Some experiments with submodular function maximization via integer programming

Domenico Salvagnin

Department of Information Engineering (DEI), University of Padova
domenico.salvagnin@unipd.it

Abstract. Submodular function maximization is a classic problem in optimization, with many real world applications, like sensor coverage, location problems and feature selection, among others. Back in the 80's, Nemhauser and Wolsey proposed an integer programming formulation for the general submodular function maximization. Being the number of constraints in the formulation exponential in the size of the ground set, a constraint generation technique was proposed. Since then, the method was not developed further. Given the renewed interest in recent years in submodular function maximization, the constraint generation method has been used as reference to evaluate both exact and heuristic approaches. However, the outcome of those experiments was that the method is utterly slow in practice, even for small instances. In this paper we propose several algorithmic enhancements to the constraint generation method. Preliminary computational results show that a proper implementation, while still not scalable to big instances, can be significantly faster than the obvious implementation by the book. A comparison with direct mixed integer linear programming formulations on some classes of models that admit one also show that the submodular framework, in its generality, is clearly slower than dedicated formulations, so it should be used only when those approaches are not viable.

1 Introduction

Let N be a finite set of n elements, called the ground set. A set function $f : 2^N \to \mathbb{R}$ is called *submodular* if it satisfies the following property:

$$f(S) + f(T) \geq f(S \cup T) + f(S \cap T) \quad \forall S, T \subseteq N \tag{1}$$

Equivalently [17], a set function f is submodular if and only of it satisfies the *diminishing returns* property:

$$f(S \cup j) \geq f(T \cup j) \quad \forall S \subseteq T \subseteq N, j \in N - T \tag{2}$$

In other words, the later we add an element j to a set, the smaller its effect is on the objective. Many real-world optimization problems give rise to submodular functions like, e.g., location problems and sensor coverage [14]. In addition, many

problems in computer science and machine learning, such as exemplar clustering [8], influence spread [13], image denoising [5], and feature selection [11], can be formulated as submodular maximization problems.

A submodular function is *nondecreasing* (or *monotone*) if $f(S) \leq f(T) \quad \forall S \subseteq T \subseteq N$, and $f(\emptyset) = 0$. Clearly, maximizing a nondecreasing submodular function is trivial in the absence of further constraints. In this paper we will consider the cardinality constrained version:

$$\begin{aligned} \max f(S) \\ S \subseteq N \\ |S| \leq k \end{aligned}$$

for a given $0 < k < n$. We further assume the standard *value oracle* model, i.e., the submodular function f is known only through a black box oracle that is able to compute the value $f(S)$ for an arbitrary $S \subseteq N$. It is well-known that the simple greedy algorithm [18], which adds at each iteration the element j with the largest marginal benefit w.r.t. the current set until its cardinality is k, achieves an approximation ratio of $(1 - 1/e)$. The greedy algorithm uses $O(nk)$ functions evaluations in the worst case, although this can be improved in practice exploiting definition (2), i.e., the fact that as we add elements to the set, the marginal benefits of the remaining elements to consider can only decrease, see [16]. We will see in the following that the role of the greedy algorithm is quite prominent even in exact methods.

The structure of the paper is as follows. In Section 2 we will describe the MIP model of Nemhauser and Wolsey, which is the basis of all our MIP methods for general submodular function maximization. Then in Section 3 we describe a modern implementation of the basic constraint generation model based on such model, and introduce some algorithmic enhancements to the method, in terms of primal heuristics and cut separation. Computational results are given in Section 4. Finally, conclusions and future directions of research are drawn in Section 5.

