On the Initial Set of Constraints for Graph-Based Submodular Function Maximization*

Eszter Csókás[ab] and Tamás Vinkó[acd]

Abstract

A crucial problem in combinatorial optimization is the submodular function maximization (SFM), and in many cases it involves graphs on which the maximization is specified. The problem is well-studied and hence there are several proposed algorithms in the literature. The greedy strategy quickly finds a feasible solution that guarantees an approximation of $(1 - 1/e)$. However, there are many applications that expect an optimal result within a reasonable computational time. One popular method for finding the global optimum is the constraint generation (CG) algorithm. Traditionally, the initial feasible solution of CG is given by the greedy algorithm. It turns out that choosing different starting point than the greedy solution might lead to better performance in terms of running time. In this paper we introduce such a strategy which is beneficial on non-complete bipartite graphs. Benchmarking results on different versions of the solution methods are shown to demonstrate the efficiency of the proposed methods.

Keywords: submodular function maximization, combinatorial optimization, integer programming

1 Introduction

Nemhauser and Wolsey concluded in their seminal paper [17] that four problems are fundamental to solving an integer linear problem in terms of practicality: formulation of the model, selection of an initial set of constraints, decisions in a branch-and-bound algorithm, finding good feasible solutions. Among these essential issues, we are concerned with that of the starting point selection, i.e., the selection of initial constraints that would make the algorithms more efficient in terms of runtime.

*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.

[a]University of Szeged, Hungary
[b]E-mail: csokas@inf.u-szeged.hu, ORCID: 0009-0009-4863-2596
[c]University College Roosevelt, Middelburg, The Netherlands
[d]E-mail: tvinko@inf.u-szeged.hu, ORCID: 0000-0002-3724-4725

DOI: 10.14232/actacyb.309396

There are integer or mixed-integer linear programs that require exponentially large numbers of linear constraints. A well-known example is the traveling salesman problem, which has exponentially many constraints to eliminate subtours [4]. A general group of these problems are mixed integer linear programs to be solved by Benders decomposition. The basic approach is to start with a reduced relaxed problem with a small number of constraints. During the algorithm execution, additional constraints are generated if the existing ones are violated. When selecting the initial constraints, care should be taken to ensure that they are important or significant in some sense during the solution process. The recommended initial constraints are the solution of the greedy algorithm [15, 17].

1.1 Submodular function

Let the finite set N be defined as $N = \{1, \dots, n\}$. A function $f : 2^N \to \mathbb{R}$ is called *submodular* if for every $S, T \subseteq N$,

$$f(S) + f(T) \geq f(S \cap T) + f(S \cup T).$$

Perhaps more intuitive is the following equivalent definition:

$$f(\{i\} \mid S) \geq f(\{i\} \mid T)$$

holds for every $S \subseteq T \subseteq N$ and $i \in N \setminus S$ where $f(\{i\} \mid S) := f(S \cup \{i\}) - f(S)$.

The function f is *monotone* if $f(\{i\} \mid S) \geq 0$ for all $i \in N \setminus S$.

A submodular function f is *non-decreasing* if $f(S) \leq f(T)$ holds for all $S \subseteq T \subseteq N$. In the following we assume that f satisfies this property, i.e., it is a non-decreasing submodular function.

The goal of the submodular maximization problem with a *cardinality constant* $0 < k \leq n$ is to find a subset $S \subseteq N$ maximizing $f(S)$ such that $|S| \leq k$.

1.2 Solvability

In contrast to the task of minimizing submodular functions, which can be done in polynomial time [8, 10, 22], maximizing non-decreasing submodular functions is known to be NP-hard [5, 14]. The greedy method for solving the non-decreasing, monotone submodular maximization problem with cardinality constraint was proposed in [16]. They showed that the greedy strategy achieves an $(1 - 1/e)$-approximation of the optimal solution.

Although this is a simple and efficient technique for solving many optimization problems and is therefore very commonly used, the global optimum is often more needed in real-world applications, which was the motivation for a new solution method proposed in [17]. This is based on a mixed integer programming (MIP) model:

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

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 $\mid S \mid \leq k$.

Since the number of constraints increases exponentially in (1), this motivated a new procedure, the so-called constraint generation (CG) algorithm, proposed in [17]. CG is an iterative algorithm that starts with solving a reduced problem. The reduced problem consists of a set of constraints generated from the initial set, which is extended on demand at each iteration by the addition of a feasible solution. So in essence it solves many reduced MIP problems, which are not always sufficiently efficient in applications. For this reason, the branch-and-bound algorithmic approach is often used, which exploits the relaxation of MIP.

1.3 Constraint generation algorithm (CG)

A constraint generation algorithm was proposed in [17], which starts from a reduced MIP problem to start with a few constraints to solve. It is an iterative algorithm, and in every iteration solving the reduced MIP problem while adding a new constraint. Let define the reduced problem MIP(Q), where $Q \subseteq F$ is a set of feasible solutions:

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

Note that MIP(Q) is a reduced problem of (1). An optimal solution vector $\hat{\mathbf{y}}$ of MIP(Q) for which the corresponding set $\hat{S}$ is in Q is also an optimal solution of the original problem (1).

