As mentioned above, the second aspect that differs when comparing the evaluation of an individual with the construction of a solution in Algorithm 2 is the choice of the set of neighbors $C(v^*)$ to be covered by v^* at each step. Remember that, in Algorithm 2, $\min\{|N_{\text{uncov}}|, Cap(v^*)\}$, vertices are randomly chosen from $N_{\text{uncov}}(v^*)$ and stored in $C(v^*)$. In contrast, for the evaluation of a solution, vertices are sequentially picked from $N_{\text{uncov}}(v^*)$ and added to $C(v^*)$ until either $\min\{|N_{\text{uncov}}|, Cap(v^*)\}$ vertices are picked or no further uncovered neighbor v_i of v^* exists with $\pi(i + |V|) > 0$. Assuming that we denote the set of remaining (still unpicked) uncovered vertices of v^* by N, at each step the following vertex is picked from N:

$$v' := \text{argmax}\{|N_{\text{uncov}}(v_i)| \cdot \pi(i + |V|) \mid v_i \in N\} \tag{9}$$

This is also shown in line 13 of Algorithm 4. Note that, as a greedy function for choosing from the uncovered neighbors of v^*, we make use of the number of uncovered neighbors of these vertices, preferring those with a large number of uncovered neighbors. Note also that, for selecting the vertices that are dominated by a chosen node v^*, the second half of an individual is used, that

is, values $\pi(|V|+1), \ldots, \pi(2|V|)$. In other words, with the second half of an individual, the BRKGA learns by which node a non-chosen node should be covered.

3.2.3. From Brkga to BARRAKUDA

Finally, it remains to describe lines 12–15 of Algorithm 3 that are an addition to the standard BRKGA algorithm and that convert the algorithm into BARRAKUDA. This part makes use of three BARRAKUDA-specific parameters:

- Parameter t_{solver} is used for establishing the time limit given to the ILP solver in function $\mathsf{ApplyExactSolver}(V', t_{\text{solver}})$.
- Parameter ex_{mode} is used for determining the way in which individuals/solutions are chosen from the current BRKGA population for building a sub-instance V'. This is done in function $\mathsf{ChooseSolutions}(P, ex_{\text{mode}}, n_{\text{a}})$.
- Parameter n_{a} refers to the number of individuals that must be chosen from P for the construction of the sub-instance.

The BARRAKUDA-specific part of the algorithm starts by choosing exactly $n_{\text{a}} < p_{\text{size}}$ individuals from the current population P and storing their corresponding solutions (which are known, because these individuals are already evaluated) in a set P_{ilp}. Parameter ex_{mode} indicates the way in which these n_{a} solutions are chosen. In case $ex_{\text{mode}} =$ ELITIST, the n_{a} best individuals from P are chosen. Otherwise (when $ex_{\text{mode}} =$ RANDOM), the best individual from P in addition to $(n_{\text{a}} - 1)$ randomly chosen solutions are added to P_{ilp}. Subsequently, the vertices of all solutions from P_{ilp} are merged into set V', and CPLEX is applied in function $\mathsf{ApplyExactSolver}(V', t_{\text{solver}})$ in the same way as in the case of CMSA++. Finally, the solution S'_{opt} returned by the solver is fed back to BRKGA in function $\mathsf{IntegrateSolverSolution}(P, S'_{\text{opt}})$. More specifically, S'_{opt} is transformed into an individual π, which is then added to P. Finally, the worst individual is deleted from P in order to keep the population size constant. The integration process is quite straightforward, because it only considers information about the vertices (dominators) used in the solvers' solution and not the domination relationships (edges). More specifically, BARRAKUDA creates a new individual π where $\pi(i) := 0.5, \forall i \in \{|V|+1, |V|+2, \ldots, 2|V|\}$. Moreover, for each dominating vertex v_i in the solution of the solver $\pi(i) := 1.0$, whereas for each remaining vertex v_j, the value of $\pi(j)$, is set to zero.

4. Experimental Evaluation

In this section, we present the comparison of our two algorithmic proposals: CMSA++ and BARRAKUDA. For this purpose, we employ four different sets of problem instances. Two of them were presented in [9] and previously used in [11]. The remaining two instance sets are newly generated, because we noticed that most of the existing problem instances from the first two data sets are too small—respectively, too easy to solve—in order to be challenging. All four benchmark sets will be described in the following subsection.

We implemented all algorithms from scratch using ANSI C++ and compiled the code with GCC 8.3.0 (Ubuntu/Linux). Moreover, IBM ILOG CPLEX Optimization Studio V12.8.0 was executed in one-threaded mode, both as a standalone application (henceforth simply called CPLEX) and within CMSA++ and BARRAKUDA for solving the generated sub-instances. All experiments were conducted on the computing cluster of the Engineering Faculty of the University of Concepción (Luthier). Luthier is composed of 30 computing nodes, which all have the same hardware and software configuration. In particular, all nodes have an Intel CPU Xeon E3-1270 v6 at 3.8 GHz with 64 GB RAM. The cluster uses SLURM v17.11.2 for management and job scheduling purposes. The time limit for all experiments was 1000 CPU seconds per algorithm and per run, for all problem instances.

4.1. Benchmark Instances

The first two benchmark sets were taken from the literature. They were initially presented in [9] and later used for the experimental evaluation of the current state-of-the-art method in [10]. The first benchmark set contains 120 unit disk graphs (UDGs) created using the topology generator from [33]. In these instances, the vertices of each graph are randomly distributed over a Euclidean square of size 1000×1000. Moreover, the graphs are generated with two different range values: 150 and 200 units. Note that with a range value of 150, for example, an undirected edge is introduced between each pair of vertices whose Euclidean distance is mostly 150 units. Accordingly, graphs on the same number of vertices become more dense with a growing range value used for generating them. The second benchmark set consists of 180 general graphs taken from the set of so-called type I instances originally proposed by Shyu et al. [34] in the context of the minimum weight vertex cover problem. For each graph size (in terms of the number of vertices) this benchmark set contains graphs from three different densities as indicted by their number of undirected edges. In both benchmark sets, the number of vertices of the graphs is from $\{50, 100, 250, 500, 800, 1000\}$. Finally, graphs with vertices of two types of capacities—namely uniform capacities and variable capacities—can be found in these benchmark sets. In the case of uniform capacity, three different capacities of 2, 5 and α are tested, where α is graph-specific and is taken as the average degree of the corresponding graph. In the case of variable capacities, the vertex capacities are randomly chosen from the following three intervals: $(2, 5)$, $(\alpha/5, \alpha/2)$ and $[1, \alpha]$. Note that the instance files come with the capacities already explicitly assigned. Note that for each combination of graph size (number of vertices), density (as determined by range values, respectively, number of edges) and capacity type/range, the benchmark sets consist of 10 randomly generated problem instances.

As we had already noticed in our preliminary work [11] that many of the problem instances from the above-mentioned benchmark sets are easily solved by CPLEX, we decided generate more challenging instances, as follow. The first set is composed of general random graphs with 1000 or 5000 nodes. In this set, the number of edges depends on a parameter called edge probability (ep). This parameter is used in the construction of the graphs, where each possible edge is generated with probability ep. The probabilities that we considered are from $\{0.05, 0.15, 0.25\}$. Finally, we chose to generate instances with a uniform capacity of α, and others with a variable capacity for each vertex randomly chosen from $[1, \alpha]$. The second one of the new benchmark sets is composed of random geometric graphs that are generated in a similar way as the unit disc graph instances mentioned above. All vertices are randomly distributed over the euclidean square—that is, a square of size 1.0×1.0—and a connection radius r determines the edges. In particular, all vertices whose Euclidean distance is at most r are connected by an edge. Note that the radius r determines the density of the resulting graph. We used radius values from $\{0.14, 0.24, 0.34\}$. For this set we also considered instances with 1000 and 5000 vertices, as well as uniform capacity graphs with capacity α assigned to each node and variable capacity problems with a random capacity from $[1, \alpha]$ assigned to each vertex. In both cases (general random graphs and random geometric graphs) we generated 10 instances for each combination of graph size, density, and capacity type/size.

4.2. Tuning Process

As mentioned before, the reason for introducing CMSA++, which is an extension of the CMSA algorithm presented in preliminary work [11], was that CMSA was not robust at all. Its parameters had to be tuned separately for many sub-groups of the set of available benchmark instances. Only in this way, the algorithm obtained very good results. In order to confirm that CMSA++ and BARRAKUDA are sufficiently robust algorithms, a significantly lower number of algorithm configurations than those produced in [11] for CMSA are produced for the final evaluation of CMSA++ and BARRAKUDA. In particular, we produce different algorithm configurations only for 12 subsets of the instances from the literature. This is in contrast to 36 configurations that were produced for CMSA in [11]. Remember in this context that CMSA++ is—from an algorithmic point of view—a superset of CMSA.

This means that, if the proposed extensions are not found to be useful, the tuning process would choose the parameter settings, such that CMSA++ is the same as CMSA.

As outlined in the previous section, CMSA++ requires well-working parameter values for parameters {age_{max}, n_{a}, t_{solver}, l_{size}, d_{rate^L} , d_{rate^U}, A_{type}, g_{opt} }, while BARRAKUDA required values for parameters {p_{size}, p_e, p_m, $prob_{\text{elite}}$,n_{a}, ex_{mode}, t_{solver}}. Please refer to Section 3 for a comprehensive description of their function. We employ the scientific parameter tuning tool *irace* [35] for determining the values of these parameters. The parameters' value domains for all the tuning runs were chosen as follows.