2 The general IP model

In [17], Nemhauser and Wolsey introduced a mixed-integer-programming formulation for the general submodular function maximization problem. Let us denote with $\Delta_j(S)$ the marginal value of adding an element j to S, i.e. $\Delta_j(S) = f(S \cup j) - f(S)$. The formulation is based on the following general property of submodular functions:

$$f(T) \leq f(S) + \sum_{j \in T-S} \Delta_j(S) - \sum_{j \in S-T} \Delta_j(S \cup T - j) \quad \forall S, T \subseteq N \tag{3}$$

If f is monotone, (3) can be further simplified to

$$f(T) \leq f(S) + \sum_{j \in T-S} \Delta_j(S) \quad \forall S, T \subseteq N \tag{4}$$

which immediately suggests the MIP formulation:

$$P = \max \quad z \tag{5}$$

$$z \leq f(S) + \sum_{j \notin S} \Delta_j(S) x_j \quad \forall S \subseteq N \tag{6}$$

$$\sum_{j \in N} x_j \leq k \tag{7}$$

$$x_j \in \{0, 1\} \quad \forall j \in N \tag{8}$$

where variables x_j basically encode the indicator function $1(S)$, i.e., $x_j = 1$ iff $j \in S$, and constraints (6) encode (4) as linear inequalities. Note that the model does not exploit any knowledge that cannot be obtained under the value oracle model. Model P has an exponential number of constraints (one for each subset of N, plus the cardinality constraint): thus a constraint generation approach was proposed in [17], where constraints (6) are added iteratively.

Given a solution (x^*, z^*) with x^* integer, it is trivial to solve the separation problem over the family of constraints (6): x^* uniquely defines a subset S^* and we just need to evaluate $f(S^*)$. If $z^* \leq f(S^*)$, then (x^*, z^*) cannot be cut. Otherwise, we just need to construct the cut corresponding to S^* at the cost of additional $n - k$ function evaluations, and this is guaranteed to be violated by (x^*, z^*) by the amount $z^* - f(S^*)$.

It is important to emphasize that formulation P does not give the convex hull of the points $(1(S), f(S))$, so even if we would add all constraints (6), we would not be able to solve the model as a linear program. Surprisingly, constraints (6) can be strengthened by ignoring the fact that the function is nondecreasing, and reverting to the general expression (3). Given that we do not know T in advance, we relax the coefficients $\Delta_j(S \cup T \cup -j)$ to $\Delta_j(N - j)$, and obtain the inequality:

$$z \leq [f(S) - \sum_{j \in S} \Delta_j(N - j)] + \sum_{j \in S} \Delta_j(N - j) x_j + \sum_{j \notin S} \Delta_j(S) x_j \quad \forall S \subseteq N \tag{9}$$

It is easy to show that (9) dominates (6): if $T \supseteq S$ the two expressions coincide, otherwise the right hand side expression of (9) is strictly better. Note that cuts (9) are a little denser than their (6) counterparts. On the other hand, they are not more expensive to compute, as the coefficients $\Delta_j(N - j)$ do not depend on S and can thus be computed once and for all at the beginning. Unfortunately, these cuts are still not enough to obtain a convex hull formulation, as can be easily proven by constructing small counterexamples. Still, we will use cuts (9) in place of (6) in the rest of the paper, as preliminary computational results showed that the resulting formulation gives a stronger dual bound.

3 A modern implementation

The constraint generation method based on model P is somehow reminiscent of Benders decomposition [4]. The model P that is solved iteratively acts a a

Benders master, while constraints (6)—or (9)—are the analogous of Benders optimality cuts. While the analogy is only superficial—in the submodular case there is no variable splitting and there is no (LP) duality theory to exploit to derive a cut—some of the algorithmic improvements proposed for Benders decomposition over the years easily carry over. In particular, we do not need to solve to proven optimality a MIP at each iteration, but we can separate constraints (9) on the fly each time an integer solution is found by the branch-and-cut, exploiting modern MIP solvers support for so-called *lazy constraints*. In other words, only a single enumeration tree is needed.

Then, we can improve the overall method in at least two directions: (i) use a more sophisticated primal heuristic to find a tighter primal bound at the beginning and (ii) separate constraints (9) not only at integer solutions, but also at fractional ones, in order to improve the dual bound more quickly.