The pseudo code of CG is shown in Algorithm 1. The starting point of the algorithm is a set $Q = \{S^{(0)}_{[0]}, \ldots, S^{(0)}_{[k]}\}$, where $S_{[i]}$ is the first i elements of a feasible solution $S^{(0)}$ which comes from the order of the greedy algorithm's solution. In the t-th iteration we solve the problem MIP(Q), $Q = \{S^{(0)}_{[0]}, \ldots, S^{(0)}_{[k]}, \ldots, S^{(t-1)}\}$ to obtain the optimal solution $\mathbf{y}^{(t)} = (y^{(t)}_1, \ldots, y^{(t)}_n)$ and $z^{(t)}$ the optimal value which gives an upper bound of the problem (1). Let S^* be the best feasible solution of problem (1) up to this point and $S^{(t)} \in F$ be the set which is generated the optimal solution $\mathbf{y}^{(t)}$ of MIP(Q), i.e., $\mathbf{y}^{(t)}$ is the characteristic vector of $S^{(t)}$. When $f(S^{(t)}) > f(S^*)$ holds, then update S^* with $S^{(t)}$. If $z^{(t)} > f(S^*) \geq f(S^{(t)})$ holds,

Algorithm 1 CG($S^{(0)}$)

```
Input The initial feasible solution $S^{(0)}$.
Output The optimal solution $S^*$ of problem (1).
Step 1: Set $Q \leftarrow S^{(0)}$, $S^* \leftarrow S^{(0)}$ and $t \leftarrow 1$.
Step 2: Solve MIP($Q$). Let $z^{(t)}$ be the optimal value of MIP($Q$) and $S^{(t)}$ is the set
        corresponding to the optimal solution $\mathbf{y}^{(t)}$ 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 $Q \leftarrow Q \cup \{S^{(t)}\}$, $t \leftarrow t+1$ and return to Step 2.
```

then we have that $S^{(t)} \notin Q$, so (in Step 5) add $S^{(t)}$ to Q. This effectively adds the following constraint to MIP(Q):

$$z \leq f(S^{(t)}) + \sum_{i \in N \setminus S^{(t)}} f(\{i\} \mid S^{(t)}) \cdot y_i. \tag{3}$$

The algorithm stops when $z^{(t)} = f(S^*)$ is satisfied which means that it is proven that the optimal solution is found.

1.4 From where to start the CG algorithm?

It can be observed that the choice of the starting point plays a role in the efficiency (i.e., its runtime) of the CG algorithm. More precisely, starting from a high function-valued initial point *might not* provide the fastest runtime. This effect is illustrated in Figure 1. Exact details of the test problem (called C.60.5.3) will be given later in Section 5, right now it is enough to specify that the task of the CG algorithm was to select the optimal $k = 5$ nodes from a graph of 60 vertices. The globally optimal solution for this benchmark example was known.

We created 250 test cases, part of which started from a random starting point, while in the other part we chose 3 of the best 5 vertices belonging to the global optimum, fixed them and randomly added 2 other nodes. The reason for this is that we did not get an initial set with a larger function value in the random choices, so we have also biased the sensitivity analysis a bit towards the more interesting scenarios.

Figure 1 shows the scatter plot of the 250 test cases. It is important to note that in this figure, the x-axis shows the runtime of the CG algorithm and the y-axis shows the function value of the initial set. In the figure, two sets of points can be roughly separated, due to the semi-random chosen test cases. The green dot indicates the original CG algorithm starting from the initial point proposed by

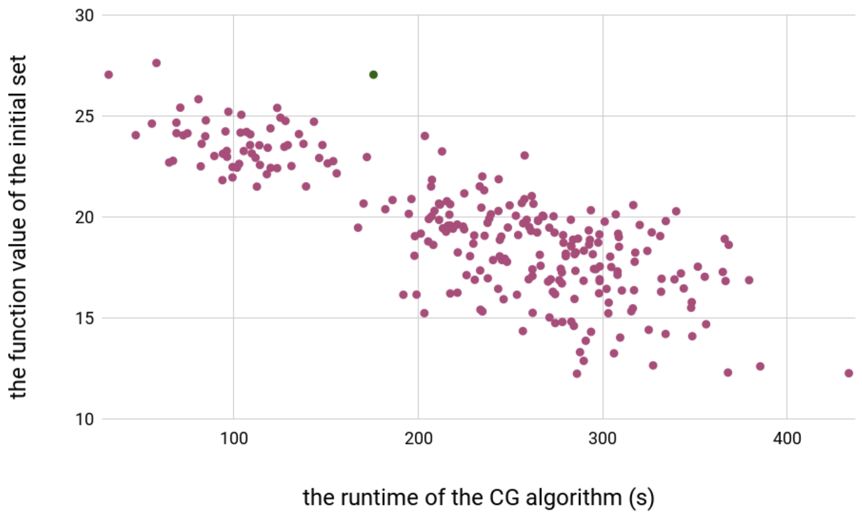

Figure 1: Visualization of the sensibility of the starting point: starting points with similar function values can have rather different running times

greedy. Note that for the other results, we used all subsets of randomly generated points as starting point since we did not have any order. This figure perfectly illustrates that the running time of a CG algorithm can be very different even if the function value of the initial points are similar.

The phenomenon introduced informally above is the main motivation for our paper.

