

# Constraint generation approaches for submodular function maximization leveraging graph properties

Eszter Csókás<sup>1</sup> and Tamás Vinkó<sup>1</sup>

<sup>1</sup>Department of Computational Optimization, University of Szeged,  
Hungary.

Contributing authors: [csokas@inf.u-szeged.hu](mailto:csokas@inf.u-szeged.hu);  
[tvinko@inf.u-szeged.hu](mailto:tvinko@inf.u-szeged.hu);

## Abstract

Submodular function maximization is an attractive optimization model and also a well-studied problem with a variety of algorithms available. Constraint generation (CG) approaches are appealing techniques to tackle the problem with, as the mixed-integer programming formulation of the problem suffers from the exponential size of the number of constraints. Most of the problems in these topics are of combinatorial nature and involve graphs on which the maximization is defined. Inspired by the recent work of Uematsu *et al.* [1], in this paper we introduce variants of the CG algorithm which take into account certain properties of the input graph aiming at informed selection of the constraints. Benchmarking results are shown to demonstrate the efficiency of the proposed methods.

**Keywords:** Submodular function maximization, combinatorial optimization, integer programming

## 1 Introduction

In recent years, the theory of submodular maximization has been improved and it has played a key role in extraordinarily varied application areas [2]. Examples include several classes of important combinatorial optimization problems [2], namely, the Simple or “uncapacitated” Plant (facility) Location Problem (SPLP) and its competitive version [3], the Quadratic Cost Partition Problem (QCP) with non-negative

## 2 Constraint generation approaches for submodular function maximization

edge weights, and its special case – the Max-Cut Problem [4], the generalized transportation problem. Many different problems of data mining and knowledge discovery in biomedical and bioinformatics research (e.g., diagnostic hypothesis generation, logical methods of data analysis, conceptual clustering, and proteins functional annotations) as well as applied to the statistics, machine learning and experimental design [5, 6], multiobject tracking [7], sparse reconstruction [8], influence spread [9], and also combining multiple heuristics online [10]. There are models in mathematics [2], including the rank function of elementary linear algebra, which is a special case of matroid rank functions [11, 12] require the solution of submodular maximization problem.

Solving submodular optimization problems on graphs are also popular line of study as set functions can easily be defined on graphs. The objectives in these graph based problems vary from simultaneous localization and mapping problem for robots [13], route planning, such as mobile robot sensing and door-to-door marketing [14], and investigation of more general class involving, e.g.,  $s$ - $t$  path constraint [15].

This paper contributes to the research effort invested into submodular function maximization with cardinality constraints by introducing efficient variations of a recently proposed constraint generation algorithm by [1].

**Submodular function.** Let  $N = \{1, \dots, n\}$  be a finite set. The function  $f : 2^N \rightarrow \mathbb{R}$  is called *submodular* if it fulfills  $f(S) + f(T) \geq f(S \cap T) + f(S \cup T)$  for all  $S, T \subseteq N$ . There are many natural linkage between submodular functions and both convexity and concavity, see, e.g., in [10]. A submodular function  $f$  is called *non-decreasing* if  $f(S) \leq f(T)$  holds for all  $S \subseteq T \subseteq N$ . In the rest of the paper it is assumed that  $f$  fulfills this property, i.e., it is a non-decreasing submodular function.

The submodular maximization problem with a cardinality constant  $k$  (where  $0 < k \leq n$ ) is defined in the following way:

$$\begin{aligned} \max \quad & f(S) \\ \text{subject to} \quad & |S| \leq k, S \subseteq N. \end{aligned} \tag{1}$$

**Solvability.** Interestingly, the submodular function minimization problem can be solved in polynomial time [16, 17], whereas the non-decreasing submodular function maximization is NP-hard [18, 19]. The greedy strategy is an often applied approach to solve (1) as it guarantees the  $(1 - 1/e)$  approximation of the optimal solution [20], however, it might be computationally inefficient for large-scale instances. On the other hand, many applications expect a globally optimal result, and that was the aim of the approach proposed in [21]. The algorithm is based on the following mixed integer programming (MIP) model:

$$\begin{aligned} \max \quad & z \\ \text{s.t.} \quad & z \leq f(S) + \sum_{i \in N \setminus S} f(\{i\} | S) \cdot y_i, \quad S \in F, \\ & \sum_{i \in N} y_i \leq k, \\ & y_i \in \{0, 1\}, \quad i \in N, \end{aligned} \tag{2}$$

where  $f(T \mid S) := f(S \cup T) - f(S)$  for all  $S, T \subseteq N$  and  $F$  denotes the set of all feasible solutions satisfying the cardinality constraint  $|S| \leq k$ . Notably, the number of constraints in (2) grows exponentially, hence a constraint generation (CG) algorithm was proposed in [21]. The CG approach starts with a reduced problem with a small number of constraints and then iterates the solution of the reduced problem by adding a new constraint condition. In practice, the CG algorithm solves many reduced MIP problems, which might lead to poor efficiency in applications. This can be mitigated by the branch-and-bound approach using linear programming relaxation of the MIP.

**Roadmap.** The rest of the paper is organized as follows. First, we discuss the relevant literature for the non-decreasing submodular function maximization in Section 2. Then, Section 3 is about the problem instances that we use for testing the algorithms. In Section 4 we describe the algorithms. The first one is the improved constraint generation (ICG) algorithm proposed in [1]. This is followed by the introduction of our ICG modifications, where we motivate and describe three variants. The numerical experiments are presented in Section 5, including the description of the computational environments, the properties of the test graphs and finally the details and discussion of the benchmarking results. The paper is concluded in Section 6.

## 2 Related works

Optimization of submodular functions has been actively studied. In this section we discuss the relevant contributions for the non-decreasing submodular function maximization.

The paper by [10] provides analysis on the theoretical approximation guarantees of the solver algorithm (greedy as well as more complex methods). They also considered the different types of extensions of submodular optimization such as the online settings and adaptive optimization problems. In contrast to our work, the entire paper is dedicated to approximation (incomplete) algorithms.

An  $A^*$  search algorithm was proposed in [22]. More precisely, it is a framework for solving non-decreasing submodular optimization problems. The approximate guarantee based on submodularity property is also provided for non-submodular functions. In principle, the  $A^*$  algorithm is a heuristic, whereas our approach solves the problem to optimality.

A new implementation of the CG approach was proposed in [23] to solve the problem using the formalism (2). The implementation uses lazy constraints, the modern functionality of MIP solvers, such as CPLEX. Moreover, a GRASP heuristic and two (heuristic) procedures for separating submodular cuts from fractional solutions were also developed. Based on numerical experiments, improved efficiency compared to the standard CG was shown. The paper used the same benchmark problem set which we also use, see Section 3.

[24] formulated the submodular maximization under submodular cover problem and proposed an approximation framework to solve it. The algorithm provably produces nearly optimal solutions. As this paper aims at solving a specified version of (1) its applicability differs from those of our approaches.

4 *Constraint generation approaches for submodular function maximization*

A deterministic algorithm based on a new greedy strategy for solving problem (1) was proposed by [25]. It is shown by mathematical proof that this new strategy outperforms the traditional greedy algorithm provided that function  $f$  fulfills certain assumptions.

Finally, [1] presented an improved constraint generation (ICG) algorithm. Being an iterative method, it starts from a small subset of constraints and repeatedly solves relaxed problems while adding a promising set of constraints at each iteration. The ICG method was included into a branch-and-cut algorithm to attain good upper bounds while solving a smaller number of reduced MIP problems. Computational results were obtained for well-known benchmark instances. In Section 4 we present this ICG algorithm because our work is based on it. In fact, we created three variants of it with the aim of even better computational efficiency. Note that we did not use the branch-and-cut algorithm.

### 3 Benchmark sets

We use 3 types of well-known and frequently used examples which have the non-decreasing submodular property, termed by facility location (LOC), weighted coverage (COV) and bipartite influence (INF), see [10, 26, 27]. It is important to mention that LOC and COV examples have straightforward MIP formulations, which can be solved by standard MIP solvers quite efficiently, see [1]. On the other hand, the INF problem, to be introduced below cannot be formulated as straightforward MIP and this gives reasons for the attempt to make the universal submodular maximization framework more efficient [23].