CMSA++ parameter domains:

- $\text{age}_{\text{max}} : \{1, 2, 3, 5, 10, 1000\}$
- $n_{\text{a}} : \{1, 2, 5, 10, 30, 50\}$
- $t_{\text{solver}} : \{3, 5, 10, 50, 75, 100, 250, 500\}$
- $l_{\text{size}} : \{3, 5, 10, 20, 50, 100\}$
- $d_{\text{rate}^L} : [0, 0.9]$
- $d_{\text{rate}^U} : [0, 0.9]$
- $A_{\text{type}} : \{0, 1\}$
- $g_{\text{opt}} : \{1, 210, 2\}$

BARRAKUDA parameter domains:

- $p_{\text{size}} : \{10, 20, 50, 100, 200, 500\}$
- $p_e : \{0.1, 0.15, 0.2, 0.25\}$
- $p_m : \{0.1, 0.15, 0.2, 0.25, 0.3\}$
- $prob_{\text{elite}} : [0.5, 0.9]$
- $n_{\text{a}} : \{1, 3, 5, 10, 30, 50\}$
- $ex_{\text{mode}} : \{0, 1\}$
- $t_{\text{solver}} : \{3, 5, 10, 50, 75, 100, 200, 500\}$

Hereby, real-valued parameters, such as d_{rate^L}, for example, are all treated with a precision of two positions behind the comma—that is, parameter d_{rate^L} whose domain is $[0, 0.9]$ can take values from $\{0.0, 0.01, \ldots, 0.89, 0.9\}$.

Tables 1 and 2 show the parameter value configurations of CMSA++ and BARRAKUDA for the problem instances from the literature. Note that we distinguish between small problem instances (marked by S in the first table columns) and large problem instances (marked by L). Hereby, configuration S is for all instances with $\{50, 100, 250\}$ vertices, and configuration L for the remaining (large) problem instances. Moreover, tuning is performed for three different capacity types; see the second table columns. Parameter values from the rows with $\{2, (2, 5)\}$ in the second column are, for example, for all instances with a uniform capacity of 2, and for all instances with a variable capacity from $(2, 5)$.

Table 1. CMSA++ parameter values generated by irace for the instances from the literature.

Size	Capacity	d_{rate^L}	d_{rate^u}	l_{size}	age_{max}	n_{a}	t_{solver}	A_{type}	g_{opt}
S	$\{2,(2,5)\}$	0.6	0.9	5	5	1	3	1	10
	$\{5,(\alpha/5,\alpha/2)\}$	0.1	0.7	10	2	1	100	0	10
	$\{\alpha,[1,\alpha]\}$	0.6	0.7	10	1	2	50	0	10
L	$\{2,(2,5)\}$	0.2	0.2	50	5	2	100	1	10
	$\{5,(\alpha/5,\alpha/2)\}$	0.6	0.9	50	3	1	250	0	10
	$\{\alpha,[1,\alpha]\}$	0.2	0.3	10	3	5	100	1	10

(**a**) Settings for unit disk graphs.

Size	Capacity	d_{rate^L}	d_{rate^u}	l_{size}	age_{max}	n_{a}	t_{solver}	A_{type}	g_{opt}
S	$\{2,(2,5)\}$	0.3	0.4	50	10	5	10	1	210
	$\{5,(\alpha/5,\alpha/2)\}$	0.5	0.8	20	3	30	50	1	2
	$\{\alpha,[1,\alpha]\}$	0.1	0.2	50	10	30	100	1	2
L	$\{2,(2,5)\}$	0.4	0.7	3	1	5	100	1	10
	$\{5,(\alpha/5,\alpha/2)\}$	0.8	0.9	3	3	5	250	0	10
	$\{\alpha,[1,\alpha]\}$	0.5	0.9	20	1	5	250	0	10

(**b**) Settings for general graphs.

Table 2. BARRAKUDA parameter values generated by irace for the instances from the literature.

Size	Capacity	p_{size}	p_e	p_m	$prob_{\text{elite}}$	n_{a}	ex_{mode}	t_{solver}
S	$\{2,(2,5)\}$	100	0.25	0.15	0.8	1	0	10
	$\{5,(\alpha/5,\alpha/2)\}$	100	0.25	0.3	0.8	30	0	10
	$\{\alpha,[1,\alpha]\}$	200	0.2	0.15	0.9	10	0	75
L	$\{2,(2,5)\}$	800	0.25	0.2	0.8	2	1	100
	$\{5,(\alpha/5,\alpha/2)\}$	20	0.2	0.15	0.5	2	1	50
	$\{\alpha,[1,\alpha]\}$	500	0.1	0.15	0.6	10	0	75

(**a**) Settings for unit disk graphs.

Size	Capacity	p_{size}	p_e	p_m	$prob_{\text{elite}}$	n_{a}	ex_{mode}	t_{solver}
S	$\{2,(2,5)\}$	500	0.15	0.25	0.6	50	1	200
	$\{5,(\alpha/5,\alpha/2)\}$	800	0.2	0.15	0.7	30	0	10
	$\{\alpha,[1,\alpha]\}$	200	0.15	0.15	0.9	30	0	100
L	$\{2,(2,5)\}$	200	0.2	0.15	0.9	50	0	75
	$\{5,(\alpha/5,\alpha/2)\}$	500	0.2	0.15	0.6	30	1	200
	$\{\alpha,[1,\alpha]\}$	800	0.15	0.15	0.8	30	0	10

(**b**) Settings for general graphs.

Tables 3 and 4 show the parameter values determined by irace for the instances from the new data set of large random graphs (RGs) and large random geometric graphs (RGGs) for CMSA++ and BARRAKUDA, respectively. In these cases, the tuning for the uniform capacity problems was done separately from the variable capacity problems.

Finally, note that the original BRKGA algorithm can be obtained from BARRAKUDA by a setting of $n_a = 1$, which means that the sub-instance V' is built from only one solution and that the ILP solver can therefore not find any better solution in the sub-instance. In fact, the setting of $n_a = 1$ was determined by irace very few times, such as, for example, for the instances of the first row in Table 2a. In other words, in these cases the original BRKGA algorithm is better than BARRAKUDA. In the overwhelming majority of the cases, however, BARRAKUDA is significantly better than BRKGA.

Table 3. CMSA++ parameter values generated by irace for the new data set of large problem instances.

Type	n	ep/r	Capacity	d_{rate^L}	d_{rate^u}	l_{size}	age_{max}	n_{a}	t_{solver}	A_{type}	g_{opt}
		$ep = 0.05$	α	0.19	0.73	20	10	1	75	1	10
RG	1000	$ep = 0.15$	α	0.43	0.73	3	3	1	75	1	10
		$ep = 0.25$	α	0.1	0.52	20	1	1	75	1	10
		$ep = 0.05$	α	0.73	0.78	5	3	2	500	0	10
RG	5000	$ep = 0.15$	α	0.32	0.66	5	5	1	250	1	10
		$ep = 0.25$	α	0.32	0.68	3	3	2	500	0	10
		$ep = 0.05$	$[1, \alpha]$	0.47	0.61	3	1000	1	75	1	10
RG	1000	$ep = 0.15$	$[1, \alpha]$	0.02	0.56	3	2	1	100	1	10
		$ep = 0.25$	$[1, \alpha]$	0.1	0.76	10	3	1	100	1	10
		$ep = 0.05$	$[1, \alpha]$	0.09	0.21	3	1000	1	500	1	10
RG	5000	$ep = 0.15$	$[1, \alpha]$	0.21	0.54	5	1	1	500	0	10
		$ep = 0.25$	$[1, \alpha]$	0.26	0.8	3	10	2	250	1	10
		$r = 0.14$	α	0.11	0.55	100	3	5	5	1	10
RGG	1000	$r = 0.24$	α	0.48	0.81	3	3	5	75	0	10
		$r = 0.34$	α	0.47	0.9	10	3	2	50	1	2
		$r = 0.14$	α	0.7	0.78	10	2	5	500	1	10
RGG	5000	$r = 0.24$	α	0.46	0.58	10	3	5	250	1	10
		$r = 0.34$	α	0.15	0.89	100	2	10	500	0	10
		$r = 0.14$	$[1, \alpha]$	0.05	0.6	100	10	2	500	1	10
RGG	1000	$r = 0.24$	$[1, \alpha]$	0.12	0.62	100	10	1	250	1	210
		$r = 0.34$	$[1, \alpha]$	0.36	0.73	5	10	50	10	0	2
		$r = 0.14$	$[1, \alpha]$	0.11	0.16	20	5	2	500	1	10
RGG	5000	$r = 0.24$	$[1, \alpha]$	0.55	0.62	50	3	5	500	1	10
		$r = 0.34$	$[1, \alpha]$	0.65	0.89	20	10	5	500	1	2

Table 4. BARRAKUDA parameter values generated by irace for the new data set of large problem instances.