3.1 GRASP heuristic

The simple greedy algorithm is known to perform very well in practice, a behaviour that was confirmed in our computational evaluation. Still, it yields the true optimal solution only on the smaller models. A first improvement can be obtained by adding a local search phase at the end of the greedy phase. In order to do so, we need to define a neighborhood structure on the solutions. In the following, we assume to have at hand a subset S of cardinality k. The easiest choice is to consider an *exchange* neighborhood, i.e., consider the set of subsets T that can be obtained by dropping an element from S and adding an element not in S. More formally:

$$\mathcal{N}(S) = \{T \subseteq N : T = S \cup i - j, \forall i \in S, \forall j \notin S\}$$

Each such neighborhood can be explored at the cost of additional $O(nk)$ evaluations. Then the local search phase consists in iteratively exploring the neighborhood of the current set S, and updating it until it is locally optimal.

We can extend the greedy plus local search combination into the full-blown meta-heuristic scheme called GRASP[9]. The main idea is to introduce a randomized component into the greedy procedure, where at each step, instead of picking the element with the largest marginal gain, we pick randomly among the best C (say) candidates, the so-called *restricted candidate list*. Then, each solution found this way is improved by local search, and the process is repeated until some iteration/resource limit is reached.

Interestingly, in our setting we can derive an additional benefit from this most sophisticated heuristic than just an hopefully better primal bound. For each locally optimal solution found by GRASP, we can construct a cut (9), so that we can warm-start the MIP P with an initial pool of cuts.

3.2 Separating fractional solutions

While separating integer solution is trivial, separating fractional solutions is far more challenging, as we cannot directly map the point to cut to a subset of N.

Ideally, we would like to solve, for a given (x^*, z^*), the separation problem:

$$\max \quad z^* - [f(S) - \sum_{j \in S} \Delta_j(N-j)] - \sum_{j \in S} \Delta_j(N-j)x_j^* - \sum_{j \notin S} \Delta_j(S)x_j^*$$
$$S \subseteq N$$
$$|S| \leq k$$

but this is basically as hard as the original problem. As such, we resort to heuristic separation algorithms. In the following, we will describe two such procedures, one based on the greedy algorithm and one based on the Lovász extension of f.

Let F_1 (resp. F_0) be the set of variables fixed to 1 (resp. 0) at the current node of the branch-and-cut tree. In the following, we will denote a node by the pair (F_1, F_0). Clearly, we can modify the greedy algorithm to take those local domains into account. Let S' be the greedy set computed in this way, i.e., $T = F_1 \cup \{x_{j_1}, \ldots, x_{j_p}\}$ for some p, with the variables added by the greedy algorithm exactly in this order. Then for each $0 \leq q \leq p$ we can consider the set $T_q = F_1 \cup \{x_{j_1}, \ldots, x_{j_q}\}$ (the current set after q greedy steps), construct the corresponding cut and check whether it is violated by the current fractional solution. So we test each T_q in sequence, and keep adding elements as long as the violation increases.

There is a nice connection between the cuts that can be obtained this way and the modular heuristic h_{mod} used in A^* search approaches for submodular function maximization. We recall that A^* search picks the next node to explore as the one maximizing $f(F_1)+h(F_1, F_0)$, where $h(\cdot)$ is a so-called *(admissible) heuristic* function that bounds the objective value of any node in the current subtree—in the mathematical programming terminology, any such function would yield a valid dual bound. The admissible heuristic h_{mod}, proposed in [6], is computed as:

$$h_{mod}(F_1, F_0) = \sum_{j \in T - F_1} \Delta_{F_1}(j)$$

where T is the greedy solution computed at node (F_1, F_0). It is easy to see that any fractional solution with $z^* > f(F_1) + h_{mod}(F_1, F_0)$ would be violated by a cut computed by the procedure above. Indeed, the fractional solution is already violated by the cut computed from F_1, and we enlarge the set only if it gives an improvement w.r.t. violation. As a corollary, if the separation procedure above is applied at all nodes of the branch-and-bound tree, we have that the LP relaxation computed at node (F_1, F_0) cannot be worst than $f(F_1)+h_{mod}(F_1, F_0)$, and thus the LP bound dominates the h_{mod} bound, albeit at the cost of solving LPs.