**Facility location (LOC)** . Let  $n$  be the number of locations and  $m$  be the number of clients. The set of locations  $N = \{1, \dots, n\}$  and the set of clients  $M = \{1, \dots, m\}$  are given. Define  $g_{i,j} > 0$  as a non-negative profit when client  $i \in M$  is served by location  $j \in N$ , for all possible pairs. We select a set of  $k$  locations  $S \subseteq N$  to build the facilities. Each client  $i \in M$  gets the profit from the best opened facility, and we want to maximize the overall profit, so the total benefit is defined as:

$$f(S) = \sum_{i \in M} \max_{j \in S} g_{i,j}. \quad (3)$$

**Weighted coverage (COV)** . Let  $n$  be the number of sensors and  $m$  be the number of items. The set of sensors  $N = \{1, \dots, n\}$  and the set of items  $M = \{1, \dots, m\}$  are given. Each sensor  $j \in N$  covers the subset of items  $M_j \subseteq M$  and each item  $i \in M$  has a non-negative weight  $w_i$ . To cover the items we select a set of sensors  $S \subseteq N$ . We have  $a_{i,j} = 1$ , if  $i \in M_j$  and  $a_{i,j} = 0$  otherwise. Then, the total weighted coverage is the following:

$$f(S) = \sum_{i \in M} w_i \max_{j \in S} a_{i,j}. \quad (4)$$

**Bipartite influence (INF)** . Let  $M = \{1, \dots, m\}$  be the set of targets, where  $m$  is the number of targets and  $N = \{1, \dots, n\}$  be the set of items, where  $n$  is the number of items. Define an influence maximization problem on a bipartite graph  $G = (M, N; E)$ ,

where  $E \subseteq M \times N$  is a set of directed edges. The activation probability  $p_j \in [0, 1]$  of every  $j \in N$  item is given. Let  $1 - \prod_{j \in S} (1 - q_{ij})$  be the probability that a set of items  $S \subseteq N$  activates a target  $i \in M$ , where  $q_{ij} = p_j$  if  $(i, j)$  is a directed edge in  $E$ , otherwise  $q_{ij} = 0$  holds. The definition of the number of targets activated by the element set  $S \subseteq N$  is the following:

$$f(S) = \sum_{i \in M} \left( 1 - \prod_{j \in S} (1 - q_{ij}) \right). \quad (5)$$

Note that the INF problem as a submodular function given in formula (5) contains a product involving elements from the set  $S$ , which, in case of formulating it using binary variables corresponding to set element selections would result in a polynomial type non-linear program. That *might* be possibly converted into MILP using, e.g., McCormick formalism, but it is neither straightforward nor promising with respect to its solvability.

## 4 Algorithms

In this section, first we describe the so-called improved constraint generation (ICG) solution method which was introduced recently by [1]. We propose three modifications of ICG in which either certain characteristics of the graph describing the problem is used or the submodularity property of the function to be maximized is exploited.

### 4.1 Improved constraint generation (ICG)

An improved constraint generation (ICG) algorithm was proposed in [1] and is shown in Algorithms 1 and 2. What follows is that we give a quick overview and summary of the most important concepts and notations of ICG, the reader is referred to the paper of [1] for the full details. Note that we changed Step 5 in Algorithm 1 to expand the set  $Q$  with  $S^{(t)}$  only *after* executing Algorithm 2 (SUB-ICG) since for the inputs it is assumed that  $S^{(t)} \notin Q$ .

The ICG algorithm refers to a reduced problem of (2) as  $MIP(Q)$ , where  $Q$  is a set of feasible solutions, and it is defined as follows<sup>1</sup>:

$$\begin{aligned} \max \quad & z \\ \text{s.t.} \quad & z \leq f(S) + \sum_{i \in N \setminus S} f(\{i\} | S) \cdot y_i, \quad S \in Q, \\ & \sum_{i \in N} y_i \leq k, \\ & y_i \in \{0, 1\}, \quad i \in N, \end{aligned} \quad (6)$$

Using the CG algorithm of [21] as baseline, in the  $t$ -th iteration let the optimal solution be  $\mathbf{y}^{(t)} = (y_1^{(t)}, \dots, y_n^{(t)})$  and let the optimal value of the problem  $MIP(Q)$  be  $z^{(t)}$ . In this case  $z^{(t)}$  is an upper bound for problem (2), and that is what we aim to decrease

---

<sup>1</sup>Note that problem (6) is referred as  $BIP(Q)$  in [1]

**Algorithm 1** ICG( $S^{(0)}, \lambda$ )

**Input** The initial feasible solution  $S^{(0)}$  and the number of feasible solution to be generated at each iteration  $\lambda$ .

**Output** The optimal solution  $S^*$ .

**Step 1:** Set  $Q \leftarrow S^{(0)}$ ,  $Q^+ \leftarrow \{S_{[0]}^{(0)}, \dots, S_{[k]}^{(0)}\}$ ,  $S^* \leftarrow S^{(0)}$  and  $t \leftarrow 1$ .

**Step 2:** Solve  $MIP(Q^+)$ . Let  $S^{(t)}$  and  $z^{(t)}$  be, respectively, an optimal solution and the optimal value of  $MIP(Q^+)$ .

**Step 3:** If  $f(S^{(t)}) > f(S^*)$ , then let  $S^* \leftarrow S^{(t)}$ .

**Step 4:** If  $z^{(t)} = f(S^*)$  holds, then output the solution  $S^*$  and exit.

**Step 5:** Set  $Q^+ \leftarrow Q^+ \cup \{S^{(t)}\} \cup \text{SUB-ICG}(Q, S^{(t)}, \lambda)$ ,  $Q \leftarrow Q \cup \{S^{(t)}\}$  and  $t \leftarrow t + 1$ .

**Step 6:** For each feasible solution  $S \in \text{SUB-ICG}(Q, S^{(t)}, \lambda)$ , if  $f(S) > f(S^*)$  holds, then set  $S^* \leftarrow S$ . Return to Step 2.

**Algorithm 2** SUB-ICG ( $Q, S^{(t)}, \lambda$ )

**Input** A set of feasible solutions  $Q \subseteq F$ , a feasible solution  $S^{(t)} \notin Q$  and the number of feasible solutions to be generated  $\lambda$ .

**Output** A set of feasible solutions  $Q' \subseteq F$ .

**Step 1:** Set  $Q' \leftarrow \emptyset$  and  $h \leftarrow 1$ .

**Step 2:** Select a feasible solution  $S^{\natural} \in Q$  satisfying the equation (8) at random, therefore solve  $MIP(Q)$ . Set a random value  $r_i$  ( $0 \leq r_i \leq p_i$ ) for  $i \in S^{\natural} \cup S^{(t)}$ .

**Step 3:** If  $|S^{\natural}| = k$  holds, then take the  $k$  largest elements  $i \in S^{\natural} \cup S^{(t)}$  with respect to  $r_i$  to generate a feasible solution  $S' \in F$ . Otherwise, take the largest element  $i \in S^{(t)} \setminus S^{\natural}$  with respect to  $r_i$  to generate a feasible solution  $S' = S^{\natural} \cup \{i\} \in F$ .

**Step 4:** If  $S' \notin Q'$ , then set  $Q' \leftarrow Q' \cup \{S'\}$  and  $h \leftarrow h + 1$ .

**Step 5:** If  $h = \lambda$  holds, then the output  $Q'$  and exit. Otherwise, return to Step 2.

in the subsequent iterations. In order to do so, it is required to add a new feasible solution  $S' \in F$  to the set  $Q$  to fulfill the succeeding inequality:

$$z^{(t)} > f(S') + \sum_{i \in N \setminus S'} f(\{i\} | S') y_i^{(t)}. \quad (7)$$

By solving  $MIP(Q)$  we obtain at least one feasible solution  $S^{\natural} \in Q$  which satisfies the equation

$$z^{(t)} = f(S^{\natural}) + \sum_{i \in N \setminus S^{\natural}} f(\{i\} | S^{\natural}) y_i^{(t)}. \quad (8)$$

For all  $i \in N$  let  $q_i$  be the number of feasible solutions  $S \in Q$  including an element  $i$ . Using this quantity the occurrence rate  $p_i$  of element  $i$  is calculated as:

$$p_i = \frac{q_i}{\sum_{j \in N} q_j} \quad (9)$$

For the heuristic part of the algorithm, meaning selecting the added nodes, we use  $r_i$  that is generated by uniformly at random from 0 and  $p_i$ , i.e.,  $0 \leq r_i \leq p_i$ . We choose one solution at random from  $S^{\natural} \in Q$  in that case when multiple feasible solutions exist which are satisfying the equation (8).