Type	n	ep/r	Capacity	p_{size}	p_e	p_m	$rhoe$	n_{a}	f_{mode}	t_{solver}
		$ep = 0.05$	α	500	0.15	0.15	0.8	10	0	3
RG	1000	$ep = 0.15$	α	500	0.1	0.15	0.8	2	1	100
		$ep = 0.25$	α	50	0.1	0.25	0.69	2	1	5
		$ep = 0.05$	α	500	0.2	0.3	0.9	50	0	5
RG	5000	$ep = 0.15$	α	20	0.1	0.2	0.71	1	0	75
		$ep = 0.25$	α	20	0.1	0.15	0.82	2	1	75
		$ep = 0.05$	$[1, \alpha]$	500	0.15	0.15	0.81	5	0	5
RG	1000	$ep = 0.15$	$[1, \alpha]$	500	0.1	0.1	0.53	2	0	100
		$ep = 0.25$	$[1, \alpha]$	100	0.2	0.2	0.64	1	0	500
		$ep = 0.05$	$[1, \alpha]$	500	0.2	0.1	0.6	50	1	10
RG	5000	$ep = 0.15$	$[1, \alpha]$	50	0.15	0.2	0.9	1	0	75
		$ep = 0.25$	$[1, \alpha]$	100	0.15	0.3	0.68	1	0	100
		$r = 0.14$	α	50	0.25	0.3	0.56	30	1	100
RGG	1000	$r = 0.24$	α	500	0.2	0.1	0.68	2	1	75
		$r = 0.34$	α	50	0.25	0.1	0.71	5	0	10
		$r = 0.14$	α	50	0.15	0.1	0.84	30	1	75
RGG	5000	$r = 0.24$	α	50	0.15	0.15	0.68	50	1	100
		$r = 0.34$	α	50	0.15	0.2	0.68	30	0	500
		$r = 0.14$	$[1, \alpha]$	50	0.25	0.2	0.69	50	0	75
RGG	1000	$r = 0.24$	$[1, \alpha]$	200	0.25	0.25	0.67	10	0	500
		$r = 0.34$	$[1, \alpha]$	50	0.1	0.1	0.56	5	0	75
		$r = 0.14$	$[1, \alpha]$	500	0.15	0.2	0.58	5	0	500
RGG	5000	$r = 0.24$	$[1, \alpha]$	100	0.1	0.1	0.71	10	0	500
		$r = 0.34$	$[1, \alpha]$	50	0.25	0.15	0.54	10	0	500

4.3. Results

In addition to CPLEX, CMSA++ and BARRAKUDA were applied with the corresponding parameter values from above to all problem instances exactly once. The time limit for each run was set to

1000 CPU seconds for each algorithm. The results for the problem instances from the literature are presented in numerical form in Tables 5–8. Each table row contains the solution quality (columns with heading $\bar{q}$) and the time at which this result was obtained (columns with heading $\bar{t}$) averaged over 10 problem instances for each of the three algorithms. Moreover, we show the results of LS_PD, which is an algorithm based on a local search that was the state-of-the-art approach for the CapMDS problem before the publication of CMSA. In addition to solution quality and computation time, in the case of CMSA++ and BARRAKUDA we also provide the average gap (in percent) over the respective 10 problem instances, either with respect to the optimal solutions obtained by CPLEX, or—if not possible—with respect to the optimal MDS solutions of the respective problem instances. In the former case, the gaps are marked by a superscript ''a", and in the latter one by a superscript ''b". Those cases in which not even the optimal MDS solutions could be computed (due to excessive computation times) are marked with ''– –". The best results per table row are indicated in bold font. Moreover, the result of CPLEX is shown with a grey background in case the result is proven optimal. Finally, results with a preceding asterisk indicate new best-known results. In order to summarize these results, we additionally generated so-called *critical difference* (CD) plots for subsets of these problem instances (see the graphics in Figure 4). For their generation, first, the Friedman test was used to compare CPLEX, CMSA++ and BARRAKUDA simultaneously. (LS_PD was not included in the critical difference plots because, unfortunately, the detailed results per instance were not available. In any case, LS_PD is clearly worse than the other three techniques.) The hypothesis that the techniques perform equally was rejected. Subsequently, the algorithms were compared pairwise by the Nemenyi post-hoc test [36]. The obtained results can graphically be shown in the form of the above-mentioned CD plots in which each algorithm is placed on the horizontal axis according to its average ranking for the considered subset of problem instances. The performances of those algorithm variants that are below the critical difference threshold (computed with a significance level of 0.05) are considered as statistically equivalent. Such cases are shown by additional horizontal bars joining the average ranking markers of the respective algorithm variants. (The statistical evaluation was conducted using R's **scmamp** package [37], available at https://github.com/b0rxa/scmamp.)

The following observations can be made:

- First of all, CPLEX as well as CMSA++ and BARRAKUDA clearly outperform LS_PD on most problem instances.
- However, most of the instances from the literature are no challenge, neither for the exact solver (CPLEX), nor for the heuristic solvers proposed in this work (CMSA++ and BARRAKUDA). In fact, CPLEX is able to solve most problem instances to optimality; see the cases with a grey background. Interestingly, CPLEX seems to have more problems with unit disk graphs than with random graphs. Moreover, instances with variable node capacities seem slightly harder as well. This is also indicated by the CD plots in Figure 4b–g. In particular, the CD plots in Figure 4b,d,f show that CPLEX is not significantly worse than the best-performing algorithm. In the case of medium capacities (Figure 4d) CPLEX even ranks best on average. In contrast, Figure 4c,e,g show that BARRAKUDA is (apart from the case of medium capacities) significantly better than CPLEX and CMSA++.
- Even though we can only detect rather small differences between the three best-performing algorithms, when considering all problem instances from the literature together, BARRAKUDA performs significantly better than CPLEX, which—in turn—performs significantly better than CMSA.

Table 5. Results for general graphs with uniform capacity.