The greedy solution T computed at node (F_1, F_0) is completely oblivious to the fractional solution (x^*, z^*), a fact that can make the discovery of violated inequalities quite rare. A different approach consists in using the so-called Lovász extension $\hat{f}$ of f, i.e., the extension of function f to the unit cube $[0, 1]^n$ [15]. The extension is defined as follows. Let $x^* \in \mathbb{R}^n_+$ be an arbitrary fractional vector in the unit cube. Then we can express x^* as a convex combination of $n + 1$

vertices $x^0, x^1, \ldots, x^n$ of the unit cube, with the additional property that those vertices form a nested sequence, i.e., $x^i \subseteq x^{i+1}$ (with a small abuse of notation, we identify the integer vertices with the subsets of N of which they are the indicator vectors). Given the coefficients $\lambda_0, \ldots, \lambda_n$ of the convex combination, we can then define $\hat{f}(x^*)$ as

$$\hat{f}(x^*) = \sum_{i=0}^{n} \lambda_i f(S_i)$$

where S_i is the set associated to vertex x^i. Instrumentally, an efficient procedure is known to compute, given an arbitrary point x^*, both the sequence $S_0, \ldots, S_n$, and the corresponding multipliers. Intuitively, the sequence is obtained by starting from the empty set, and adding elements according to a permutation of the indices that sorts the values x^*_j in non-increasing order, while the multipliers are obtained as differences of pairs of consecutive coefficients in the sorted sequence, see [15] for more details about the procedure. Once we have computed the sequence (and the corresponding multipliers), we can easily perform two operations:

1. compute $\hat{f}(x^*)$. If $z^* \leq \hat{f}(x^*)$, then we have a proof that no violated cut of the form (9) exists, as (x^*, z^*) belongs to the convex hull of the feasible solutions of P.
2. We can construct a cut for each set S_i in the sequence, and check whether it is violated.

Unfortunately, even this machinery does not give an exact separation procedure, as it can happen that $z^* > \hat{f}(x^*)$, but no cut obtained from the sets in the sequence is violated, and this does not rule out the existence of violated inequalities associated with other subsets of N. The reason is that, intuitively, the Lovász construction gives only one among many possible ways of constructing x^* as a convex combination of vertices of the unit cube.

4 Computational results

We implemented all the methods under comparison in C++, using IBM ILOG CPLEX 12.8 [10] as MIP solver. In the following, we will denote by `base` the basic implementation of the constraint generation method, by `bc` its counterpart where constraints (9) are separated on the fly as lazy constraints, and by `bc+` the improved version that also uses GRASP and separation of fractional solutions in the tree. All codes were run on a cluster of 24 identical machines, each equipped with an Intel Xeon CPU E3-1220 V2 CPU running at 3.10 GHz, and 16 GB of RAM. All codes take full advantage of multi-threading. Each method was run on each instance with a time limit of 1 hour. The parameters used by our code are as follows:

- the GRASP heuristic is run with a limit of 100 iterations. Its restricted candidate list has size $\max(5, n/4)$.

- CPLEX is called with defaults parameters for `base`, as we use it as a pure black box there. On the other hand, for methods `bc` and `bc+` we disable dual and nonlinear reductions, as needed to guarantee correctness because of lazy constraints, and we set the variable selection strategy to *strong branching* [2,3], as this resulted in an improved performance in preliminary tests.
- We separate fractional solutions at the root and at all nodes whose relaxation is within 1% of the best bound node. In addition, the generated cuts are added to the current node only if their violation is at least 0.1% of the best bound. The rationale is that in our case violation is a measure of how much z^* is currently overestimated, hence we can directly compare this value against the objective value. Finally, all separated cuts are added as local cuts—although they would legitimately be globally valid—to ensure a more aggressive purging.