## 4.2 ICG with reduced $k$ (ICG( $k - 1$ ))

As a first variation of the ICG, we modified Step 3 of Algorithm 2 to select not the  $k$  largest elements if  $S^{\natural} = k$  holds, but select the  $k - 1$  largest elements ( $i \in S^{\natural} \cup S^{(t)}$  with respect to  $r_i$ ). Considering the reduced problem's definition in (6), with this modification we get the  $k$ -th element's function value when we add it to the set.

According to our empirical investigations, to be shown in Section 5, ICG( $k - 1$ ) is more efficient in terms of average running time than ICG, thus in the following we use this version of the algorithm as a base for further improvements.

## 4.3 ICG using graph structure (GCG)

Our second variant uses the structure of the input graph instances to calculate the  $p_i$  value in (9) that is needed to generate  $r_i$  by random, see Step 2 in Algorithm 2. For this we distinguish the problems defined on fully connected graphs (that is the LOC problem in our case) and on non-fully connected graphs (those are the COV and INF problems we consider).

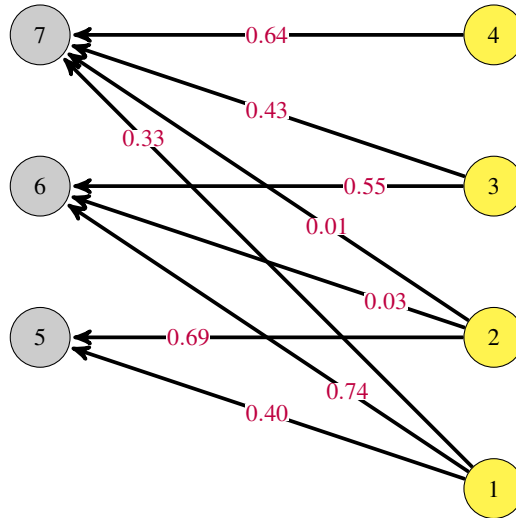
In the case when the problem is fully connected bipartite graph, for every vertex  $j \in M$  we calculate the median  $m_j$  of the outgoing edges' weights. Recall that  $M$  refers to the set of nodes with outgoing edges only. To calculate the value of  $p_i$ , we add up the weights of the incoming edges at node  $i \in N$  normalized with the degree of the targets node:

$$p_i = \begin{cases} \sum_{\substack{j:(j,i) \in E(G), \\ w_{ji} \geq m_j}} \frac{w_{ji}}{d_j} & \text{if } G \text{ is fully connected bipartite graph,} \\ \sum_{j:(j,i) \in E(G)} \frac{w_{ji}}{d_j} & \text{otherwise,} \end{cases} \quad (10)$$

where  $G$  is the input graph of the optimization problem,  $E(G)$  is the set of edges of  $G$ , the edges have  $w_{ji}$  weights and  $d_j$  is the degree of the node  $j \in M$ . This defines the value of  $p_i$  and based on this we set the value of  $r_i$  by uniformly at random such that  $0 \leq r_i \leq p_i$  holds.

As an illustrative example, see the graph on Fig. 1, where the labels of the nodes are indicated as black numbers. The results of equation (10) are reported in Table 1 containing the label of the node and the value of  $p_i$ .

Although, we choose  $k$  nodes from  $N$  we give a selection probability  $p_i$  by using nodes in  $M$ . The weight of outgoing edges of  $j \in M$  are divided by the degree of  $j$ , which is used to normalize the effect of the edges. We sum this normalized edges weights for each node  $i \in N$ , which expresses the average impact of selecting node  $i \in N$  relative to the other  $N$  nodes.



**Fig. 1** Example graph to calculate the  $p_i$  values

**Table 1** The calculated  $p_i$  values of the  $i \in N$  nodes for the graph in Figure 1

node	$p_i$ value
5	0.36
6	0.53
7	0.97

Let's revisit the example in Figure 1. There is only one edge from node 4 to node 7 so node 7 is important: node 4 can be served only if node 7 is selected. For this reason, we add to the selection probability value ( $p_i$ ) an edge weight divided by 1 (i.e., it remains itself), so that the probability of selection of node 7 becomes high. In contrast, node 1 is connected to three nodes, indicating that it can be reached from nodes 5, 6 and 7. These nodes are interchangeable for reaching node 1 and therefore less important. If we investigate the  $p_i$  value of the nodes for the graph in Table 1, we can see that those values are corresponding to the number and the weight of edges from nodes in set  $N$ . Accordingly, node 7 has the highest value because node 3 and 4 have one and two edges, respectively. Node 5 has the smallest  $p_i$  value because node 2 and 1 which are connected to it have edges to all vertices in  $N$ .

#### 4.4 ICG using enumeration (ECG)

The formal description of our third approach is shown in Algorithm 3. The first four steps, namely Step 1 - 4 are similar to those used in the previous variants. Step 5 selects a feasible solution  $S^{\natural} \in \mathcal{Q}$  uniformly at random which satisfies the equation (8). Then, in Step 6 the set  $\Sigma$  is initialized as the union of the sets  $S^{\natural}$  and  $S^{(t)}$ . The size of  $\Sigma$  is controlled by the parameter  $\kappa$ . In the subsequent steps the algorithm deals with the subsets of  $\Sigma$  (this is the enumeration part), where we need to balance between



**Algorithm 3** ECG ( $S^{(0)}, \lambda, \kappa$ )

**Input** The initial feasible solution  $S^{(0)}$ , the number of feasible solution  $\lambda$  and  $\kappa$  is the number of the elements of  $\Sigma$ .

**Output** The optimal solution  $S^*$ .

**Step 1:** Set  $Q \leftarrow S^{(0)}$ ,  $Q^+ \leftarrow \{S_{[0]}^{(0)}, \dots, S_{[k]}^{(0)}\}$ ,  $S^* \leftarrow S^{(0)}$  and  $t \leftarrow 1$ .

**Step 2:** Solve  $MIP(Q^+)$ . Let  $S^{(t)}$  and  $z^{(t)}$  be an optimal solution and the optimal value of  $MIP(Q^+)$ .

**Step 3:** If  $f(S^{(t)}) > f(S^*)$ , then set  $S^* \leftarrow S^{(t)}$ .

**Step 4:** If  $z^{(t)} = f(S^*)$  holds, then output the solution  $S^*$  and exit.

**Step 5:** Select a feasible solution  $S^{\natural} \in Q$  satisfying the equation (8) at random.

**Step 6:** Set  $\Sigma = S^{\natural} \cup S^{(t)}$ . If  $|\Sigma| > \kappa$  then let  $\Sigma$  be the first  $\kappa$  elements with the largest  $p_i$  values from  $S^{\natural} \cup S^{(t)}$ , where  $p_i$  is defined in (10).

**Step 7:** Let  $\mathcal{P}_{k-1}$  be the set of all the subsets of the power set of  $\Sigma$  which have at most  $k-1$  elements and assign the corresponding function values to these subsets.

**Step 8:** Keep at most  $\lambda$  elements in  $\mathcal{P}_{k-1}$ . So at this point  $|\mathcal{P}_{k-1}| \leq \lambda$ .

**Step 9:** Set  $Q \leftarrow Q \cup \{S^{(t)}\}$ ,  $Q^+ \leftarrow Q^+ \cup \{S^{(t)}\} \cup \mathcal{P}_{k-1}$  and  $t \leftarrow t+1$ .