1.5 Roadmap

The relevant literature summary follows in Section 2. The algorithms investigated in this paper are reviewed in Section 3. A new starting point generation rule for accelerating the algorithms is discussed in Section 4. In Section 5, we present our numerical experiments, the benchmark instances that we used for testing the algorithms, including a description of the computational environment, the properties of the test graphs and then the benchmarking results. Finally, Section 6 concludes the work.

2 Related work

CG-type algorithms are often used to solve mixed-integer linear problems (MILP), hence their efficiency is of high interest. A popular approach is to use machine learning to improve the speed of these algorithms. By achieving a good initial set, the number of iterations is reduced, which reduces the computational cost. In this spirit, a learning strategy was proposed in [19] that employs a modified nearest neighbor method to filter out redundant constraints in the Unit Commitment (UC) problem. In [11], a machine-learning-aided warm-start constraint generation algorithm was introduced which speeds up the search for the optimal solution of a MILP. The method is constructed on the offline detection of the invariant constraint sets of earlier occurrences of the target MILP. This significantly improves the prediction of invariant constraint sets for instances that have not yet been seen. Thus, much fewer iterations are required to run the constraint generation algorithm and the online computational burden is significantly reduced. A similar idea for solving MIP problems can be found in [25], where a machine learning technique has been proposed. This is based on extracting efficient data from previous instances in order to improve the solution for similar instances. Good initial feasible solutions, affine subspaces, and redundant constraints were predicted based on statistical data, leading to a significant reduction in problem size.

Focusing on the submodular function maximization problem, most solving algorithms use the result of the greedy approach as a starting point, i.e., the constraints of the initial set, as proposed in [15, 17]. Correspondingly, in [23], the constraint generation procedure and its improved versions used the subsets generated from the greedy result as starting points. The study in [24] investigated fairness and balancing utility in submodular maximization, which was formulated as a bicriteria optimization problem. For this, two instance-dependent approximation algorithms were introduced. In these solving methods, the initial set is also the solution of the greedy algorithm. The well-known benchmarks (maximum coverage, influence maximization and facility location) were used to test the efficiency.

A cutting-plane algorithm for submodular maximization problems was described in [12], which is in fact an iterative binary-integral linear programming model. They used a so-called submodular cut plane, based on the submodularity of a set function via the Lovász extension, which ensures that the algorithm converges to the optimum in finite iterations. In the experiments, they used the solution by a greedy algorithm as initial subset.

3 Variants of constraint generation algorithm

In this section, we briefly describe the algorithms that are used in the paper. The basic algorithm, the constraint generation algorithm, was presented earlier, so variations of it follow. As the procedures are already available in the literature, we will only give a general description of them, without the exact details. For the complete discussion the reader is referred to the cited papers below.

3.1 Improved constraint generation (ICG)

In [23], the authors proposed an improved generation method based on CG, where not one but several constraints are added per iteration. It complements Algorithm

1 by creating a new set Q^+ containing the elements of the set Q and the result of the internal sub-algorithm (SUB-ICG). Step 5 of the CG algorithm is completed by calling SUB-ICG and adding its return value to the set Q^+. If a solution is found in this part of the algorithm whose function value is greater than the current $f(S^*)$, it is also updated.

The SUB-ICG is an iterative algorithm that generates $\lambda = 10 \cdot k$ new feasible solutions (i.e., k vertices are selected). To do this, it uses a heuristic method to assign a value p_i to the vertices $i \in N$. This is based on the number of times the vertices $i \in N$ appear in the sets $S \in Q$. These are recalculated after updating the set Q. Finally, the final selection order is given by r_i, which is generated randomly such that $0 \le r_i \le p_i$.

3.2 ICG with reduced k (ICG($k-1$))

We proposed ICG with reduced k in [2]. What has been changed from the ICG is that in SUB-ICG we choose $k-1$ nodes for the constraints. Thus, the function value calculated in (2) computes the value of the k-th vertex when adding it to the set. We have kept this change for the remaining algorithm variants, i.e., we choose $k-1$ nodes for the constraints in both GCG and ECG, which are presented below.

3.2.1 ICG using graph structure (GCG)

In this variant of ICG($k-1$), that we proposed in [2], we changed the heuristic that calculated the value p_i to select the nodes in the SUB-ICG. The new heuristic is based on the structure of the input graph. If the graph is fully connected, then we do not consider all of the edges. Specifically, we compute the median of the outgoing edges' weights for every vertex $j \in M$. Edges with weights less than the median are ignored. The p_i value is the sum of the weights of the incoming edges at node $i \in N$ normalized with the targets node's degree. Accordingly, the value of r_i is uniformly randomly adjusted so that $0 \le r_i \le p_i$. We also use the concept coming from ICG($k-1$), so we select $k-1$ nodes.

3.2.2 ICG using enumeration (ECG)

In contrast to ICG and GCG, in this variation no sub-algorithm is repeated within the main algorithm, which can result in less computation time. ECG was proposed in [2]. Instead of the sub-algorithm, we add a few additional steps to the Algorithm 1 so that we still generate λ constraints per iteration. We choose some nodes from the union of the set $S^{(t)}$ and a randomly chosen set Q of feasible solutions satisfying certain conditions. The choice is based on a graph structure according to the largest p_i values. From these we generate subsets with cardinality at most $k-1$. From these we generate subsets with cardinality at most $k-1$ and compute their function values. Of these we keep λ subsets with the smaller function values. These correspond to the return sets of SUB-ICG.