4.1 Benchmark sets

We considered three classes of problems that give rise to submodular functions, namely *location*, *weighted coverage* and *bipartite inference*. We will now briefly describe each of them.

Location [12,19] We are given a ground set of N locations, a set M of clients, and a non-negative profit g_{ij} if client i is served by location j, for all possible pairs. Each client gets the profit from the best opened location, and we want to maximize the overall profit. This corresponds to the submodular function:

$$f(S) = \sum_{i \in M} \max_{j \in S} g_{ij}$$

Weighted Coverage [14,19] We are given a ground set of N sensors, and a set M of possible targets. Each target i has an associated non-negative weight of $w_i \geq 0$. Each sensor j covers the subset of targets $M_j \subseteq M$. We want to maximize the total weight of the covered targets. This corresponds to the submodular function:

$$f(S) = \sum_{i \in \bigcup_{j \in S} M_j} w_i$$

Bipartite Inference[19] We are given a ground set of N items, and a set M of targets. We are also given a bipartite graph $G = (N, M, A)$, where the set of arcs A encodes which targets can be *influenced* by which items. Finally, we get an activation probability p_j for each item j. Given a subset $S \subseteq N$, the graph structure and the activation probabilities p_j, we can compute the activation probability $p_S(i)$ of each target $i \in M$ as:

$$p_S(i) = 1 - \prod_{j \in S : (j,i) \in A} (1 - p_j)$$

We want to maximize the submodular function $f(S) = \sum_{i \in M} p_S(i)$.

We generated random instances for all the classes above, using the following rules:

- $n \in \{20, 50, 100, 200\}$
- $m = \mu n$ with $\mu \in \{2, 5, 10\}$
- $k \in \begin{cases} \{5, 10\} & \text{if } n = 20 \\ \{10, 20\} & \text{if } n \in \{50, 100\} \\ \{20, 50\} & \text{if } n = 200 \end{cases}$

where $n = |N|$ and $m = |M|$. For each combination $(n, m.k)$ we generated 5 random models, obtaining in total 360 instances. For location instances, g_{ij} are randomly picked in the interval $[0, 1]$. For weighted coverage instances, w_i are randomly picked in the interval $[0, 1]$, and a target is covered by a sensor with probability 0.07. Finally, for bipartite inference instances, p_j are again randomly picked in the interval $[0, 1]$, while each arc in the bipartite graph exists with probability 0.07.

4.2 Results

We first compared the three methods `base`, `bc` and `bc+` on the whole testbed of 360 models. Aggregated results are reported in Table 1. The structure of the table is as follows. Instances are divided in different subsets, based on the *difficulty* of the models. To avoid any bias in the analysis, the level of difficulty is defined by taking into account all methods under comparison. The subclasses "[l, 3600}" ($l = 0, 1, 10, 100$), contain the subset of models for which at least one of the methods took at least l seconds to solve and that were solved to optimality within the time limit by at least one of the methods. The subset "all" contains all models. For each subset of models, we report three performance indicators for each of the compared methods: $\#S$ reports the number of instances solved to optimality, $\#T$ the number of instances for which the method hit the time limit, and the shifted geometric mean [1] of the solution time, with a shift of one second. Note that for all methods except the reference method (`base` in our case), we report the ratio w.r.t. to the reference of the runtime (columns "tQ") rather than the value itself. Ratios $t < 1$ indicate a speedup factor of $1/t$.

According to the table, `bc+` is significantly better than `bc`, both in terms of number of instances solved and average runtime, and `bc` is in turn significantly better than the baseline method `base`, again according to both criteria. If we compare `bc+` to `base` directly, we see that `bc+` can solve 62 more models and is on average $4\times$ faster. If we restrict to the set of models that at least one method can solve (193 models), then the speedup is even more impressive, up to $20\times$. In addition, the speedup further increases as we consider harder models.