**Step 10:** For each feasible solution  $S \in \mathcal{P}_{k-1}$ , if  $f(S) > f(S^*)$  holds, then set  $S^* \leftarrow S$ . Return to Step 2.

computational cost and the benefit of obtaining lower bounds of high quality. Thus, if  $|\Sigma| > \kappa$ , then we keep at most  $\kappa$  elements with the largest  $p_i$  values which is given by equation (10). Step 7 defines the set  $\mathcal{P}_{k-1} \subseteq 2^{\Sigma}$ . From all elements in  $\mathcal{P}_{k-1}$  we keep only those with cardinality at most  $k-1$  and calculate their function values. In Step 8, based on their function values we keep at most  $\lambda$  elements in  $\mathcal{P}_{k-1}$ , where  $\lambda$  is another control parameter of the algorithm.

The motivation behind Steps 5 - 8 is based on the following facts. Firstly, in contrast to ICG and GCG, we do not iterate any sub-algorithm inside the main algorithm, which could lead to less computational time. Secondly, in order to select the elements (i.e., nodes of the input graph) for  $\Sigma$  we use the  $p_i$  values, which are based on the degree properties of the input graph. Finally, the algorithm generates a good amount of promising feasible solutions, and, similarly to ICG, these solutions can help decreasing the upper bound of problem (2), see (7). This depends on the strategy to be used for keeping the elements in  $\mathcal{P}_{k-1}$  in Step 8. Note that in our experiments we used the strategy of keeping the elements with the smallest function values, as this turned out to be numerically efficient, see Section 5.

## 5 Numerical experiments

### 5.1 Computational environment

The implementation of all the investigated algorithms and models were done in AMPL [28]. For the numerical experiments the solver CPLEX 20.1.0.0 was used with the default options. The computer used had Intel Core CPU i5-6500 at 3.20GHz with 64G memory running Ubuntu Linux 22.04.

## 5.2 Test graphs

The authors of [1] made the graphs they used in their paper available online, thus for benchmarking the proposed methods we used those instances for the LOC and the COV problems. For the INF problems we generated new random graphs because the original graphs become rather easy to solve, all of the algorithms attain the optimal solutions. Our bipartite INF graphs were generated using Erdős-Rényi model with probability  $p = 0.3$  probability. Note that [26] also used  $p = 0.3$  in their experiments, whereas in [1] the parameter  $p = 0.1$  was used resulting sparser graphs.

Following the approach in [1] we had:

- $N = 20, 30, 40, 50, 60$  for LOC instances and  $N = 20, 40, 60, 80, 100$  for COV and INF instances;
- $M = N + 1$  and  $k = 5, 8$  for LOC, COV and INF.
- For LOC instances,  $g_{ij}$  is a random value taken from interval  $[0, 1]$ ;
- for COV instances, a sensor  $j \in N$  randomly covers an item  $i \in M$  with probability 0.15, and  $w_i$  is a random value taken from interval  $[0, 1]$ ; and
- for INF instances,  $p_j$  is a random value taken from the interval  $[0, 1]$ .
- We had  $\lambda = 10 \cdot k$ .
- Finally, for controlling the cardinality of set  $\Sigma$  in ECG we had  $\kappa = 12$  based on computational results of prior experiments.
- Note that all the random parameters were generated with uniform distribution.

Regarding the analysis of the parameters  $\lambda$  and  $\kappa$  we have created a Supplementary Information file which is available online [29].

## 5.3 Benchmarking results

The results are summarized in Tables 2 - 9. For each class 5 problem instances were tested, indicated in the last digit of the name of the instance. For every instance all the algorithms were run 5 times using different random seeds for the heuristics choices. The time limit was set to two hours (7 200 seconds). The solution of the greedy algorithm was the the input initial feasible solution  $S^{(0)}$  for every algorithm.

Tables 2 - 7 report the average running times (in seconds) and the mean values of the number of iterations (in brackets). The cases when an algorithm was running out of the time limit is indicated by the ⌚ symbol. Note that in these cases the number of iterations is not reported. Those cases when an algorithm was able to solve the problem for less than 5 different runs are indicated with a star (★) next to the reported running time. Only those instances are reported for which we had at least one algorithm solving the problem at least one of the cases.




Tables 8 and 9 report the mean relative gap<sup>2</sup> and that how many times the algorithm was *not* able to solve the instance out of the 5 independent runs (in brackets). Note that only those instances are reported for which we had at least one algorithm which was running out of time at least once.

In the following we give detailed analysis of the reported results. The performance metrics of the proposed algorithms are discussed for the benchmark problems.

---

<sup>2</sup> $(z_{UB} - z_{LB})/z_{LB} \times 100$ , where  $z_{UB}$  and  $z_{LB}$  are the upper and lower bounds reported by the algorithms, respectively

**Table 2** Results for LOC  $k = 5$ . Mean running time in seconds (avg. nr. of iterations in brackets)

graphs	ICG		ICG( $k - 1$ )		GCG		ECG	
L.20.5.1	0.96	(9.4)	0.75	(8.6)	0.50	(5.6)	<b>0.49</b>	(5.0)
L.20.5.2	0.34	(5.8)	0.29	(5.0)	0.27	(3.8)	<b>0.21</b>	(3.0)
L.20.5.3	0.34	(5.8)	0.23	(4.4)	0.25	(4.0)	<b>0.15</b>	(2.0)
L.20.5.4	0.16	(3.4)	0.19	(4.0)	0.18	(3.0)	<b>0.12</b>	(2.0)
L.20.5.5	0.36	(5.6)	0.33	(5.0)	0.29	(4.2)	<b>0.24</b>	(3.0)
L.30.5.1	8.24	(17.8)	3.75	(12.8)	3.42	(10.4)	<b>2.31</b>	(8.0)
L.30.5.2	5.33	(16.6)	1.89	(10.8)	1.76	(8.6)	<b>1.15</b>	(6.0)
L.30.5.3	4.16	(14.4)	1.85	(10.4)	1.78	(8.2)	<b>1.35</b>	(6.0)
L.30.5.4	6.94	(18.4)	3.26	(13.2)	2.24	(9.6)	<b>2.59</b>	(10.0)
L.30.5.5	2.34	(12.2)	1.39	(10.0)	<b>1.21</b>	(7.0)	1.48	(7.0)
L.40.5.1	50.97	(27.6)	22.63	(22.0)	15.39	(16.2)	<b>13.77</b>	(15.0)
L.40.5.2	267.48	(49.0)	54.18	(30.0)	43.23	(25.6)	<b>26.36</b>	(21.0)
L.40.5.3	531.63	(54.4)	79.39	(32.4)	66.00	(27.4)	<b>30.49</b>	(19.0)
L.40.5.4	95.76	(38.2)	28.15	(24.8)	19.96	(19.0)	<b>12.44</b>	(15.0)
L.40.5.5	469.13	(58.4)	57.55	(31.2)	50.59	(26.6)	<b>39.25</b>	(24.0)
L.50.5.1	836.00	(57.2)	180.02	(35.2)	109.65	(28.2)	<b>107.15</b>	(25.0)
L.50.5.2	2 502.17	(81.8)	423.52	(47.0)	300.89	(37.0)	<b>185.98</b>	(29.0)
L.50.5.3	4 206.50	(89.8)	511.73	(46.4)	413.37	(38.0)	<b>396.25</b>	(37.0)
L.50.5.4	183.40	(35.8)	51.46	(25.2)	31.78	(18.6)	<b>26.12</b>	(16.0)
L.50.5.5	6 621.33	(22.4)	1079.33	(64.4)	776.84	(52.0)	<b>490.68</b>	(42.4)
L.60.5.1	 (–)	6 016.81	(84.6)	4 988.19	(96.6)	<b>3 217.13</b>	(75.0)	
L.60.5.2	 (–)	1 643.06	(69.4)	1 351.41	(60.8)	<b>1 030.39</b>	(52.0)	
L.60.5.3	485.37	(40.8)	184.59	(30.4)	<b>128.39</b>	(24.0)	143.99	(27.0)
L.60.5.4	 (–)	4 394.68	(92.4)	3 075.40	(75.4)	<b>1 968.20</b>	(62.0)	
L.60.5.5	170.65	(30.4)	66.25	(22)	<b>49.76</b>	(16.6)	51.67	(19.0)

For the different  $k$  values we compare our methods with ICG. The result of the fastest algorithm is indicated with boldface.