Capacity	n	Range	CPLEX		LS_PD		CMSA++			BARRAKUDA		
			$\bar{q}$	$\bar{t}$	$\bar{q}$	$\bar{t}$	$\bar{q}$	$\bar{t}$	gap (%)	$\bar{q}$	$\bar{t}$	gap (%)
2	50	100	**17.0**	<0.1	**17.0**	<0.1	**17.0**	<0.1	0.0 [a]	**17.0**	0.1	0.0 [a]
		250	**17.0**	<0.1	**17.0**	<0.1	**17.0**	0.1	0.0 [a]	**17.0**	0.1	0.0 [a]
		500	**17.0**	0.1	**17.0**	<0.1	**17.0**	0.1	0.0 [a]	**17.0**	0.2	0.0 [a]
2	100	100	**34.0**	0.2	**34.0**	<0.1	**34.0**	0.1	0.0 [a]	**34.0**	0.1	0.0 [a]
		250	**34.0**	0.1	**34.0**	<0.1	**34.0**	0.1	0.0 [a]	**34.0**	0.2	0.0 [a]
		500	**34.0**	0.1	**34.0**	<0.1	**34.0**	0.1	0.0 [a]	**34.0**	0.2	0.0 [a]
2	250	250	**84.0**	1.7	**84.0**	3.6	**84.0**	0.2	0.0 [a]	**84.0**	0.4	0.0 [a]
		500	**84.0**	0.4	**84.0**	5.7	**84.0**	0.3	0.0 [a]	**84.0**	0.6	0.0 [a]
		1000	**84.0**	0.6	**84.0**	10.0	**84.0**	0.3	0.0 [a]	**84.0**	0.8	0.0 [a]
2	500	500	**167.0**	10.8	168.6	<0.1	**167.0**	6.9	0.0 [a]	**167.0**	0.7	0.0 [a]
		1000	**167.0**	1.7	170.2	55.5	**167.0**	0.8	0.0 [a]	**167.0**	1.2	0.0 [a]
		2000	**167.0**	1.7	167.5	38.4	**167.0**	0.8	0.0 [a]	**167.0**	1.3	0.0 [a]
2	800	1000	**267.0**	2.7	274.2	160.9	**267.0**	1.3	0.0 [a]	**267.0**	2.2	0.0 [a]
		2000	**267.0**	3.4	272.7	127.1	**267.0**	2.0	0.0 [a]	**267.0**	3.2	0.0 [a]
		5000	**267.0**	4.6	267.8	44.4	**267.0**	1.8	0.0 [a]	**267.0**	3.5	0.0 [a]
2	1000	1000	**334.0**	145.8	338.4	136.5	**334.0**	58.4	0.0 [a]	**334.0**	2.7	0.0 [a]
		5000	**334.0**	5.7	336.0	126.0	**334.0**	2.9	0.0 [a]	**334.0**	4.9	0.0 [a]
		10,000	**334.0**	12.6	**334.0**	4.2	**334.0**	3.0	0.0 [a]	**334.0**	7.1	0.0 [a]
Avg.			**150.5**	12.0	151.9	64.8	**150.5**	4.4		**150.5**	1.6	
5	50	100	**11.9**	0.1	**11.9**	<0.1	**11.9**	<0.1	0.0 [a]	**11.9**	0.1	0.0 [a]
		250	**9.0**	0.2	**9.0**	<0.1	**9.0**	0.1	0.0 [a]	**9.0**	<0.1	0.0 [a]
		500	**9.0**	0.2	**9.0**	<0.1	**9.0**	0.1	0.0 [a]	**9.0**	<0.1	0.0 [a]
5	100	100	**33.6**	<0.1	**33.6**	0.2	**33.6**	<0.1	0.0 [a]	**33.6**	0.1	0.0 [a]
		250	**20.0**	0.9	20.3	5.9	20.3	0.3	1.5 [a]	**20.0**	0.5	0.0 [a]
		500	**17.0**	0.6	**17.0**	0.1	**17.0**	0.2	0.0 [a]	**17.0**	0.4	0.0 [a]
5	250	250	**83.3**	0.2	83.5	11.6	**83.3**	8.0	0.0 [a]	**83.3**	0.5	0.0 [a]
		500	**57.8**	9.7	59.9	46.0	58.5	0.1	1.7 [a]	57.9	12.0	0.2 [a]
		1000	**42.0**	15.7	45.0	32.6	42.0	1.6	0.0 [a]	**42.0**	12.3	0.0 [a]
5	500	500	**167.0**	0.9	168.3	10.7	**167.0**	371.7	0.0 [a]	**167.0**	1.5	0.0 [a]
		1000	**114.9**	821.4	121.5	80.5	115.6	258.9	0.8 [b]	115.4	540.9	0.6 [b]
		2000	84.1	254.1	92.2	71.3	**84.0**	0.8	– –	**84.0**	136.7	– –
5	800	1000	**242.5**	5.8	253.2	101.8	250.4	8.6	3.3 [a]	**242.5**	10.2	0.0 [a]
		2000	**164.8**	1000.0	176.2	223.6	167.5	616.7	– –	166.2	161.6	– –
		5000	**134.0**	69.3	140.4	131.6	**134.0**	15.4	0.0 [a]	**134.0**	25.1	0.0 [a]
5	1000	1000	**333.7**	3.1	338.7	75.7	**333.7**	3.8	0.0 [a]	**333.7**	4.5	0.0 [a]
		5000	**167.0**	205.6	181.0	140.3	**167.0**	18.0	0.0 [a]	**167.0**	72.0	0.0 [a]
		10,000	**167.0**	149.7	171.0	62.7	**167.0**	50.7	0.0 [a]	**167.0**	54.6	0.0 [a]
Avg.			**103.3**	149.3	107.3	66.3	103.9	75.3		103.4	57.4	
α	50	100	**12.0**	0.2	**12.0**	<0.1	**12.0**	0.2	0.0 [a]	**12.0**	0.1	0.0 [a]
		250	**6.0**	0.5	**6.0**	0.1	**6.0**	0.1	0.0 [a]	**6.0**	0.1	0.0 [a]
		500	**3.8**	0.9	**3.8**	<0.1	**3.8**	0.2	0.0 [a]	**3.8**	<0.1	0.0 [a]
α	100	100	**34.0**	0.2	**34.0**	<0.1	**34.0**	<0.1	0.0 [a]	**34.0**	<0.1	0.0 [a]
		250	**20.0**	1.0	20.2	5.9	20.2	0.3	1.0 [a]	20.1	0.3	0.5 [a]
		500	**12.2**	5.5	**12.2**	2.6	**12.2**	2.8	0.0 [a]	**12.2**	3.0	0.0 [a]
α	250	250	**84.0**	2.1	**84.0**	2.1	85.2	0.1	1.4 [a]	**84.0**	0.2	0.0 [a]
		500	**58.3**	30.6	61.7	17.8	58.4	21.5	0.2 [a]	**58.3**	28.5	0.0 [a]
		1000	36.7	999.8	37.9	41.1	36.6	326.6	1.4 [b]	*__36.5__	215.3	1.1 [b]
α	500	500	**167.0**	15.8	168.4	32.3	**167.0**	0.9	0.0 [a]	**167.0**	1.7	0.0 [a]
		1000	**116.2**	966.2	126.4	59.0	117.0	628.5	2.0 [b]	116.3	95.6	1.4 [b]
		2000	75.3	1000.0	78.6	101.0	75.7	553.5	– –	**74.3**	244.3	– –
α	800	1000	**267.0**	3.6	274.0	120.8	**267.0**	1.4	0.0 [a]	**267.0**	4.8	0.0 [a]
		2000	164.9	1001.0	178.1	159.2	167.1	589.1	– –	*__162.6__	265.4	– –
		5000	91.1	993.6	92.2	329.4	91.2	575.2	– –	*__89.6__	569.7	– –
α	1000	1000	**334.0**	131.7	338.4	34.7	**334.0**	5.9	0.0 [a]	**334.0**	6.8	0.0 [a]
		5000	**132.5**	951.3	137.4	352.2	133.6	758.5	– –	134.2	711.1	– –
		10,000	87.6	993.9	81.3	604.8	83.2	610.6	– –	*__80.7__	663.6	– –
Avg.			94.6	394.3	97.0	103.5	94.7	226.4		**94.0**	156.1	
Total avg.			116.1	185.2	118.8	78.2	116.4	102.0		**116.0**	71.7	

Table 6. Results for general graphs with variable capacity.

Capacity	n	Range	CPLEX		LS_PD		CMSA++			BARRAKUDA		
			$\bar{q}$	$\bar{t}$	$\bar{q}$	$\bar{t}$	$\bar{q}$	$\bar{t}$	gap (%)	$\bar{q}$	$\bar{t}$	gap (%)
$(2,5)$	50	100	**13.0**	0.1	**13.0**	0.2	**13.0**	0.1	0.0 [a]	**13.0**	0.1	0.0 [a]
		250	**9.0**	0.1	**9.0**	<0.1	**9.0**	<0.1	0.0 [a]	**9.0**	0.1	0.0 [a]
		500	**9.0**	0.1	**9.0**	<0.1	**9.0**	<0.1	0.0 [a]	**9.0**	<0.1	0.0 [a]
$(2,5)$	100	100	**33.7**	0.1	**33.7**	0.1	**33.7**	<0.1	0.0 [a]	**33.7**	0.1	0.0 [a]
		250	**21.9**	0.7	22.3	8.9	**21.9**	0.3	0.0 [a]	**21.9**	0.3	0.0 [a]
		500	**17.0**	0.4	17.2	6.5	**17.0**	0.2	0.0 [a]	**17.0**	0.3	0.0 [a]
$(2,5)$	250	250	**83.7**	0.7	**83.7**	6.6	**83.7**	0.1	0.0 [a]	**83.7**	0.4	0.0 [a]
		500	**63.2**	11.9	66.5	20.7	**63.2**	1.9	0.0 [a]	**63.2**	1.9	0.0 [a]
		1000	**43.7**	177.1	48.4	20.6	43.9	124.0	21.9 [b]	**43.7**	59.1	21.4 [b]
$(2,5)$	500	500	167.0	4.5	168.2	60.4	167.2	9.2	0.1 [a]	**167.0**	0.8	0.0 [a]
		1000	**125.5**	702.0	135.2	104.5	125.8	272.4	9.7 [b]	**125.5**	302.0	9.4 [b]
		2000	88.2	725.3	98.9	63.2	87.9	570.4	– –	*__87.1__	603.4	– –
$(2,5)$	800	1000	**248.1**	23.1	261.7	180.3	259.5	23.0	4.6 [a]	**248.1**	18.8	0.0 [a]
		2000	181.2	1000.0	199.1	174.9	*180.7	641.5	– –	*__179.2__	643.9	– –
		5000	**134.1**	34.9	144.5	161.5	**134.1**	9.1	0.0 [a]	**134.1**	11.2	0.0 [a]
$(2,5)$	1000	1000	**333.8**	31.0	338.6	139.9	334.0	82.8	0.06 [a]	**333.8**	2.9	0.0 [a]
		5000	**169.0**	123.7	189.4	175.7	**169.0**	50.6	0.0 [a]	**169.0**	46.2	0.0 [a]
		10,000	**167.0**	94.8	172.2	207.2	**167.0**	8.6	0.0 [a]	**167.0**	23.7	0.0 [a]
Avg.			106.0	162.8	111.7	83.2	106.6	99.7		**105.8**	95.3	
$(\alpha/5,\alpha/2)$	50	100	**17.7**	<0.1	**17.7**	0.4	**17.7**	<0.1	0.0 [a]	**17.7**	0.1	0.0 [a]
		250	**9.0**	0.1	**9.0**	0.2	**9.0**	<0.1	0.0 [a]	**9.0**	0.1	0.0 [a]
		500	**5.0**	0.4	**5.0**	<0.1	**5.0**	0.1	0.0 [a]	**5.0**	<0.1	0.0 [a]
$(\alpha/5,\alpha/2)$	100	100	**50.0**	<0.1	**50.0**	0.1	**50.0**	<0.1	0.0 [a]	**50.0**	0.1	0.0 [a]
		250	**34.3**	0.1	34.9	6.7	**34.3**	<0.1	0.0 [a]	**34.3**	0.2	0.0 [a]
		500	**17.0**	0.4	17.3	6.4	**17.0**	0.2	0.0 [a]	**17.0**	0.3	0.0 [a]
$(\alpha/5,\alpha/2)$	250	250	**125.0**	0.2	**125.0**	8.9	**125.0**	0.1	0.0 [a]	**125.0**	0.5	0.0 [a]
		500	**86.8**	0.4	91.4	19.3	**86.8**	0.2	0.0 [a]	**86.8**	0.7	0.0 [a]
		1000	**51.4**	1.0	54.9	20.2	**51.4**	0.5	0.0 [a]	**51.4**	1.1	0.0 [a]
$(\alpha/5,\alpha/2)$	500	500	**250.0**	0.8	251.1	61.2	**250.0**	0.4	0.0 [a]	**250.0**	1.3	0.0 [a]
		1000	**172.9**	1.4	183.5	36.5	**172.9**	0.7	0.0 [a]	**172.9**	1.6	0.0 [a]
		2000	**101.3**	5.7	110.8	59.8	**101.3**	3.8	0.0 [a]	**101.3**	3.0	0.0 [a]
$(\alpha/5,\alpha/2)$	800	1000	**400.0**	2.1	401.6	85.9	**400.0**	0.9	0.0 [a]	**400.0**	3.2	0.0 [a]
		2000	**273.4**	3.1	292.2	107.6	**273.4**	2.5	0.0 [a]	**273.4**	3.9	0.0 [a]
		5000	**115.0**	74.3	127.0	101.8	**115.0**	74.3	0.0 [a]	**115.0**	45.4	0.0 [a]
$(\alpha/5,\alpha/2)$	1000	1000	**500.0**	3.0	505.6	108.9	**500.0**	1.4	0.0 [a]	**500.0**	5.0	0.0 [a]
		5000	**168.1**	131.6	188.0	118.4	**168.1**	59.5	0.0 [a]	**168.1**	39.3	0.0 [a]
		10,000	105.7	841.6	104.7	123.0	*__102.0__	254.6	– –	109.8	630.9	– –
Avg.			137.9	66.6	142.8	50.9	**137.7**	22.2		138.2	40.9	
$[1,\alpha]$	50	100	**13.7**	0.1	**13.7**	0.8	14.9	<0.1	8.8 [a]	**13.7**	0.1	0.0 [a]
		250	**7.2**	0.4	**7.2**	0.6	8.4	<0.1	16.7 [a]	**7.2**	0.1	0.0 [a]
		500	**3.9**	1.2	**3.9**	<0.1	4.1	<0.1	5.1 [a]	**3.9**	<0.1	0.0 [a]
$[1,\alpha]$	100	100	**40.1**	0.1	**40.1**	<0.1	**40.1**	<0.1	0.0 [a]	**40.1**	<0.1	0.0 [a]
		250	**23.3**	0.7	23.7	7.4	26.4	<0.1	13.3 [a]	**23.3**	0.2	0.0 [a]
		500	**13.5**	5.7	14.1	5.9	15.7	0.2	16.3 [a]	**13.5**	1.9	0.0 [a]
$[1,\alpha]$	250	250	**98.9**	0.3	99.0	6.6	**98.9**	0.1	0.0 [a]	**98.9**	0.2	0.0 [a]
		500	**67.3**	3.4	71.5	18.7	72.1	0.2	7.1 [a]	**67.3**	1.0	0.0 [a]
		1000	40.5	948.8	44.6	27.3	46.8	2.2	30.0 [b]	**40.3**	376.2	11.9 [b]
$[1,\alpha]$	500	500	**202.7**	0.9	203.2	28.3	**202.7**	0.6	0.0 [a]	**202.7**	1.8	0.0 [a]
		1000	**135.9**	51.8	147.7	55.0	137.8	92.8	1.4 [a]	136.0	25.1	0.07 [a]
		2000	83.1	1000.0	92.6	99.8	83.5	527.2	– –	**81.8**	164.1	– –
$[1,\alpha]$	800	1000	**300.2**	2.1	310.5	90.1	**300.2**	2.2	0.0 [a]	**300.2**	4.6	0.0 [a]
		2000	186.6	1000.0	208.2	124.8	190.2	554.6	– –	*__185.8__	163.6	– –
		5000	100.3	985.7	104.8	227.2	101.6	608.0	– –	*__97.3__	590.1	– –
$[1,\alpha]$	1000	1000	**400.8**	3.6	405.0	146.0	**400.8**	3.5	0.0 [a]	**400.8**	6.8	0.0 [a]
		5000	149.5	949.9	156.4	335.5	147.3	730.8	– –	*__142.7__	619.9	– –
		10,000	132.9	1000.0	92.4	387.0	91.6	644.9	– –	*__85.6__	670.4	– –
Avg.			111.1	330.8	113.3	86.7	110.2	176.0		**107.8**	145.9	
Total avg.			118.4	186.8	122.6	73.6	118.2	99.3		**117.3**	94.0	