More detailed results are given in Table 2, where we aggregate only over the 5 different models generated for each parameter combination. For each problem class, and for each combination of (n, m, k), we report, for all methods under comparison, the number of instances solved (out of 5), the shifted geometric mean of runtime, and the final relative gap at the end of the solve—the latter being a significant measure for the case in which we frequently hit the time limit. Note that we also report intermediate aggregate results by problem class.

instances	base #S	base #T	base time (s)	bc #S	bc #T	bc tQ	bc+ #S	bc+ #T	bc+ tQ
all	121	239	349.42	153	207	0.42	193	167	0.24
$[0, 3600\}$	121	72	45.67	153	40	0.19	193	0	0.05
$[1, 3600\}$	39	72	739.86	71	40	0.07	111	0	0.01
$[10, 3600\}$	24	72	1703.62	56	40	0.05	96	0	0.01
$[100, 3600\}$	17	72	2340.09	49	40	0.05	89	0	0.00

Table 1: Aggregated results over whole testbed.

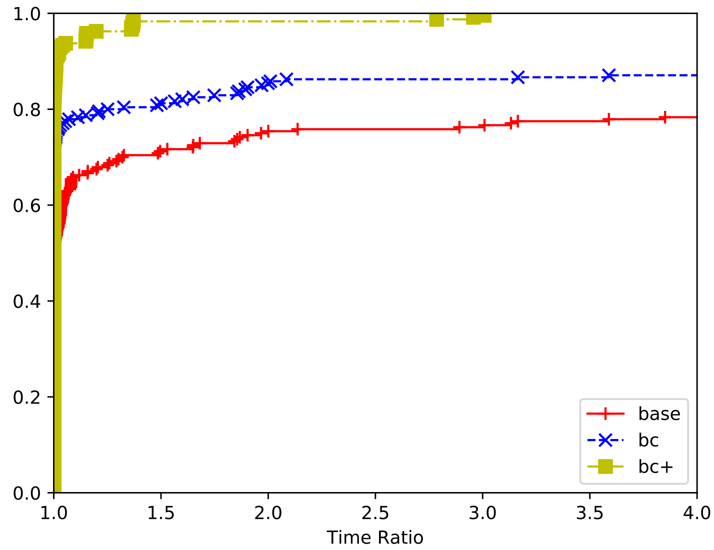

Fig. 1: Performance profile over whole testbed.

According to the table, most parameter combinations either end up being in the easy or unsolvable class, with only a few combinations in between. Still, the more sophisticated methods—and `bc+` in particular—do not exhibit any slowdown on the easy models, while being up to two orders of magnitude faster on the medium models, most of which the `base` cannot even solve. As for the subsets on which all methods hit the time limit, the average final gap is reduced by approximately a factor of 2, which is significant.

In Figure 1 we report the performance profile [7] of the three methods on the whole testbed, which largely confirms the finding of Table 1.

4.3 Comparison with a direct MIP model

Both location and weighted coverage problems can be formulated directly as MIP models. For the location case, the model reads:

$$
\begin{aligned}
\max \sum_{i \in M} \sum_{j \in N} w_{ij} y_{ij} \\
\sum_{j \in N} y_{ij} \leq 1 \quad \forall i \in M \\
y_{ij} \leq x_j \quad \forall i \in M, j \in N \\
\sum_{j \in N} x_j \leq k \\
x_j \in \{0,1\} \quad \forall j \in N \\
y_i j \in \{0,1\} \quad \forall i \in M, j \in N
\end{aligned}
$$

Binary variables x_j encode which locations are open—as in the Nemhauser and Wolsey model—while binary variables y_{ij} encode which location is serving a given client. Note that variables y_{ij} could be relaxed to continuous without affecting the model, as for x fixed the resulting matrix is totally unimodular.