**LOC,  $k = 5$ .** The results are reported in Table 2. The ICG( $k - 1$ ) algorithm reduced the running time for almost all the cases (except once). To be precise, it was 3.46 times faster on average than ICG and it reduced the number of iterations to 82.62%. Furthermore, GCG was 4.49 times faster on average and reduced the number of iterations to 64.77%, while ECG was 6.10 times faster than ICG and the number of iterations were almost halved (53.71%). There are three cases where GCG was faster than ECG.

The relative gap values for those three graphs for which the ICG did not get the results are reported in Table 8. We can see that the gaps were below 1%.

**LOC,  $k = 8$ .** The results are reported in Table 3. Generally, the ICG( $k - 1$ ) obtained the result 1.79 times faster while the number of iteration was 94.76% of the ICG's iterations. GCG was 2.58 times faster and the number of iterations was reduced to 76.83%. Note that while ECG reduced the iterations to a similar extent (76.21%), it

**Table 3** Results for LOC  $k = 8$ . Mean running time in seconds (avg. nr. of iterations in brackets)

graphs	ICG		ICG( $k - 1$ )		GCG		ECG	
L.20.8.1	0.34	(3.6)	0.31	(4.0)	<b>0.26</b>	(3.2)	0.78	(2.0)
L.20.8.2	0.27	(3.2)	0.30	(4.0)	<b>0.27</b>	(3.0)	1.75	(4.0)
L.20.8.3	0.20	(2.2)	0.19	(3.0)	<b>0.15</b>	(2.0)	0.81	(2.0)
L.20.8.4	<b>0.24</b>	(3.0)	0.25	(4.0)	0.25	(3.0)	1.43	(3.0)
L.20.8.5	0.23	(3.0)	<b>0.20</b>	(3.0)	0.26	(3.0)	0.97	(3.0)
L.30.8.1	35.93	(25.6)	14.65	(17.0)	<b>10.04</b>	(13.8)	17.15	(12.0)
L.30.8.2	2.30	(7.0)	1.65	(6.8)	<b>1.44</b>	(5.4)	6.69	(6.0)
L.30.8.3	2.62	(7.8)	2.05	(7.8)	<b>1.81</b>	(6.4)	6.65	(7.0)
L.30.8.4	8.56	(12.6)	7.08	(12.4)	<b>5.18</b>	(10.2)	14.48	(10.0)
L.30.8.5	1.45	(6.2)	1.45	(6.6)	<b>1.15</b>	(5.2)	5.24	(5.0)
L.40.8.1	1 502.32	(48.0)	563.23	(34.2)	<b>323.94</b>	(29.8)	568.86	(35.4)
L.40.8.2	3 831.26	(71.8)	1 108.92	(46.4)	598.51	(35.8)	<b>429.17</b>	(35.0)
L.40.8.3	76.76	(18.4)	35.70	(15.8)	<b>19.96</b>	(13.0)	38.46	(13.0)
L.40.8.4	229.16	(30.0)	78.09	(23.0)	<b>47.64</b>	(19.2)	76.09	(20.0)
L.40.8.5	147.42	(28.2)	56.17	(20.8)	<b>47.54</b>	(18.4)	55.27	(14.0)
L.50.8.1	⌚	(-)	⌚	(-)	6 357.27	(47.2)	<b>6 186.57</b>	(62.0)
L.50.8.2	3 483.92	(50.0)	1 359.93	(37.6)	1 010.56	(33.8)	<b>946.79</b>	(31.0)
L.50.8.3	556.33	(24.2)	360.34	(21.6)	218.02	(18.2)	<b>149.28</b>	(17.0)
L.50.8.5	⌚	(-)	⌚	(-)	<b>6 936.66*</b>	(64.5)	⌚	(-)

was faster only by 1.86 times compared to ICG. This confirms the fact that the technical time for generating the set of sets in the ECG increases when  $k = 8$  (compared to  $k = 5$ ), since GCG is faster with almost the same number of iterations.

For those instances which were not solved by any of the tested algorithms we can compare the relative gap values, as reported in Table 8. Note that none of the algorithms were able to solve the  $N = 60$  problems. Most of the cases all of our algorithms achieved a smaller gap than ICG. Overall, the relative gaps remained below 1% for all reported problems.

**COV**,  $k = 5$ . The results are reported in Table 4. All of our proposed algorithms were able to solve the instances within the time limit, whereas ICG was running out of time for 7 cases, mostly for the largest dimensional problems (for these problems the relative gaps are reported in Table 8). For the successful instances, GCG was able to speed-up the running time the most, it was 5.10 times faster, while the iterations were decreased to 48.39% compared to ICG. Next was ICG( $k - 1$ ) with 3.79 times speed-up and 70.82% reduction in the number of iterations. ECG was 3.23 times faster and solved the problems in less than half (49.46%) iterations.

**COV**,  $k = 8$ . The results are reported in Table 5. In this scenario GCG is the only one which is overall faster than ICG. Namely, GCG was 1.06 faster and reduced the number of iterations to 93.56%. ICG( $k - 1$ ) was 0.87 slower whereas ECG was 0.49 slower than ICG. Even the number of iterations increased. Note that this is the only group of problems where the baseline ICG algorithm showed better performance metrics in some of the cases.

**Table 4** Results for COV  $k = 5$ . Mean running time in seconds (avg. nr. of iterations in brackets)

graphs	ICG		ICG( $k - 1$ )		GCG		ECG	
C.20.5.1	0.11	(3.0)	0.11	(3.0)	<b>0.08</b>	(2.0)	0.10	(2.0)
C.20.5.2	0.10	(3.0)	0.10	(3.0)	<b>0.07</b>	(2.0)	0.11	(2.0)
C.20.5.3	0.15	(4.4)	0.09	(3.0)	<b>0.09</b>	(2.0)	<b>0.09</b>	(2.0)
C.20.5.4	0.14	(4.0)	0.10	(3.0)	<b>0.08</b>	(2.0)	<b>0.08</b>	(2.0)
C.20.5.5	0.10	(2.4)	0.10	(3.0)	<b>0.08</b>	(2.0)	0.10	(2.0)
C.40.5.1	1.49	(8.4)	0.77	(6.0)	<b>0.68</b>	(4.4)	1.33	(5.0)
C.40.5.2	2.04	(9.0)	<b>0.95</b>	(7.2)	1.02	(5.2)	1.32	(5.0)
C.40.5.3	1.48	(9.0)	0.73	(6.0)	<b>0.52</b>	(3.8)	0.93	(4.0)
C.40.5.4	4.07	(13.0)	<b>1.16</b>	(7.4)	1.23	(6.0)	1.29	(5.0)
C.40.5.5	3.11	(11.8)	<b>0.79</b>	(6.4)	0.81	(4.6)	1.01	(4.0)
C.60.5.1	36.29	(20.6)	9.39	(13.0)	<b>8.11</b>	(9.2)	11.02	(9.0)
C.60.5.1	29.78	(18.6)	11.54	(14.2)	<b>8.23</b>	(9.4)	8.79	(8.0)
C.60.5.1	22.26	(16.2)	8.37	(11.4)	<b>4.19</b>	(6.8)	8.43	(8.0)
C.60.5.1	34.15	(20.6)	8.43	(12.2)	<b>6.21</b>	(8.6)	10.73	(9.0)
C.60.5.1	95.97	(29.0)	18.70	(16.8)	<b>11.08</b>	(10.6)	24.61	(14)
C.80.5.1	5 296.05	(85.6)	339.04	(34.8)	<b>273.19</b>	(29.0)	398.06	(29.0)
C.80.5.2	☹	(-)	666.73	(48.8)	<b>489.90</b>	(39.4)	941.31	(44.0)
C.80.5.3	168.03	(26.4)	26.40	(15.4)	<b>19.08</b>	(10.4)	43.37	(12.0)
C.80.5.4	☹	(-)	718.39	(46.2)	<b>523.22</b>	(35.4)	780.66	(38.0)
C.80.5.5	534.24	(38.6)	63.66	(19.8)	<b>37.34</b>	(12.6)	69.62	(15.0)
C.100.5.1	☹	(-)	676.98	(41.2)	<b>377.66</b>	(28.2)	756.57	(35.0)
C.100.5.2	☹	(-)	904.86	(44.0)	<b>452.01</b>	(28.2)	1 018.30	(35.4)
C.100.5.3	☹	(-)	498.38	(36.8)	<b>405.62</b>	(28.6)	555.21	(29.0)
C.100.5.4	☹	(-)	1 331.61	(49.0)	<b>973.07</b>	(37.0)	2 906.53	(49.0)
C.100.5.5	☹	(-)	1 446.34	(52.8)	<b>997.95</b>	(39.0)	1 781.81	(43.0)