3.3 A modern implementation

In this subsection, we would like to present a brief summary of a relevant and efficient method for submodular maximization, as found in [21]. A modern implementation of the MIP model of Nemhauser and Wolsey based on lazy constraint generation was presented. These procedures were also used to test the new initial point selection strategy. An analogy was discovered between the MIP model of submodular function maximization and the Benders decomposition [1]. Specifically, they were able to exploit some of the algorithmic improvements proposed for the Benders decomposition. That is, they took advantage of the support of modern MIP solvers for lazy constraints, so they could provide a stronger initial primal constraint and could also improve the dual constraint faster.

In terms of runtime, these algorithms are much faster than the algorithms presented previously, as can be seen from the numerical results. However, we would like to note that we are not demonstrating the effectiveness of the solving algorithms, but to show how the starting point selection strategy works.

3.3.1 GRASP heuristic

At the end of the greedy phase, a local search was inserted using a neighborhood structure. This was based on the fact that replacement neighbourhoods were examined: subsets of elements that had been dropped and a new element not yet in the set added. The GRASP heuristic can be obtained with the greedy algorithm extended by local search, based on [6]. Then, a randomized component is added to the greedy procedure, i.e., a candidate is chosen randomly, which is then corrected by local search. This is iterated until an iteration/resource limit is reached.

3.3.2 Separating fractional solutions

Separating fractional solutions is not trivial, but separating integer solutions is easy. This is because for fractional solutions, the point to be cut cannot be mapped to a subset of the ground set. This was the motivation for proposing two heuristic solutions for decomposition in [21]. One is based on the greedy algorithm, while the other is based on the Lovász-expansion.

Of the methods summarized above, 3 algorithms have been constructed and tested in [21]: *base* is the basic constraint generation method, *bc* has constraints separated as lazy constraints on the fly without using custom branching, while *bc+* is the improved version of *bc* and custom branching are included. *bc* and *bc+* use GRASP as a warm start, so we used these algorithms to test our new starting point selection strategy.

4 A new starting point for constraint generating algorithms

To find a new starting point for a CG-type algorithm we use the input graph's structure. The idea is based on our previous work [2] (cf. calculate the p_i value), but the procedure used there is not directly appropriate. In fact, the current approach is completely new, and it was not used in our previous work. Moreover, in this method, the nodes are dynamically selected.

To choose k node as a new starting point, first of all, we calculate a new centrality[1] value ns_i to every node $i \in N$. This centrality is adding up the weights of the incoming edges at node $i \in N$ normalized with the degree of the targets node, then multiplying the sum with the degree of the source node:

$$ns_i = d_i \cdot \sum_{j:(j,i)\in E(G)} \frac{w_{ji}}{d_j}, \tag{4}$$

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, d_j is the degree of the node $j \in M$ and d_i is the degree of node $i \in N$. Observe that this formula works well for graphs that are not fully connected, because weighting by the degree of the vertices only makes sense then.

Choose node i with highest ns_i value and delete node i with their edges and recalculate all the ns_i for every node $i \in N$. The next vertex is chosen for the starting point based on the recomputed centrality value. Repeat this method until k nodes are selected.

See the graph on Figure 2 as a small illustrative example, where the labels of the nodes are marked with black numbers. The vertices signed by their labels and their corresponding ns_i values calculated by (4) are shown next to them highlighted by tanning color. Taking the ns_i values into account, we first choose node 7 and then delete this vertex with its edges. Then, the result graph is shown on Fig. 3 with its recalculated ns_i values. Accordingly, the next selected vertex is 5 and not node 6, but note that in the first step it seemed that node 6 is the better choice.

Although we choose k nodes from N, we also use the vertices from M to calculate the new centrality metric. We divide the weight of the outgoing edges $j \in M$ by the degree of j, which is used to normalize the effect of the edges. We sum this normalized incoming edge weights for each $i \in N$ nodes, which expresses the average effect of selecting $i \in N$ nodes relative to the other N nodes. This is weighted by the degree of node i, which helps to better scale the effect of the N vertices.

Let's revisit the example in Figure 2. Node 4 can be served only if node 7 is selected, because there is only one edge from node 4 to node 7, so node 7 is important. When computing the centrality metric, we add an edge weight to the value of this peak, which is divided by 1 (i.e., it remains itself), thus greatly increasing the value of ns_7. There are two other vertices connected to vertex 7 with two edges,

[1]importance of graph nodes is indicated by a centrality measure; it is a non-negative value assigned to each vertex [18].