Similarly, for weighted coverage models, the model reads:

$$
\begin{aligned}
\max \sum_{i \in M} w_i y_i \\
\sum_{j : i \in M_j} x_j \geq y_i \quad \forall i \in M \\
\sum_{j \in N} x_j \leq k \\
x_j \in \{0,1\} \quad \forall j \in N \\
y_i \in \{0,1\} \quad \forall i \in M
\end{aligned}
$$

Here, binary variables x_j encode which sensors are deployed, while binary variables y_i encode which targets are covered. Again, variables y_{ij} could be relaxed to continuous without affecting the model.

class	n	m	k	#solved			time(s)			gap		
				base	bc	bc+	base	bc	bc+	base	bc	bc+
loc	20	40	5	5	5	5	1.08	0.05	0.03	0.00%	0.00%	0.00%
			10	5	5	5	0.03	0.01	0.02	0.00%	0.00%	0.00%
		100	5	5	5	5	75.03	0.62	0.17	0.00%	0.01%	0.00%
			10	5	5	5	0.66	0.03	0.06	0.00%	0.01%	0.01%
		200	5	5	5	5	346.59	1.21	0.33	0.00%	0.01%	0.00%
			10	5	5	5	26.36	0.51	0.31	0.00%	0.01%	0.00%
	50	100	10	0	0	5	*t.l.*	*t.l.*	21.22	0.76%	0.34%	0.01%
			20	5	5	5	61.56	2.78	1.20	0.00%	0.01%	0.01%
		250	10	0	0	2	*t.l.*	*t.l.*	3077.35	2.19%	1.93%	0.47%
			20	0	0	3	*t.l.*	*t.l.*	1137.97	0.44%	0.33%	0.05%
		500	10	0	0	0	*t.l.*	*t.l.*	*t.l.*	2.82%	2.74%	1.88%
			20	0	0	0	*t.l.*	*t.l.*	*t.l.*	0.86%	0.73%	0.41%
	100	200	10	0	0	0	*t.l.*	*t.l.*	*t.l.*	3.02%	2.46%	1.56%
			20	0	0	0	*t.l.*	*t.l.*	*t.l.*	1.18%	0.81%	0.49%
		500	10	0	0	0	*t.l.*	*t.l.*	*t.l.*	4.06%	3.92%	3.17%
			20	0	0	0	*t.l.*	*t.l.*	*t.l.*	1.79%	1.58%	1.24%
		1000	10	0	0	0	*t.l.*	*t.l.*	*t.l.*	4.74%	4.74%	4.04%
			20	0	0	0	*t.l.*	*t.l.*	*t.l.*	2.31%	2.10%	1.77%
	200	400	20	0	0	0	*t.l.*	*t.l.*	*t.l.*	2.11%	1.79%	1.41%
			50	0	0	0	*t.l.*	*t.l.*	*t.l.*	0.39%	0.37%	0.23%
		1000	20	0	0	0	*t.l.*	*t.l.*	*t.l.*	2.71%	2.50%	2.09%
			50	0	0	0	*t.l.*	*t.l.*	*t.l.*	0.65%	0.64%	0.48%
		2000	20	0	0	0	*t.l.*	*t.l.*	*t.l.*	3.10%	2.96%	2.61%
			50	0	0	0	*t.l.*	*t.l.*	*t.l.*	0.83%	0.84%	0.69%
				35	35	45	725.32	375.15	269.63	1.42%	1.28%	0.94%
wcov	20	40	5	5	5	5	0.02	0.01	0.01	0.00%	0.00%	0.00%
			10	5	5	5	0.01	0.00	0.01	0.00%	0.00%	0.00%
		100	5	5	5	5	0.47	0.02	0.02	0.00%	0.00%	0.00%
			10	5	5	5	0.05	0.01	0.01	0.00%	0.00%	0.00%