Table 7. Results for Unit Disk Graphs with uniform capacity.

Capacity	n	Range	CPLEX		LS_PD		CMSA++			BARRAKUDA		
			$\overline{q}$	$\overline{t}$	$\overline{q}$	$\overline{t}$	$\overline{q}$	$\overline{t}$	gap (%)	$\overline{q}$	$\overline{t}$	gap (%)
2	50	150	**17.2**	<0.1	**17.2**	0.1	**17.2**	0.1	0.0 [a]	**17.2**	1.8	0.0 [a]
2	50	200	**17.0**	0.1	**17.0**	<0.1	**17.0**	<0.1	0.0 [a]	**17.0**	<0.1	0.0 [a]
2	100	150	**34.0**	0.1	34.5	0.1	**34.0**	0.1	0.0 [a]	**34.0**	0.1	0.0 [a]
2	100	200	**34.0**	0.2	34.1	<0.1	**34.0**	<0.1	0.0 [a]	**34.0**	0.1	0.0 [a]
2	250	150	**84.0**	0.8	86.0	<0.1	**84.0**	0.2	0.0 [a]	**84.0**	0.3	0.0 [a]
2	250	200	**84.0**	0.9	85.4	<0.1	**84.0**	0.2	0.0 [a]	**84.0**	0.4	0.0 [a]
2	500	150	**167.0**	3.1	171.6	0.2	**167.0**	0.8	0.0 [a]	**167.0**	6.2	0.0 [a]
2	500	200	**167.0**	4.4	170.3	0.1	**167.0**	0.9	0.0 [a]	**167.0**	12.1	0.0 [a]
2	800	150	**267.0**	8.0	273.9	0.2	**267.0**	2.2	0.0 [a]	**267.0**	25.2	0.0 [a]
2	800	200	**267.0**	11.7	272.4	0.1	**267.0**	2.2	0.0 [a]	**267.0**	46.3	0.0 [a]
2	1000	150	**334.0**	13.9	342.6	<0.1	**334.0**	3.0	0.0 [a]	**334.0**	45.7	0.0 [a]
2	1000	200	**334.0**	19.1	340.4	<0.1	**334.0**	3.7	0.0 [a]	**334.0**	82.4	0.0 [a]
Avg.			**150.5**	5.7	153.8	0.1	**150.5**	1.1		**150.5**	18.4	
5	50	150	**12.9**	<0.1	**12.9**	0.1	13.1	<0.1	1.6 [a]	**12.9**	<0.1	0.0 [a]
5	50	200	**10.0**	0.1	**10.0**	<0.1	10.1	0.1	1.0 [a]	**10.0**	<0.1	0.0 [a]
5	100	150	**18.7**	0.3	18.8	<0.1	**18.7**	0.4	0.0 [a]	**18.7**	0.2	0.0 [a]
5	100	200	**17.4**	0.4	**17.4**	0.1	**17.4**	0.1	0.0 [a]	**17.4**	0.2	0.0 [a]
5	250	150	**42.0**	3.5	43.7	0.4	**42.0**	0.2	0.0 [a]	**42.0**	1.1	0.0 [a]
5	250	200	**42.0**	3.5	43.0	<0.1	**42.0**	0.2	0.0 [a]	**42.0**	1.1	0.0 [a]
5	500	150	**84.0**	25.7	86.2	0.2	**84.0**	0.6	0.0 [a]	**84.0**	0.9	0.0 [a]
5	500	200	**84.0**	50.7	85.0	<0.1	**84.0**	0.5	0.0 [a]	**84.0**	1.1	0.0 [a]
5	800	150	**134.0**	215.9	137.0	1.0	**134.0**	1.3	0.0 [a]	**134.0**	2.3	0.0 [a]
5	800	200	**134.0**	387.9	135.7	0.1	**134.0**	1.8	0.0 [a]	**134.0**	3.2	0.0 [a]
5	1000	150	**167.0**	459.6	171.0	2.7	**167.0**	2.1	0.0 [a]	**167.0**	3.8	0.0 [a]
5	1000	200	173.3	787.6	169.6	<0.1	**167.0**	2.3	– –	**167.0**	5.3	– –
Avg.			76.6	175.9	77.5	0.7	**76.1**	0.8		**76.1**	1.6	
α	50	150	**14.4**	<0.1	**14.4**	79.4	**14.4**	<0.1	0.0 [a]	**14.4**	<0.1	0.0 [a]
α	50	200	**10.0**	0.1	**10.0**	22.5	10.1	<0.1	1.0 [a]	**10.0**	<0.1	0.0 [a]
α	100	150	**18.7**	0.3	18.9	<0.1	**18.7**	0.3	0.0 [a]	**18.7**	0.2	0.0 [a]
α	100	200	**11.0**	0.4	**11.0**	<0.1	**11.0**	0.1	0.0 [a]	**11.0**	0.2	0.0 [a]
α	250	150	**18.5**	4.5	19.4	8.3	**18.5**	2.9	0.0 [a]	18.6	1.5	0.5 [a]
α	250	200	**11.3**	6.3	11.5	1.7	**11.3**	2.0	0.0 [a]	11.4	0.9	0.9 [a]
α	500	150	**18.6**	164.1	20.7	250.1	18.7	72.3	0.5 [a]	19.1	14.2	2.7 [a]
α	500	200	**11.3**	90.9	12.1	129.5	**11.3**	31.9	0.0 [a]	11.6	16.0	2.7 [a]
α	800	150	19.4	906.2	21.3	319.7	*__19.2__	239.2	1.6 [b]	19.4	80.7	2.1 [b]
α	800	200	12.3	733.4	12.6	21.4	*__11.9__	34.6	1.7 [b]	12.0	30.5	2.6 [b]
α	1000	150	25.8	942.4	21.5	2.6	*__19.4__	259.7	1.6 [b]	19.6	143.0	2.6 [b]
α	1000	200	14.7	953.4	12.9	<0.1	**12.0**	104.9	0.0 [b]	**12.0**	99.1	0.0 [b]
Avg.			15.5	345.6	15.5	92.8	**14.7**	62.3		14.8	32.2	
Total avg.			80.9	175.7	82.3	31.2	**80.4**	21.4		80.5	17.4	

Table 8. Results for Unit Disk Graphs with variable capacity.