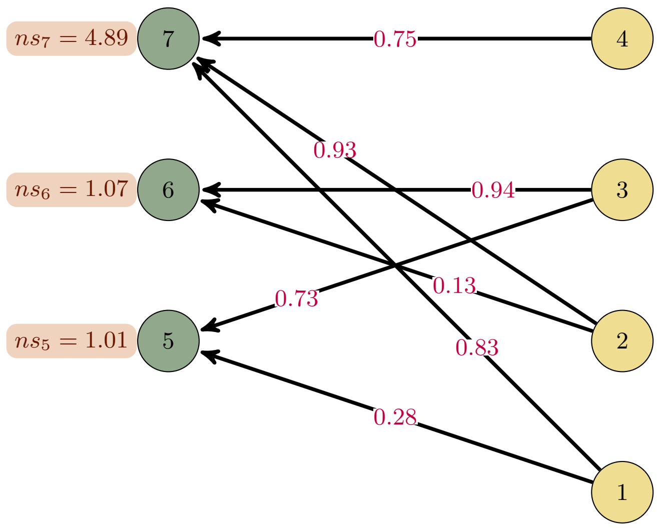

Figure 2: Example graph to calculate the ns_i values; initial step

but in both cases the edge with the higher weight is connected to the node 7. This effect is further increased by multiplying the value by the degree of the node.

Next, we delete node 7 with its edges, thus obtaining a new graph showing the case where vertex 7 no longer serves a function. This is the reason why, in the case of the graph in Figure 3, we now choose vertex 5. In the current state of the graph, the value of ns_5 is larger, i.e., the vertex is more important, because there is only one edge coming out of vertex 1, and its value is larger than the edge coming into vertex 6 from vertex 2. Notice that we always choose the most important vertex of the current ones. This is regardless of the edge weight going to the previously selected vertex from vertices $j \in M$ (e.g., the vertex 7 has a high weight edge going to it from 3). This is crucial for the strength of the method.

Finally, we generate all subsets from the k vertices proposed by the new centrality metric. We start a CG-type algorithm from the constraints defined by these subsets.

5 Numerical experiments

5.1 Computational environment

The implementation of the above proposed new centrality were done in R version 4.3.2 using its `igraph` 1.4.2 package. For numerical experiments, CG-type algorithms [2] are implemented in AMPL [7], using the original code with the parameters in [2]. The solver CPLEX 22.1.1.0 called from AMPL using default options. The modern implementations, the *bc* and *bc*+ algorithms, were previously provided by the author of [21]. Thus the original C++ code was used for testing and the

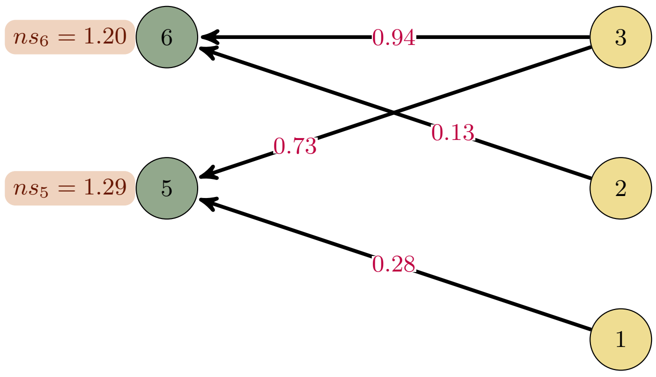

Figure 3: Example graph to calculate the ns_i values; result of the first iteration

settings in [21] were not changed. The MIP solver is IBM ILOG CPLEX 22.1.1 [9]. The computer used had Intel Core CPU i5-6500 at 3.20GHz with 64G memory running Ubuntu Linux 22.04.3.

5.2 Problem instances

We use two types of well-known and often employed instances with a non-decreasing submodular property, called weighted coverage (COV) and bipartite influence (INF) [13, 20]. It is important to note that the COV problem has simple MIP formulation that can be solved rather efficiently with standard MIP solvers, see [23]. In contrast, the INF problem, which we discussed below, cannot be formulated as a simple MIP, and this justifies the attempt to make the universal submodular maximization framework more efficient [21].

5.2.1 Weighted coverage (COV)

Let $M = \{1, \ldots, m\}$ be the set of items, where m is the number of items and $N = \{1, \ldots, n\}$ be the set of sensors, where n is the number of sensors. Each sensor $j \in N$ covers a subset of elements $M_j \subseteq M$, and each item $i \in M$ has a non-negative weight w_i. We select a set of sensors $S \subseteq N$ to cover the items. Let $a_{i,j} = 1$ be, if $i \in M_j$ and $a_{i,j} = 0$ otherwise. The definition of the the total weighted coverage is the following:

$$f(S) = \sum_{i \in M} w_i \max_{j \in S} a_{i,j}. \tag{5}$$

5.2.2 Bipartite influence (INF)

Let n be the number of items and m be the number of targets. The set of items $N = \{1, \ldots, n\}$ and the set of targets $M = \{1, \ldots, m\}$ are given. Let the influence maximization problem be defined on a bipartite graph $G = (M, N; E)$, where $E \subseteq M \times N$ is the 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. Then, the determination of the number of targets activated by the set of elements $S \subseteq N$:

$$f(S) = \sum_{i \in M} \Big(1 - \prod_{j \in S} (1 - q_{ij})\Big). \tag{6}$$

Let's notice that the formula of INF as a submodular function (6) includes a product containing the elements of the set S, which contains the product of the elements from the set S. It could be formulated with binary variables to select the set elements, and that would result in a polynomial type non-linear program. This could possibly be converted to MILP with e.g., McCormick formalism but it is neither simple nor promising in terms of its solvability.