The same phenomenon can be seen when examining the relative gap values, see Table 8. Compared to ICG, the ICG( $k - 1$ ) and ECG resulted in larger average gaps, only GCG ended up with smaller gaps on average.

**INF**,  $k = 5$ . The results are shown in Table 6. The ICG algorithm could not solve the cases where  $N = 60, 80, 100$  (except one of them). Notably, the most efficient algorithm on these problems were ECG in terms of overall running time and success rate, as it was able to solve all the instances. Compared to the baseline (where ICG was able to finish within the time limit) the average running time of ECG was 296.24 times faster, and the iteration number decreased to 24.09%. The other two variants, ICG( $k - 1$ ) and GCG also had nice results: ICG( $k - 1$ ) was 14.94 times faster and the number of iteration decreased to 39.23%, while the speed up of GCG was 20.26 with iteration's ratio of 27.30%.

Investigating the gap values, as reported in Table 9, we can see that the results of ICG show the biggest gaps, for the larger dimensional instances it even goes above 10%. The ICG( $k - 1$ ) variant resulted much smaller relative gaps, and GCG also had tighter final results than ICG.

**Table 5** Results for COV  $k = 8$ . Mean running time in seconds (avg. nr. of iterations in brackets)

graphs	ICG		ICG( $k - 1$ )		GCG		ECG	
C.20.8.1	0.14	(2.2)	<b>0.08</b>	(2.0)	0.11	(2.0)	0.62	(2.0)
C.20.8.2	<b>0.05</b>	(1.0)	<b>0.05</b>	(1.0)	<b>0.05</b>	(1.0)	<b>0.05</b>	(1.0)
C.20.8.3	<b>0.05</b>	(1.0)	<b>0.05</b>	(1.0)	<b>0.05</b>	(1.0)	<b>0.05</b>	(1.0)
C.20.8.4	0.10	(2.0)	<b>0.07</b>	(2.0)	0.10	(2.0)	0.21	(2.0)
C.20.8.5	0.09	(2.0)	<b>0.08</b>	(2.0)	0.13	(2.0)	0.20	(2.0)
C.40.8.1	1.23	(4.6)	1.24	(4.8)	<b>0.80</b>	(3.2)	9.20	(4.8)
C.40.8.2	0.42	(3.0)	0.39	(3.2)	<b>0.35</b>	(2.4)	5.15	(3.4)
C.40.8.3	<b>1.24</b>	(4.4)	8.76	(6.8)	4.23	(4.2)	14.97	(5.6)
C.40.8.4	0.88	(4.0)	1.12	(4.6)	<b>0.81</b>	(3.2)	7.41	(4.2)
C.40.8.5	<b>0.06</b>	(1.0)	<b>0.06</b>	(1.0)	<b>0.06</b>	(1.0)	0.07	(1.0)
C.60.8.1	<b>21.25</b>	(7.0)	44.72	(8.4)	32.16	(6.6)	136.43	(10.8)
C.60.8.2	0.09	(1.0)	<b>0.08</b>	(1.0)	<b>0.08</b>	(1.0)	0.09	(1.0)
C.60.8.3	<b>0.08</b>	(1.0)	<b>0.08</b>	(1.0)	<b>0.08</b>	(1.0)	0.09	(1.0)
C.60.8.4	<b>7.63</b>	(6.6)	12.43	(7.2)	10.94	(5.4)	52.95	(8.0)
C.60.8.5	11.78	(6.8)	11.16	(7.6)	<b>7.09</b>	(5.6)	51.11	(8.4)
C.80.8.1	284.70	(10.4)	621.12	(14.2)	<b>276.53</b>	(9.6)	705.20	(13.4)
C.80.8.2	470.94	(14.0)	579.62	(16.2)	<b>380.11</b>	(13.8)	721.66	(17.4)
C.80.8.3	<b>199.77</b>	(11.4)	362.43	(14.4)	221.57	(11.4)	546.51	(15.2)
C.80.8.4	1 529.46	(18.4)	2 067.69	(22.2)	<b>1 010.48</b>	(17.8)	2 382.21	(22.6)
C.80.8.5	<b>188.68</b>	(11.4)	361.91	(15.0)	194.43	(11.0)	473.23	(14.6)
C.100.8.1	2 797.19	(19.0)	4 166.53	(22.8)	<b>2 054.98</b>	(17.8)	5 078.67	(24.6)
C.100.8.2	5 483.76	(22.4)	⊗	(−)	<b>4 686.29</b>	(23.8)	6 596.13*	(27.0)
C.100.8.3	⊗	(−)	⊗	(−)	⊗	(−)	⊗	(−)
C.100.8.4	⊗	(−)	⊗	(−)	⊗	(−)	⊗	(−)
C.100.8.5	⊗	(−)	⊗	(−)	<b>6 113.83*</b>	(−)	⊗	(−)

**INF**,  $k = 8$ . These are the problem instances which were the most difficult ones for the tested algorithms. As it can be seen in Table 7, the algorithms could solve only the smallest graphs ( $N = 20$ ) and some of the graphs with  $N = 40$ . ECG is the only one which could solve all the problems where  $N \leq 40$ . However, examining the relative gap values in Table 9 we can see that the results of the ECG were not the smallest ones and even ICG obtained tighter gaps in some instances. Overall, GCG achieved the best gap values.

## 6 Conclusions

We proposed three different algorithm variants for the non-decreasing submodular function maximization problem based on a MIP formulation using constraint generation approach. The work was inspired by a recent paper by [1], named as ICG as it is an improved version of the standard constraint generation method. We presented an algorithm, ICG( $k - 1$ ), which uses sets of cardinality  $k - 1$ , where we calculate the case when adding the  $k$ -th element to the set with (8). Another idea was to exploit the structural properties of the input graph to select nodes, we called the algorithm as