Capacity	n	Range	CPLEX		LS_PD		CMSA++			BARRAKUDA		
			$\overline{q}$	$\overline{t}$	$\overline{q}$	$\overline{t}$	$\overline{q}$	$\overline{t}$	gap (%)	$\overline{q}$	$\overline{t}$	gap (%)
$(2,5)$	50	150	**14.5**	<0.1	**14.5**	<0.1	14.9	<0.1	2.8 [a]	**14.5**	0.4	0.0 [a]
$(2,5)$	50	200	**111.1**	0.1	**11.1**	<0.1	11.2	<0.1	0.9 [a]	**11.1**	0.1	0.0 [a]
$(2,5)$	100	150	**21.8**	1.1	**21.8**	3.5	22.0	0.1	0.9 [a]	21.9	44.7	0.5 [a]
$(2,5)$	100	200	**17.6**	0.4	17.8	3.7	**17.6**	0.9	0.0 [a]	**17.6**	0.3	0.0 [a]
$(2,5)$	250	150	**42.1**	2.6	43.0	18.4	**42.1**	0.4	0.0 [a]	**42.1**	0.9	0.0 [a]
$(2,5)$	250	200	**42.0**	2.3	**42.0**	36.0	**42.0**	0.2	0.0 [a]	**42.0**	0.3	0.0 [a]
$(2,5)$	500	150	**84.0**	11.7	84.9	35.7	**84.0**	0.8	0.0 [a]	**84.0**	5.3	0.0 [a]
$(2,5)$	500	200	**84.0**	15.2	**84.0**	33.4	**84.0**	0.8	0.0 [a]	**84.0**	9.6	0.0 [a]
$(2,5)$	800	150	**134.0**	55.4	134.9	251.8	**134.0**	1.7	0.0 [a]	**134.0**	20.5	0.0 [a]
$(2,5)$	800	200	**134.0**	71.1	134.1	595.3	**134.0**	2.0	0.0 [a]	**134.0**	40.0	0.0 [a]
$(2,5)$	1000	150	**167.0**	105.1	168.6	235.4	**167.0**	2.9	0.0 [a]	**167.0**	37.2	0.0 [a]
$(2,5)$	1000	200	**167.0**	119.0	167.5	408.1	**167.0**	3.3	0.0 [a]	**167.0**	70.6	0.0 [a]
Avg.			**76.6**	32.0	77.0	135.1	76.7	1.1		**76.6**	19.2	
$(\alpha/5,\alpha/2)$	50	150	**25.4**	0.1	**25.4**	<0.1	**25.4**	<0.1	0.0 [a]	**25.4**	<0.1	0.0 [a]
$(\alpha/5,\alpha/2)$	50	200	**17.3**	<0.1	**17.3**	0.1	**17.3**	<0.1	0.0 [a]	**17.3**	<0.1	0.0 [a]
$(\alpha/5,\alpha/2)$	100	150	**29.9**	0.2	30.3	6.7	**29.9**	0.6	0.0 [a]	**29.9**	0.1	0.0 [a]
$(\alpha/5,\alpha/2)$	100	200	**17.8**	0.4	**17.8**	3.4	**17.8**	0.1	0.0 [a]	**17.8**	0.1	0.0 [a]
$(\alpha/5,\alpha/2)$	250	150	**31.6**	4.4	32.4	25.2	**31.6**	0.5	0.0 [a]	**31.6**	0.9	0.0 [a]
$(\alpha/5,\alpha/2)$	250	200	**18.9**	26.3	19.2	13.3	**18.9**	0.5	0.0 [a]	**18.9**	3.6	0.0 [a]
$(\alpha/5,\alpha/2)$	500	150	**32.0**	661.5	32.9	35.9	**32.0**	2.5	0.0 [a]	**32.0**	2.4	0.0 [a]
$(\alpha/5,\alpha/2)$	500	200	79.6	602.3	**19.4**	116.3	**19.4**	3.1	73.2 [b]	**19.4**	1.6	73.2 [b]
$(\alpha/5,\alpha/2)$	800	150	732.0	636.5	33.3	91.8	**31.7**	28.5	66.8 [b]	**31.7**	29.0	66.8 [b]
$(\alpha/5,\alpha/2)$	800	200	796.6	1000.0	19.8	100.1	**19.6**	4.7	67.5 [b]	**19.6**	4.6	67.5 [b]
$(\alpha/5,\alpha/2)$	1000	150	997.8	1000.0	33.5	173.6	**32.5**	13.3	70.2 [b]	**32.5**	7.2	70.2 [b]
$(\alpha/5,\alpha/2)$	1000	200	996.0	1000.0	20.0	88.2	**19.2**	15.2	60.0 [b]	**19.2**	9.5	60.0 [b]
Avg.			314.6	411.0	25.1	54.6	**24.6**	5.8		**24.6**	4.9	
$[1,\alpha]$	50	150	**16.7**	<0.1	**16.7**	0.1	17.1	<0.1	2.4 [a]	**16.7**	<0.1	0.0 [a]
$[1,\alpha]$	50	200	**12.1**	0.1	**12.1**	0.5	12.6	<0.1	4.1 [a]	**12.1**	<0.1	0.0 [a]
$[1,\alpha]$	100	150	**21.4**	0.3	**21.4**	4.3	22.0	<0.1	2.8 [a]	**21.4**	0.1	0.0 [a]
$[1,\alpha]$	100	200	**12.8**	1.5	13.0	0.8	13.1	1.1	2.3 [a]	**12.8**	0.2	0.0 [a]
$[1,\alpha]$	250	150	**20.8**	38.0	21.9	6.1	22.1	4.9	1.4 [a]	**20.8**	17.4	0.0 [a]
$[1,\alpha]$	250	200	**12.8**	58.2	13.0	5.5	13.9	1.0	8.6 [a]	**12.8**	1.9	0.0 [a]
$[1,\alpha]$	500	150	21.9	940.6	22.6	55.7	22.8	12.2	18.9 [b]	**21.1**	161.3	14.1 [b]
$[1,\alpha]$	500	200	13.2	926.1	13.0	53.0	13.5	11.6	20.5 [b]	**12.7**	31.5	13.4 [b]
$[1,\alpha]$	800	150	22.7	973.2	22.7	163.0	23.0	173.8	21.1 [b]	*__21.0__	324.4	10.5 [b]
$[1,\alpha]$	800	200	726.9	997.4	13.2	76.5	14.0	70.9	19.7 [b]	*__12.5__	94.3	6.8 [b]
$[1,\alpha]$	1000	150	909.2	1000.0	22.6	500.4	22.5	191.3	17.8 [b]	*__21.0__	495.9	9.9 [b]
$[1,\alpha]$	1000	200	1000.0	1000.0	13.0	319.3	13.9	26.4	15.8 [b]	**12.6**	162.8	5.0 [b]
Avg.			232.5	494.6	17.1	98.8	17.5	41.1		**16.5**	107.5	
Total avg.			207.9	312.5	39.7	96.1	39.6	16.0		39.2	43.9	

The numerical results for the large problem instances from the new benchmark set are provided in Tables 9 and 10. Note that the results for uniform capacity instances are shown in sub-tables (a) and for variable capacity instances in subtables (b). Finally, the corresponding critical difference plots are shown in Figure 5. Most notably, CPLEX is now clearly worse than CMSA++ and BARRAKUDA. In fact, in most cases, CPLEX can only find the trivial solutions which contain all nodes of the input graph. Moreover, it is interesting to note that BARRAKUDA clearly outperforms CMSA++ in the case of the random graph instances (see Table 9 and Figure 5b), while both methods perform statistically equivalent in the case of the random geometric graphs (see Table 10 and Figure 5c). Finally, it also worth mentioning that the MDS-based lower bound could be computed for nearly all random geometric graphs—apart from the cases with 5000 nodes and a low number of edges (S)—while this was not possible in the case of the random graphs (due to excessive computation times). Note that when

the gap (see column with heading "gap (%)") is equal to zero, the algorithm's result is equal to the MDS-based lower bound, which means that the optimal solutions are obtained in this these cases.

Table 9. Experimental results for the random graphs from the new set of large problem instances.

Capacity	n	#Edges	CPLEX			CMSA++			BARRAKUDA		
			$\overline{q}$	$\overline{t}$	$\overline{g}$	$\overline{q}$	$\overline{t}$	gap (%)	$\overline{q}$	$\overline{t}$	gap (%)
		S	1000	1000.7	100	48.3	552.4	– –	**46.1**	675.8	– –
α	1000	M	1000	1000.8	100	20.1	392.2	– –	**19.9**	302.8	– –
		L	1000	1000.9	100	12.9	241.1	– –	12.9	146.6	– –
		S	5000	1018.2	100	79.2	227.7	– –	**70.7**	777.4	– –
α	5000	M	5000	1021.4	100	31.1	569.7	– –	**28.4**	328.9	– –
		L	5000	1024.8	100	19.1	167.6	– –	**17.9**	190.6	– –
Avg.			3000.0	1011.1	100	35.1	358.5		**32.65**	403.7	

(**a**) Uniform capacity problems.