5.3 Test graphs

All the algorithms presented in Section 3 were re-run during testing to ensure a fair comparison with the procedures started from the new point (i.e., same configurations and software versions). We also tested the algorithms used in [2, 21] and their versions started from a new initialization point, a short summary of this is given below. For both problems we used the benchmarks from [23], since they are available online.

Following the approach in [23] we had:

- $N = 20, 40, 60, 80, 100$;
- $M = N + 1$ and $k = 5, 8$.
- 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, an edge is randomly generated with probability $p = 0.1$ probability and p_j is a random value taken from the interval $[0, 1]$.
- We had $\lambda = 10 \cdot k$.
- The cardinality of set Σ in ECG: $\kappa = 12$, see [2] for details.
- Finally, all the random parameters were generated with uniform distribution.

5.4 Benchmarking results

Detailed results are available in an electronic supplement [3] containing 24 tables, for the transparency of the publication. The average results for each instance are given in Table 1 and Table 2.

Table 1: Summary of results: starting from *greedy* versus *NS* (using CG-type algorithms)

inst., k	algorithm	time	iter.nr.	cons.nr.	NS-time	NS-iter.nr.	NS-cons.nr.
COV, 5	CG	478.14	95.65	100.65	**115.82**	63.00	94.00
	ICG	302.05	17.80	597.28	**204.51**	15.92	542.02
	ICG($k-1$)	261.33	20.53	715.82	**186.86**	18.80	664.24
	GCG	184.53	14.79	649.67	**138.70**	14.09	628.55
	ECG	955.36	45.38	383.42	**583.16**	36.92	363.92
COV, 8	CG	**290.41**	22.78	30.78	565.18	24.17	279.17
	ICG	**541.56**	7.11	432.98	666.79	7.85	707.11
	ICG($k-1$)	441.73	7.61	478.57	**427.16**	8.85	778.79
	GCG	**394.74**	6.39	442.13	542.99	7.37	767.75
	ECG	591.54	8.65	398.30	**494.53**	8.31	745.98
INF, 5	CG	426.19	197.00	203.00	**243.38**	104.10	135.10
	ICG	904.53	51.76	1219.89	**15.92**	13.75	300.24
	ICG($k-1$)	560.43	38.77	1025.61	**68.78**	15.20	388.13
	GCG	591.83	33.28	1086.53	**72.16**	14.18	460.08
	ECG	0.86	3.68	88.68	**0.85**	3.00	93.60
INF, 8	CG	20.52	81.00	90.00	**2.93**	14.33	269.33
	ICG	524.63	26.04	1194.26	**6.66**	7.56	441.08
	ICG($k-1$)	202.74	18.12	917.20	**4.10**	5.30	425.83
	GCG	947.46	25.21	1486.46	**40.66**	9.86	770.40
	ECG	442.63	17.22	1322.61	**36.65**	6.52	703.26

Table 2: Summary of results: starting from *GRASP* versus *NS* (using the modern implementation)

inst., k	algorithm	time	NS-time
COV, 5	*bc*	3.04	**2.40**
	bc+	2.84	**1.66**
COV, 8	*bc*	49.09	**48.18**
	bc+	34.31	**33.34**
INF, 5	*bc*	0.35	**0.11**
	bc+	0.32	**0.13**
INF, 8	*bc*	**3.62**	4.31
	bc+	**5.96**	9.80

Five instances were tested for each class, indicated by the last digit of the instances. All algorithms, for each task, were run 5 times using different random seeds for the heuristic choices. The time limit used for the runs was 7,200 sec (2 hours). If, for any instance, not all of the 5 runs were completed within the time limit, the number of successful runs was indicated in brackets. The instances that did not run within the limit were described by the the mean relative gap[2] with the average number of cases counted in brackets behind it.

The following is a textual assessment of the tables. We denote new start algorithms with NS prefix. Where all of the instances were successful, we have highlighted in the table the one that was faster. In the textual evaluation, we use the ratio (*time*/NS-*time*) and (NS-*cons.nr.*/*cons.nr.*) to express the change in time and the number of constraints. For the comparison of iteration numbers, we used (NS-*iter.nr.*/*iter.nr.*) $\times$ 100 percent.

For the average values in Table 1, only graph instances were considered which could ran within the given time limit in all 5 cases using the given algorithm and its NS variant. The table shows that the NS algorithm was faster on average in most cases.

COV, $k = 5$ The results are reported in Tables 3-8 in the supplement [3]. Considering all of these methods together, the algorithms started from the new point were able to solve the problems within the time limit in more cases. For the successful instances, NS-CG was able to speed-up the running time the most, it was 2.46 times faster, while the iterations were decreased to 68.22% and the number of constraints was 1.70 times more compared to CG. Next was NS-ECG with 2.00 times speed-up, 71.06% reduction in the number of iteration and 1.54 times the number of constraining conditions. NS-ICG was 1.54 times faster and reduced iteration to 85.85%, while using 1.05 times more constraints to solve the problem. NS-ICG($k-1$) and NS-GCG achieved similar results: NS-ICG($k-1$) achieved 1.37 times faster with less iterations (90.23%) and 1.05 times more constraints; NS-GCG was 1.32 times faster, reduced iterations by 93.41% and used 1.07 times more constraints.