**Table 6** Results for INF  $k = 5$ . Mean running time in seconds (avg. nr. of iterations in brackets)

graphs	ICG		ICG( $k - 1$ )		GCG		ECG	
I.20.5.1	0.35	(7.2)	0.14	(4.0)	<b>0.06</b>	(2.0)	0.16	(4.0)
I.20.5.2	2.71	(16.0)	0.65	(7.8)	0.45	(5.4)	<b>0.23</b>	(5.0)
I.20.5.3	0.83	(9.6)	0.27	(5.0)	<b>0.14</b>	(3.0)	0.17	(4.0)
I.20.5.4	1.12	(13.4)	0.21	(5.0)	0.17	(3.2)	<b>0.10</b>	(3.0)
I.20.5.5	0.63	(9.8)	0.12	(3.4)	0.10	(2.6)	<b>0.09</b>	(3.0)
I.40.5.1	⌚	(-)	211.64	(57.4)	155.88	(45.4)	<b>5.91</b>	(17.0)
I.40.5.2	1 230.13	(101.0)	68.32	(35.4)	51.35	(29.2)	<b>5.91</b>	(16.0)
I.40.5.3	2 517.61	(114.6)	77.88	(35.4)	62.79	(29.4)	<b>2.57</b>	(11.0)
I.40.5.4	2 036.73	(131.2)	62.04	(37.2)	42.11	(27.8)	<b>3.51</b>	(13.0)
I.40.5.5	2 420.70	(121.6)	84.12	(38.8)	71.94	(33.0)	<b>3.84</b>	(13.0)
I.60.5.1	⌚	(-)	1 872.07	(87.4)	1 447.89	(71.6)	<b>14.06</b>	(19.0)
I.60.5.2	⌚	(-)	⌚	(-)	⌚	(-)	<b>99.67</b>	(41.0)
I.60.5.3	⌚	(-)	⌚	(-)	7 122.45*	(138.0)	<b>46.91</b>	(31.0)
I.60.5.4	4 115.66	(112.0)	239.77	(42.2)	158.94	(30.2)	<b>7.81</b>	(15.0)
I.60.5.5	⌚	(-)	5 101.47	(132.2)	4 346.57	(117.2)	<b>38.17</b>	(30.0)
I.80.5.1	⌚	(-)	⌚	(-)	⌚	(-)	<b>63.90</b>	(29.0)
I.80.5.2	⌚	(-)	⌚	(-)	⌚	(-)	<b>631.07</b>	(73.0)
I.80.5.3	⌚	(-)	⌚	(-)	⌚	(-)	<b>137.34</b>	(42.0)
I.80.5.4	⌚	(-)	⌚	(-)	⌚	(-)	<b>543.79</b>	(70.0)
I.80.5.5	⌚	(-)	⌚	(-)	⌚	(-)	<b>137.87</b>	(43.0)
I.100.5.1	⌚	(-)	⌚	(-)	⌚	(-)	<b>693.71</b>	(66.0)
I.100.5.2	⌚	(-)	⌚	(-)	⌚	(-)	<b>1 674.73</b>	(99.0)
I.100.5.3	⌚	(-)	⌚	(-)	⌚	(-)	<b>655.78</b>	(65.0)
I.100.5.4	⌚	(-)	⌚	(-)	⌚	(-)	<b>1 715.59</b>	(98.0)
I.100.5.5	⌚	(-)	⌚	(-)	⌚	(-)	<b>636.56</b>	(64.0)

**Table 7** Results for INF  $k = 8$ . Mean running time in seconds (avg. nr. of iterations in brackets)

graphs	ICG		ICG( $k - 1$ )		GCG		ECG	
I.20.8.1	4.44	(12.6)	1.97	(8.6)	2.10	(8.0)	<b>1.87</b>	(8.0)
I.20.8.2	1.58	(8.0)	0.95	(6.4)	<b>0.64</b>	(4.6)	2.80	(11.0)
I.20.8.3	1.28	(7.4)	<b>0.71</b>	(5.6)	0.97	(6.0)	1.34	(7.0)
I.20.8.4	2.61	(10.6)	1.27	(7.6)	1.01	(5.8)	<b>0.93</b>	(6.0)
I.20.8.5	13.37	(21.6)	3.91	(11.8)	<b>2.67</b>	(9.0)	3.28	(11.0)
I.40.8.1	⌚	(-)	⌚	(-)	⌚	(-)	<b>4 111.21</b>	(60.0)
I.40.8.2	⌚	(-)	⌚	(-)	6 658.51	(78.0)	<b>2 538.11</b>	(55.0)
I.40.8.3	⌚	(-)	⌚	(-)	⌚	(-)	<b>6 242.53</b>	(78.0)
I.40.8.4	⌚	(-)	⌚	(-)	⌚	(-)	<b>2 256.41</b>	(56.0)
I.40.8.5	⌚	(-)	6 871.58	(81.0)	5 548.98	(73.0)	<b>1 369.95</b>	(50.0)

**Table 8** Relative gaps for LOC and COV problems (number of unsuccessful runs in brackets)

graph	ICG	ICG( $k-1$ )	GCG	ECG
L.60.5.1	0.58 (5)	0.00 (0)	0.00 (0)	0.00 (0)
L.60.5.2	0.12 (5)	0.00 (0)	0.00 (0)	0.00 (0)
L.60.5.4	0.46 (5)	0.00 (0)	0.00 (0)	0.00 (0)
L.50.8.1	0.27 (5)	0.10 (5)	0.00 (0)	0.00 (0)
L.50.8.4	0.26 (5)	0.22 (5)	0.11 (5)	0.20 (5)
L.50.8.5	0.22 (5)	0.02 (5)	0.02 (2)	0.09 (5)
L.60.8.1	0.39 (5)	0.33 (5)	0.17 (5)	0.29 (5)
L.60.8.2	0.33 (5)	0.38 (5)	0.16 (5)	0.18 (5)
L.60.8.3	0.26 (5)	0.25 (5)	0.21 (5)	0.34 (5)
L.60.8.4	0.77 (5)	0.93 (5)	0.50 (5)	0.70 (5)
L.60.8.5	0.77 (5)	0.71 (5)	0.62 (5)	0.57 (5)
C.80.5.2	0.76 (5)	0.00 (0)	0.00 (0)	0.00 (0)
C.80.5.4	0.55 (5)	0.00 (0)	0.00 (0)	0.00 (0)
C.100.5.1	0.43 (5)	0.00 (0)	0.00 (0)	0.00 (0)
C.100.5.2	0.42 (5)	0.00 (0)	0.00 (0)	0.00 (0)
C.100.5.3	0.28 (5)	0.00 (0)	0.00 (0)	0.00 (0)
C.100.5.4	1.33 (5)	0.00 (0)	0.00 (0)	0.00 (0)
C.100.5.5	1.69 (5)	0.00 (0)	0.00 (0)	0.00 (0)
C.100.8.2	0.00 (0)	0.13 (5)	0.00 (0)	0.15 (3)
C.100.8.3	1.49 (5)	1.79 (5)	1.41 (5)	1.29 (5)
C.100.8.4	1.84 (5)	2.94 (5)	1.96 (5)	2.55 (5)
C.100.8.5	0.54 (5)	0.93 (5)	0.45 (4)	0.95 (5)

GCG. Finally, we proposed ECG which generates the subsets of sets instead of use an iterative sub-algorithm.

According to our benchmarking results, we cannot declare a winner among the algorithms and this is not surprising as the investigated problem is NP-hard. However, for every instances there exists at least one of our algorithms which is computationally more efficient than the ICG algorithm.

**Acknowledgements.** We thank our reviewers for their critical comments and valuable suggestions that highlighted important details and helped us in improving our paper.

The research leading to these results has received funding from the national project TKP2021-NVA-09. Project no. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme. The work was also supported by the grant SNN-135643 of the National Research, Development and Innovation Office, Hungary.