Capacity	n	#Edges	CPLEX			CMSA++			BARRAKUDA		
			$\overline{q}$	$\overline{t}$	$\overline{g}$	$\overline{q}$	$\overline{t}$	gap (%)	$\overline{q}$	$\overline{t}$	gap (%)
		S	60	308	30.5	50.8	565.4	– –	**49.3**	690.3	– –
$[1, \alpha]$	1000	M	1000	1000.8	100	20.5	438.7	– –	**20.0**	444.1	– –
		L	1000	1001	100	12.9	252.9	– –	**12.8**	294.6	– –
		S	5000	1018.2	100	84.4	481.2	– –	**72.3**	700.0	– –
$[1, \alpha]$	5000	M	5000	1022.6	100	31.8	635.8	– –	**28.8**	197.5	– –
		L	5000	1025.6	100	20.2	236.5	– –	**18.0**	171.6	– –
Avg.			2843.3	896.0	88.4	36.8	435.1		**33.5**	416.3	

(**b**) Variable capacity problems.

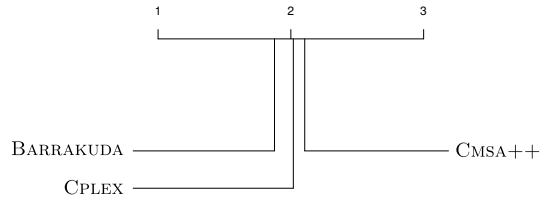
Table 10. Experimental results for the random geometric graphs from the new set of large problem instances.

Capacity	n	#Edges	CPLEX			CMSA++			BARRAKUDA		
			$\overline{q}$	$\overline{t}$	$\overline{g}$	$\overline{q}$	$\overline{t}$	gap (%)	$\overline{q}$	$\overline{t}$	gap (%)
		S	22.7	287.8	10.1	**22.1**	200.3	0.9 [b]	22.2	357.2	1.4 [b]
α	1000	M	10.6	689.3	11.9	**9.0**	14.0	0.0 [b]	**9.0**	91.8	0.0 [b]
		L	10.1	449.4	41.0	**4.9**	69.0	4.3 [b]	5.0	8.6	6.4 [b]
		S	4084.9	1018.8	100	**26.5**	720.3	– –	27.5	736.3	– –
α	5000	M	3926.0	1022.1	100	10.0	298.8	11.1 [b]	**9.1**	690.3	1.1 [b]
		L	5000	1032.2	100	5.2	350.6	16.0 [b]	**5.0**	477.1	0.0 [b]
Avg.			2175.7	749.9	60.5	**13.0**	275.5		**13.0**	393.6	

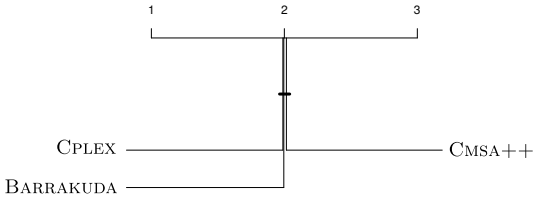
(**a**) Uniform capacity problems.

Capacity	n	#Edges	CPLEX			CMSA++			BARRAKUDA		
			$\overline{q}$	$\overline{t}$	$\overline{g}$	$\overline{q}$	$\overline{t}$	gap (%)	$\overline{q}$	$\overline{t}$	gap (%)
		S	26.7	321.2	16.7	24.5	225.6	12.4 [b]	24.5	452.7	12.4 [b]
$[1, \alpha]$	1000	M	334.5	875.9	81.7	9.0	120.6	0.0 [b]	9.0	55.3	0.0 [b]
		L	958.6	951.9	100	7.2	7.3	44.0 [b]	**5.0**	51.6	0.0 [b]
		S	5000	1018.2	100	**29.4**	646.6	– –	32.0	818.5	– –
$[1, \alpha]$	5000	M	5000	1022.6	100	**10.5**	598.8	16.7 [b]	10.8	1229.9	20.0 [b]
		L	5000	1025.6	100	5.8	551.9	16.0 [b]	**5.0**	1138.2	0.0 [b]
Avg.			2720.0	869.2	83.1	**14.4**	358.5		**14.4**	624.4	

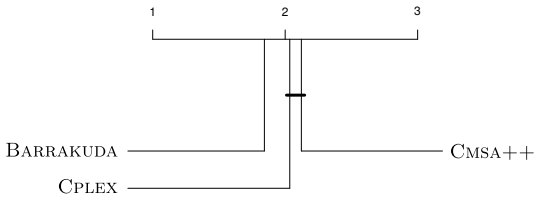
(**b**) Variable capacity problems.



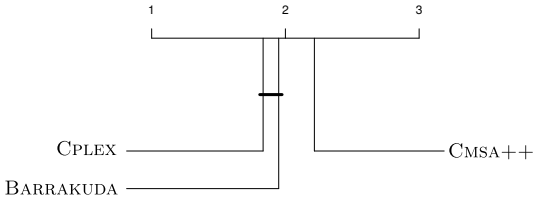
(**a**) All instances from the literature.



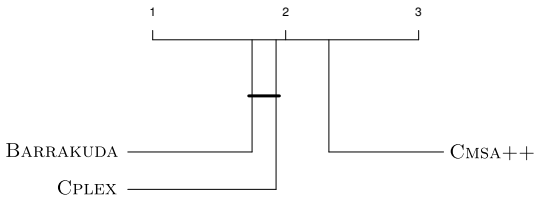
(**b**) Uniform capacity graphs, low capacities.



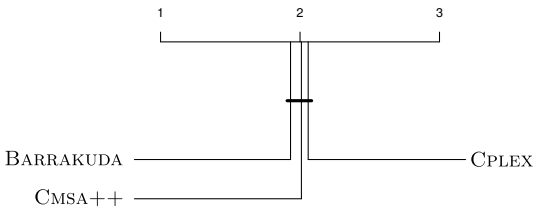
(**c**) Variable capacity graphs, low capacities.



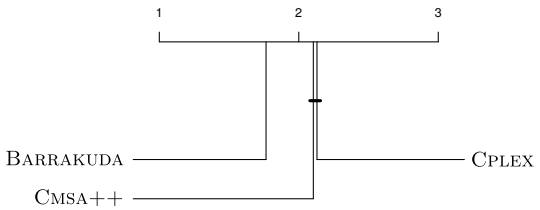
(**d**) Uniform capacity graphs, med. capacities.



(**e**) Variable capacity graphs, med. capacities.

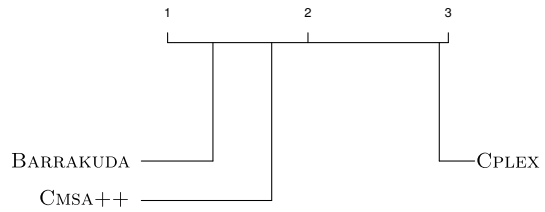


(**f**) Uniform capacity graphs, high capacities.

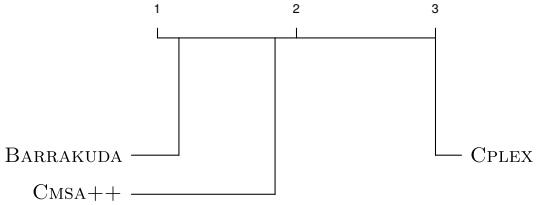


(**g**) Variable capacity graphs, high capacities.

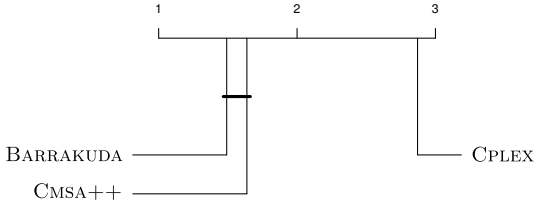
Figure 4. Critical difference plots concerning the problem instances from the literature.



(**a**) All instances from this set.



(**b**) Random graph instances.



(**c**) Random geometric graph instances.

Figure 5. Critical difference plots concerning the challenging large problem instances.

5. Conclusions and Future Work

In this paper we dealt with an NP-hard variant of the family of dominating set problems—the so-called minimum capacitated dominating set problem. Two algorithms were proposed to tackle this problem. The first one is an extended version of construct, merge, solved and adapt, of which a preliminary version was already published [11]. The aim of our extensions was to make the algorithm more robust with respect to the parameter value settings. The second algorithm that was proposed is a hybrid between a biased random key genetic algorithm and an exact solver. This solver was labeled BARRAKUDA. The main idea of Barrakuda is to use, at each iteration, the current population of individuals in order to generate a sub-instance of the tackled problem instances. This sub-instance is then solved by the exact solver (CPLEX) and the resulting solution is transformed into an individual and fed back into the population. The experimental results showed that both algorithms clearly outperform LS_PD, the currently best approach from the literature in the context of already published benchmark instances. However, the results also showed that these instances are not really a challenge for our algorithms. Therefore, we generated a new set of larger and more challenging benchmark instances. BARRAKUDA clearly outperformed the extended version of construct, merge, solve and adapt, especially in the context of general random graphs. In contrast, the performance of the two algorithms was shown to be quite similar for random geometric graphs.