Examining the *bc* and *bc*+ algorithms, we can see that by replacing the GRASP heuristic with the NS starting point, *bc* is 3.56 times faster, while *bc*+ is 3.53 times faster on average. In the case of *bc*, there are 2 graph instances where the GRASP heuristic algorithm runs faster.

COV, $k = 8$ The results are reported in Tables 9-14 in the supplement [3]. The results obtained for this instance are very interesting. Although we started all NS algorithms from the same new point, we did not achieve improvements in many cases. One reason for this is that the number of all the subsets of the $k = 8$ selected vertices is large, so there are already many constraints when starting the reduced MIP problem. Another reason is that algorithms generate new constraints

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

in different ways per each iteration after the initialization. This results in an average speedup of 1.01 times for NS-ECG, while no speedup was achieved for the other algorithms. For NS-ECG, there were 176.15% more iterations and 15.08 times more constraint conditions compared to ECG. NS-CG was 0.38, NS-ICG was 0.57, NS-ICG$(k-1)$ was 0.59 whereas NS-GCG was 0.61 slower than the corresponding greedy initiated methods. Even the number of iterations and constraints has increased. Note that this is the only group of problems where the algorithms from the new starting point did not show absolute success. Much the same phenomenon can be observed when examining the relative gap values.

Using the NS starting point instead of the GRASP heuristic, the average speedup is 3.58 for *bc* and 4.25 for *bc*+. Note that, there are a few cases where *bc*, *bc*+ runs faster than NS-*bc*, NS-*bc*+; to be precise, 8 graphs for *bc* and 8 graphs for *bc*+.

INF, $k = 5$ The results are reported in Tables 15-20 in the supplement [3]. The best results were obtained for NS-ICG, with an average 32.11 faster runtime, 32.66% reduction in the number of iterations and 80.28% reduction in the number of constraints. As with NS-ICG, NS-ICG$(k-1)$ and NS-GCG ran faster than the original algorithm for all examples and reduced both the number of iterations and the number of constraints. In numerical terms, ICG$(k-1)$ achieved 8.84 times faster runtime and reduced the number of iterations by 40.71% and the number of constraint conditions by 66.54%; GCG similarly achieved 6.82 times faster runtime and reduced the number of iterations by 40.71% and the number of constraint conditions by 54.99%. NS-CG and NS-ECG did not win in terms of runtime for all graphs, but when looking at the average runtime results, they still ran faster. CG achieved a speedup of 3.82 times, while increasing the number of iterations and constraint conditions (by 1.08 times and 1.82 times, respectively). For ECG, the average speedup was 1.18, but note that the ECG solution time for these examples is under 5 sec.

The gap values are similar: the gap values of the NS algorithms are smaller, and note that there were several times when the original algorithm could not run within the time limit, while the NS algorithms solved the problem in a short time.

For these tasks, we can achieve an average speedup of 3.31 times on *bc* and 2.64 times on *bc*+.

INF, $k = 8$ The results are reported in Tables 21-26 in the supplement [3]. For these instances, there were many cases where the algorithms could not solve within the time limit. That is why we can see several instances where only the NS procedure solved. The best average speedup here was also achieved with NS-ICG, exactly 8.33 times, while the iteration was less than halved (44.41%), but the number of constraints was almost 4 times more (3.89). We achieved similarly good results with NS-ICG$(k-1)$, with 15.50 times faster. NS-GCG also achieved 9.63 times faster results with half as many iterations (50.44%). ECG ran the most graphs, so we were able to make the most comparisons here and was 6.84 times faster with NS-ECG with no increase in the number of constraints (99.07%). In contrast, CG used 9.04 more constraints but also reduced the runtime by 8.33 times. Comparing the gap values, we can see that in most cases the NS algorithms achieved smaller gap values. In fact, there are 23 cases where the NS ran within the time limit in all 5 cases, while starting from greedy, the algorithm could not.

For the *bc* and *bc*+ algorithms, the average speedup is 2.09 and 2.04, respectively. For these instances, both for the *bc* and *bc*+ algorithms, the average runtime is lower when starting from the GRASP heuristic. Accordingly, this is where most of the graph instances where the GRASP heuristic algorithm proved to be faster (7 cases for *bc* and 8 cases for *bc*+).

6 Conclusion

A new centrality metric based on the input graph structure was proposed. The graphs under consideration are not fully connected bipartite graphs, for which we have used both the edge weights and the degrees, taking into account the baseline problem. The centrality metric is dynamically recalculated after selecting and deleting a vertex, and the resulting node ordering is used to select the initial set of submodular function maximization problems. The importance of the choice of the starting point was already stated by Nemhauser and Wolsey in [17]. In most cases, the solution proposed by the greedy method or randomly selected feasible solution is used as the starting point for solving algorithms. But there are other proposals in the literature, more precisely, we have presented here the GRASP [21] heuristic.

We used five different algorithm variants for the non-decreasing submodular function maximization problem based on a MIP formulation using constraint generation approach which we started from the greedy's solution and also from the new starting point proposed by the centrality metric. Furthermore, we used two modern implementations of Nemhauser and Wolsey's MIP model for submodular function maximization problem based on lazy constraint generation, which we started from the GRASP heuristic and also from the new starting point proposed by the centrality metric. According to our benchmarking results, algorithms starting from the new initial set reduced the runtime by a factor of 5.37 for all test cases. Overall, we can conclude the initial set suggested by the new centrality metric is worth using, as shown by our run-time tests and, in their absence, the relative gap tests.