**Data availability.** The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.



**Table 9** Relative gaps for INF problems (number of unsuccessful runs in brackets)

graph	ICG	ICG( $k-1$ )	GCG	ECG
I.40.5.1	0.29 (5)	0.00 (0)	0.00 (0)	0.00 (0)
I.60.5.1	2.94 (5)	0.00 (0)	0.00 (0)	0.00 (0)
I.60.5.2	6.39 (5)	1.35 (5)	1.02 (5)	0.00 (0)
I.60.5.3	5.92 (5)	1.17 (5)	0.82 (4)	0.00 (0)
I.60.5.5	4.71 (5)	0.00 (0)	0.00 (0)	0.00 (0)
I.80.5.1	5.38 (5)	1.27 (5)	0.71 (5)	0.00 (0)
I.80.5.2	9.91 (5)	5.39 (5)	4.83 (5)	0.00 (0)
I.80.5.3	8.26 (5)	3.74 (5)	3.16 (5)	0.00 (0)
I.80.5.4	10.89 (5)	6.19 (5)	5.46 (5)	0.00 (0)
I.80.5.5	7.83 (5)	3.26 (5)	2.67 (5)	0.00 (0)
I.100.5.1	11.48 (5)	1.42 (5)	6.16 (5)	0.00 (0)
I.100.5.2	11.44 (5)	1.58 (5)	6.72 (5)	0.00 (0)
I.100.5.3	10.79 (5)	1.60 (5)	7.43 (5)	0.00 (0)
I.100.5.4	13.31 (5)	1.79 (5)	8.43 (5)	0.00 (0)
I.100.5.5	10.98 (5)	1.36 (5)	6.00 (5)	0.00 (0)
I.40.8.1	2.33 (5)	1.07 (5)	0.89 (5)	0.00 (0)
I.40.8.2	1.10 (5)	0.19 (5)	0.07 (3)	0.00 (0)
I.40.8.3	1.83 (5)	1.04 (5)	0.80 (5)	0.00 (0)
I.40.8.4	1.03 (5)	1.02 (5)	0.00 (0)	0.00 (0)
I.40.8.5	0.91 (5)	0.62 (3)	0.00 (0)	0.00 (0)
I.60.8.1	7.36 (5)	6.79 (5)	6.23 (5)	6.50 (5)
I.60.8.2	7.98 (5)	7.68 (5)	7.31 (5)	8.95 (5)
I.60.8.3	8.57 (5)	8.16 (5)	7.50 (5)	8.43 (5)
I.60.8.4	3.35 (5)	2.96 (5)	2.66 (5)	3.51 (5)
I.60.8.5	6.21 (5)	5.65 (5)	5.36 (5)	6.59 (5)
I.80.8.1	9.12 (5)	8.57 (5)	8.15 (5)	9.23 (5)
I.80.8.2	10.21 (5)	9.91 (5)	9.35 (5)	11.06 (5)
I.80.8.3	9.33 (5)	9.19 (5)	8.72 (5)	10.22 (5)
I.80.8.4	8.58 (5)	8.23 (5)	7.79 (5)	9.56 (5)
I.80.8.5	10.24 (5)	10.18 (5)	9.49 (5)	10.45 (5)
I.100.8.1	14.12 (5)	14.11 (5)	13.40 (5)	15.29 (5)
I.100.8.2	12.59 (5)	13.01 (5)	12.56 (5)	14.49 (5)
I.100.8.3	13.87 (5)	13.78 (5)	13.17 (5)	15.78 (5)
I.100.8.4	13.06 (5)	13.01 (5)	12.51 (5)	14.65 (5)
I.100.8.5	13.50 (5)	13.41 (5)	13.08 (5)	15.24 (5)

**Conflict of interest.** The authors declare no conflict of interest.

## References

- [1] Uematsu, N., Umetani, S., Kawahara, Y.: An efficient branch-and-cut algorithm for submodular function maximization. *Journal of the Operations Research Society of Japan* **63**(2), 41–59 (2020)

- [2] Goldengorin, B.: Maximization of submodular functions: Theory and enumeration algorithms. *European Journal of Operational Research* **198**(1), 102–112 (2009)
- [3] Benati, S.: An Improved Branch & Bound Method for the Uncapacitated Competitive Location Problem. *Annals of Operations Research* **122**(1), 43–58 (2003)
- [4] Goldengorin, B., Ghosh, D.: A multilevel search algorithm for the maximization of submodular functions applied to the quadratic cost partition problem. *Journal of Global Optimization* **32**, 65–82 (2005)
- [5] Golovin, D., Krause, A.: Adaptive submodularity: A new approach to active learning and stochastic optimization. In: COLT, pp. 333–345 (2010)
- [6] Krause, A., Guestrin, C.: Submodularity and its applications in optimized information gathering. *ACM Transactions on Intelligent Systems and Technology (TIST)* **2**(4), 1–20 (2011)
- [7] Shen, J., Liang, Z., Liu, J., Sun, H., Shao, L., Tao, D.: Multiobject tracking by submodular optimization. *IEEE Transactions on Cybernetics* **49**(6), 1990–2001 (2019)
- [8] Das, A., Kempe, D.: Submodular meets spectral: Greedy algorithms for subset selection, sparse approximation and dictionary selection. In: Proceedings of the 28th International Conference on Machine Learning. ICML’11, pp. 1057–1064 (2011)
- [9] Kempe, D., Kleinberg, J., Tardos, E.: Maximizing the spread of influence through a social network. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 137–146 (2003)
- [10] Krause, A., Golovin, D.: Submodular function maximization. *Tractability* **3**, 71–104 (2014)
- [11] Edmonds, J.: Matroids, submodular functions, and certain polyhedra. In: Guy, R.K., Hanani, H., Sauer, N., Schönheim, J. (eds.) *Combinatorial Structures and Their Applications*, pp. 69–87. Gordon and Beach, New York (1970)
- [12] Frank, A.: Matroids and submodular functions. In: Dell’Amico, M., Maffioli, F., Martello, S. (eds.) *Annotated Bibliographies in Combinatorial Optimization*, pp. 65–80 (1997)
- [13] Chen, Y., Zhao, L., Zhang, Y., Huang, S., Dissanayake, G.: Anchor selection for slam based on graph topology and submodular optimization. *IEEE Transactions on Robotics* **38**(1), 329–350 (2022)

- [14] Zhang, H., Vorobeychik, Y.: Submodular optimization with routing constraints. *Proceedings of the AAAI Conference on Artificial Intelligence* **30**(1) (2016)
- [15] Sakaue, S., Nishino, M., Yasuda, N.: Submodular function maximization over graphs via zero-suppressed binary decision diagrams. *Proceedings of the AAAI Conference on Artificial Intelligence* **32**(1) (2018) <https://doi.org/10.1609/aaai.v32i1.11520>
- [16] Iwata, S., Fleischer, L., Fujishige, S.: A combinatorial strongly polynomial algorithm for minimizing submodular functions. *Journal of the ACM* **48**(4), 761–777 (2001)
- [17] Schrijver, A.: A combinatorial algorithm minimizing submodular functions in strongly polynomial time. *Journal of Combinatorial Theory, Series B* **80**, 346–355 (2001)
- [18] Feige, U., Mirrokni, V.S., Vondrák, J.: Maximizing non-monotone submodular functions. *SIAM Journal on Computing* **40**(4), 1133–1153 (2011)
- [19] Krause, A., Guestrin, C.: Optimal nonmyopic value of information in graphical models: Efficient algorithms and theoretical limits. In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pp. 1339–1345 (2005)
- [20] Nemhauser, G., Wolsey, L., Fisher, M.: An analysis of the approximations for maximizing submodular set functions - I. *Mathematical Programming* **14**, 265–294 (1978)
- [21] Nemhauser, G.L., Wolsey, L.A.: Maximizing submodular set functions: formulations and analysis of algorithms. *Studies on Graphs and Discrete Programming*, 279–301 (1981)
- [22] Chen, W., Chen, Y., Weinberger, K.: Filtered search for submodular maximization with controllable approximation bounds. In: *Artificial Intelligence and Statistics*, pp. 156–164 (2015)
- [23] Salvagnin, D.: Some experiments with submodular function maximization via integer programming. In: Rousseau, L.-M., Stergiou, K. (eds.) *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pp. 488–501. Springer, Cham (2019)
- [24] Ohsaka, N., Matsuoka, T.: Approximation algorithm for submodular maximization under submodular cover. In: *Uncertainty in Artificial Intelligence*, pp. 792–801 (2021)
- [25] Lu, C., Yang, W., Gao, S.: A new greedy strategy for maximizing monotone submodular function under a cardinality constraint. *Journal of Global Optimization* **83**(2), 235–247 (2022)

- [26] Sakaue, S., Ishihata, M.: Accelerated best-first search with upper-bound computation for submodular function maximization. Proceedings of the AAAI Conference on Artificial Intelligence **32**(1) (2018)
- [27] Kawahara, Y., Nagano, K., Tsuda, K., Bilmes, J.A.: Submodularity cuts and applications. In: Advances in Neural Information Processing Systems, vol. 22, pp. 1–9 (2009)
- [28] Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL. A Modeling Language for Mathematical Programming. Thomson, Duxbury (2003)
- [29] Csókás, E., T., V.: Constraint generation approaches for submodular function maximization leveraging graph properties - Supplementary Information. [https://www.inf.u-szeged.hu/~tvinko/CsV-submodular\\_max-supplement.pdf](https://www.inf.u-szeged.hu/~tvinko/CsV-submodular_max-supplement.pdf)