Concerning future work, we want to study the reasons why BARRAKUDA performs (in relation to construct, merge and adapt) much better for general random graphs, while this difference does not show for random geometric graphs. Moreover, we believe that BARRAKUDA-type algorithms—that is, hybrid algorithms that (1) take profit from an exact solver and (2) incorporate a learning component—can be very successful for other types of combinatorial optimization problems, and this is something that we would like to study.

Author Contributions: Both authors contributed equally in all aspects of this research. All authors have read and agreed to the published version of the manuscript

Funding: This work was supported by project CI-SUSTAIN funded by the Spanish Ministry of Science and Innovation (PID2019-104156GB-I00).

Acknowledgments: We acknowledge administrative and technical support by the Spanish National Research Council (CSIC) and the Universidad de Concepción, Chile.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Yu, J.Y.; Chong, P.H.J. A survey of clustering schemes for mobile ad hoc networks. *IEEE Commun. Surv. Tutor.* **2005**, *7*, 32–48.
2. Rajaraman, R. Topology control and routing in ad hoc networks: A survey. *ACM SIGACT News* **2002**, *33*, 60–73.
3. Moscibroda, T. Clustering. In *Algorithms for Sensor and Ad Hoc Networks: Advanced Lectures*; Wagner, D., Wattenhofer, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 37–61.
4. Garey, M.R.; Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*; W. H. Freeman & Co.: New York, NY, USA, 1979.
5. Cygan, M.; Pilipczuk, M.; Wojtaszczyk, J.O. Capacitated Domination Faster Than O(2n). In *Algorithm Theory-SWAT 2010*; Lecture Notes in Computer Science; Kaplan, H., Ed.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 74–80.
6. Liedloff, M.; Todinca, I.; Villanger, Y. Solving Capacitated Dominating Set by Using Covering by Subsets and Maximum Matching. In *Graph Theoretic Concepts in Computer Science*; Lecture Notes in Computer Science Thilikos, D.M., Ed.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 88–99.
7. Kuhn, F.; Moscibroda, T. Distributed approximation of capacitated dominating sets. *Theory Comput. Syst.* **2010**, *47*, 811–836.

8. Potluri, A.; Singh, A. A Greedy Heuristic and Its Variants for Minimum Capacitated Dominating Set. In *Contemporary Computing*; Communications in Computer and Information Science; Parashar, M.E.T., Ed.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 28–39.
9. Potluri, A.; Singh, A. Metaheuristic algorithms for computing capacitated dominating set with uniform and variable capacities. *Swarm Evol. Comput.* **2013**, *13*, 22–33.
10. Li, R.; Hu, S.; Zhao, P.; Zhou, Y.; Yin, M. A novel local search algorithm for the minimum capacitated dominating set. *J. Oper. Res. Soc.* **2018**, *69*, 849–863.
11. Pinacho-Davidson, P.; Bouamama, S.; Blum, C. Application of CMSA to the Minimum Capacitated Dominating Set Problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*; ACM: New York, NY, USA, 2019; pp. 321–328, doi:10.1145/3321707.3321807.
12. Li, R.; Hu, S.; Liu, H.; Li, R.; Ouyang, D.; Yin, M. Multi-Start Local Search Algorithm for the Minimum Connected Dominating Set Problems. *Mathematics* **2019**, *7*, 1173.
13. Li, B.; Zhang, X.; Cai, S.; Lin, J.; Wang, Y.; Blum, C. NuCDS: An Efficient Local Search Algorithm for Minimum Connected Dominating Set. In Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence IJCAI-20, Yokohama, Japan, 11–17 July 2020; Bessiere, C., Ed.; International Joint Conferences on Artificial Intelligence Organization: LA, CA, USA, 2020; pp. 1503–1510.
14. Bouamama, S.; Blum, C.; Fages, J.G. An algorithm based on ant colony optimization for the minimum connected dominating set problem. *Appl. Soft Comput.* **2019**, *80*, 672–686.
15. Wang, Y.; Li, C.; Yin, M. A two phase removing algorithm for minimum independent dominating set problem. *Appl. Soft Comput.* **2020**, *88*, 105949.
16. Wang, Y.; Chen, J.; Sun, H.; Yin, M. A memetic algorithm for minimum independent dominating set problem. *Neural Comput. Appl.* **2018**, *30*, 2519–2529.
17. Wang, Y.; Pan, S.; Li, C.; Yin, M. A local search algorithm with reinforcement learning based repair procedure for minimum weight independent dominating set. *Inf. Sci.* **2020**, *512*, 533–548.
18. Zhou, Y.; Li, J.; Liu, Y.; Lv, S.; Lai, Y.; Wang, J. Improved Memetic Algorithm for Solving the Minimum Weight Vertex Independent Dominating Set. *Mathematics* **2020**, *8*, 1155.
19. Yuan, F.; Li, C.; Gao, X.; Yin, M.; Wang, Y. A novel hybrid algorithm for minimum total dominating set problem. *Mathematics* **2019**, *7*, 222.
20. Cornejo Acosta, J.A.; García Díaz, J.; Menchaca-Méndez, R.; Menchaca-Méndez, R. Solving the Capacitated Vertex K-Center Problem through the Minimum Capacitated Dominating Set Problem. *Mathematics* **2020**, *8*, 1551.
21. Conejo, A.J.; Castillo, E.; Minguez, R.; Garcia-Bertrand, R. *Decomposition Techniques in Mathematical Programming: Engineering and Science Applications*; Springer Science & Business Media: New York, NY, USA, 2006.
22. Boschetti, M.A.; Maniezzo, V.; Roffilli, M.; Bolufé Röhler, A. Matheuristics: Optimization, Simulation and Control. In *Proceedings of HM 2009—6th International Workshop on Hybrid Metaheuristics*; Lecture Notes in Computer Science; Blesa, M.J., Blum, C., Di Gaspero, L., Roli, A., Sampels, M., Schaerf, A., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 171–177.
23. Pisinger, D.; Røpke, S. Large Neighborhood Search. In *Handbook of Metaheuristics*; International Series in Operations Research & Management Science; Gendreau, M., Potvin, J.Y., Eds.; Springer: New York, NY, USA, 2010; pp. 399–419.
24. Ahuja, R.K.; Orlin, J.B.; Sharma, D. Very large-scale neighborhood search. *Int. Trans. Oper. Res.* **2000**, *7*, 301–317.
25. Schrimpf, G.; Schneider, J.; Stamm-Wilbrandt, H.; Dueck, G. Record breaking optimization results using the ruin and recreate principle. *J. Comput. Phys.* **2000**, *159*, 139–171.
26. Fischetti, M.; Lodi, A. Local branching. *Math. Program.* **2003**, *98*, 23–47.
27. Caserta, M.; Voß, S. A corridor method based hybrid algorithm for redundancy allocation. *J. Heuristics* **2016**, *22*, 405–429.
28. Lalla-Ruiz, E.; Voß, S. POPMUSIC as a matheuristic for the berth allocation problem. *Ann. Math. Artif. Intell.* **2016**, *76*, 173–189.
29. Blum, C.; Pinacho, P.; López-Ibánez, M.; Lozano, J.A. Construct, Merge, Solve & Adapt: A new general algorithm for combinatorial optimization. *Comput. Oper. Res.* **2016**, *68*, 75–88, doi:10.1016/j.cor.2015.10.014.

30. Blum, C.; Raidl, G.R. *Hybrid Metaheuristics, Powerful Tools for Optimization*; Springer International Publishing: Cham, Switzerland, 2016.
31. Gonçalves, J.F.; Resende, M.G.C. Biased random-key genetic algorithms for combinatorial optimization. *J. Heuristics* **2011**, *17*, 487–525, doi:10.1007/s10732-010-9143-1.
32. Blum, C.; Gambini Santos, H. Generic CP-Supported CMSA for Binary Integer Linear Programs. In *Hybrid Metaheuristics*; Blesa Aguilera, M.J., Blum, C., Gambini Santos, H., Pinacho-Davidson, P., Godoy del Campo, J., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 1–15.
33. Mastrogiovanni, M. The Clustering Simulation Framework: A Simple Manual. 2007. Available online: https://www.researchgate.net/publication/265429652_The_Clustering_Simulation_Framework_A_Simple_Manual (accessed on 11 March 2020).
34. Shyu, S.J.; Yin, P.Y.; Lin, B.M. An ant colony optimization algorithm for the minimum weight vertex cover problem. *Ann. Oper. Res.* **2004**, *131*, 283–304.
35. López-Ibánez, M.; Dubois-Lacoste, J.; Cáceres, L.P.; Birattari, M.; Stützle, T. The irace package: Iterated racing for automatic algorithm configuration. *Oper. Res. Perspect.* **2016**, *3*, 43–58, doi:10.1016/j.orp.2016.09.002.
36. García, S.; Herrera, F. An Extension on "Statistical Comparisons of Classifiers over Multiple Data Sets" for all Pairwise Comparisons. *J. Mach. Learn. Res.* **2008**, *9*, 2677–2694.
37. Calvo, B.; Santafé, G. scmamp: Statistical Comparison of Multiple Algorithms in Multiple Problems. *R J.* **2016**, *8*, 2073–4859.