Acknowledgements

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

References

[1] Benders, J. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962/63. DOI: 10.1007/

BF01386316.

[2] Csókás, E. and Vinkó, T. Constraint generation approaches for submodular function maximization leveraging graph properties. *Journal of Global Optimization*, 88:377–394, 2024. DOI: 10.1007/s10898-023-01318-4.

[3] Csókás, E. and Vinkó, T. On the initial set of constraints for graph-based submodular function maximization — supplementary information. URL: https://www.inf.u-szeged.hu/~tvinko/CsV-submodular_max-supplement2.pdf.

[4] Dantzig, G., Fulkerson, R., and Johnson, S. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954. DOI: 10.1287/opre.2.4.393.

[5] Feige, U., Mirrokni, V. S., and Vondrák, J. Maximizing non-monotone submodular functions. *SIAM Journal on Computing*, 40(4):1133–1153, 2011. DOI: 10.1109/FOCS.2007.29.

[6] Feo, T. A. and Resende, M. G. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67–71, 1989. DOI: 10.1016/0167-6377(89)90002-3.

[7] Fourer, R., Gay, D. M., and Kernighan, B. W. *AMPL. A Modeling Language for Mathematical Programming.* Thomson, Duxbury, 2003. DOI: 10.1287/mnsc.36.5.519.

[8] Grötschel, M., László, L., and Schrijver, A. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981. DOI: 10.1007/BF02579273.

[9] IBM. ILOG CPLEX 22.1.1 User's Manual, 2022. URL: https://www.ibm.com/docs/en/icos/22.1.1.

[10] Iwata, S., Fleischer, L., and Fujishige, S. A combinatorial strongly polynomial algorithm for minimizing submodular functions. *Journal of the ACM*, 48(4):761–777, 2001. DOI: 10.1145/502090.502096.

[11] Jiménez-Cordero, A., Morales, J. M., and Pineda, S. Warm-starting constraint generation for mixed-integer optimization: A machine learning approach. *Knowledge-Based Systems*, 253:109570, 2022. DOI: 10.1016/j.knosys.2022.109570.

[12] Kawahara, Y., Nagano, K., Tsuda, K., and Bilmes, J. A. Submodularity cuts and applications. In *Advances in Neural Information Processing Systems*, Volume 22, pages 1–9, 2009. URL: https://api.semanticscholar.org/CorpusID:1303767.

[13] Krause, A. and Golovin, D. Submodular function maximization. *Tractability*, 3:71–104, 2014. DOI: 10.1017/CBO9781139177801.004.

[14] Krause, A. and 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*, pages 1339–1345, 2005. DOI: 10.1184/R1/6608123.v1.

[15] Minoux, M. Accelerated greedy algorithms for maximizing submodular set functions. In Stoer, J., editor, *Optimization Techniques*, pages 234–243. Springer Berlin Heidelberg, 1978. DOI: 10.1007/BFb0006528.

[16] Nemhauser, G., Wolsey, L., and Fisher, M. An analysis of the approximations for maximizing submodular set functions — I. *Mathematical Programming*, 14:265–294, 1978. DOI: 10.1007/BF01588971.

[17] Nemhauser, G. and Wolsey, L. Maximizing submodular set functions: formulations and analysis of algorithms. *Studies on Graphs and Discrete Programming*, pages 279–301, 1981. DOI: 10.1016/S0304-0208(08)734.

[18] Newman, M. *Networks: An Introduction*. Oxford University Press, 2010. DOI: 10.1093/acprof:oso/9780199206650.001.0001.

[19] Pineda, S., Morales, J. M., and Jiménez-Cordero, A. Data-driven screening of network constraints for unit commitment. *IEEE Transactions on Power Systems*, 35:3695–3705, 2019. DOI: 10.1109/TPWRS.2020.2980212.

[20] Sakaue, S. and 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. DOI: 10.1609/aaai.v32i1.11521.

[21] Salvagnin, D. Some experiments with submodular function maximization via integer programming. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 488–501. Springer International Publishing, 2019. DOI: 10.1007/978-3-030-19212-9_32.

[22] Schrijver, A. A combinatorial algorithm minimizing submodular functions in strongly polynomial time. *Journal of Combinatorial Theory, Series B*, 80:346–355, 2001. DOI: 10.1006/jctb.2000.1989.

[23] Uematsu, N., Umetani, S., and 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. DOI: 10.15807/jorsj.63.41.

[24] Wang, Y., Li, Y., Bonchi, F., and Wang, Y. Balancing utility and fairness in submodular maximization. *CoRR*, abs/2211.00980, 2023. DOI: 10.48786/EDBT.2024.01.

[25] Xavier, A. S., Qiu, F., and Ahmed, S. Learning to solve large-scale security-constrained unit commitment problems. *INFORMS Journal on Computing*, 33(2):739–756, 2021. DOI: 10.1287/ijoc.2020.0976.

Received: 6th May 2024