		200	5	5	5	5	3.05	0.06	0.03	0.00%	0.00%	0.00%
			10	5	5	5	0.10	0.01	0.02	0.00%	0.00%	0.00%
	50	100	10	3	5	5	1442.22	27.44	0.67	0.56%	0.01%	0.01%
			20	5	5	5	0.87	0.23	0.07	0.00%	0.01%	0.00%
		250	10	0	0	5	*t.l.*	*t.l.*	36.44	6.06%	3.48%	0.01%
			20	3	5	5	837.59	39.35	1.88	0.31%	0.01%	0.00%
		500	10	0	0	1	*t.l.*	*t.l.*	2787.92	13.20%	11.08%	3.13%
			20	0	0	3	*t.l.*	*t.l.*	1503.03	4.48%	2.94%	0.65%
	100	200	10	0	0	0	*t.l.*	*t.l.*	*t.l.*	13.26%	10.25%	4.83%
			20	0	0	5	*t.l.*	*t.l.*	937.31	4.89%	1.51%	0.01%
		500	10	0	0	0	*t.l.*	*t.l.*	*t.l.*	20.36%	17.70%	10.03%
			20	0	0	0	*t.l.*	*t.l.*	*t.l.*	11.72%	9.64%	6.69%
		1000	10	0	0	0	*t.l.*	*t.l.*	*t.l.*	26.59%	23.34%	13.37%
			20	0	0	0	*t.l.*	*t.l.*	*t.l.*	17.37%	14.85%	11.97%
	200	400	20	0	0	0	*t.l.*	*t.l.*	*t.l.*	11.01%	8.80%	5.16%
			50	5	5	5	0.04	0.01	0.16	0.00%	0.00%	0.00%
		1000	20	0	0	0	*t.l.*	*t.l.*	*t.l.*	18.75%	16.66%	13.39%
			50	5	5	5	0.10	0.02	0.39	0.00%	0.00%	0.00%
		2000	20	0	0	0	*t.l.*	*t.l.*	*t.l.*	24.00%	21.96%	19.53%
			50	2	3	5	145.45	26.04	2.06	0.05%	0.05%	0.00%
				53	58	74	147.31	92.69	50.40	7.19%	5.93%	3.70%
binf	20	40	5	5	5	5	0.01	0.01	0.01	0.00%	0.00%	0.00%
			10	5	5	5	0.01	0.00	0.01	0.00%	0.00%	0.00%
		100	5	5	5	5	0.02	0.01	0.01	0.00%	0.00%	0.00%
			10	5	5	5	0.00	0.00	0.01	0.00%	0.00%	0.00%
		200	5	4	5	5	4.47	0.01	0.01	0.00%	0.00%	0.00%
			10	5	5	5	0.01	0.00	0.01	0.00%	0.00%	0.00%
	50	100	10	2	5	5	318.98	0.27	0.05	2.21%	0.01%	0.00%
			20	1	5	5	753.23	0.08	0.06	0.32%	0.01%	0.00%
		250	10	0	5	5	*t.l.*	17.41	0.23	1.35%	0.01%	0.01%
			20	0	5	5	*t.l.*	1.91	0.21	5.41%	0.01%	0.00%
		500	10	0	5	5	*t.l.*	88.92	0.82	2.59%	0.01%	0.01%
			20	1	5	5	886.66	7.15	0.69	0.33%	0.01%	0.00%
	100	200	10	0	0	5	*t.l.*	*t.l.*	14.12	8.62%	5.26%	0.01%
			20	0	0	2	*t.l.*	*t.l.*	2546.50	6.00%	4.15%	1.15%
		500	10	0	0	3	*t.l.*	*t.l.*	517.44	16.68%	11.26%	1.41%
			20	0	0	0	*t.l.*	*t.l.*	*t.l.*	11.52%	8.76%	6.25%
		1000	10	0	0	4	*t.l.*	*t.l.*	837.36	16.31%	11.51%	0.92%
			20	0	0	0	*t.l.*	*t.l.*	*t.l.*	12.41%	9.25%	7.01%
	200	400	20	0	0	0	*t.l.*	*t.l.*	*t.l.*	16.78%	12.95%	10.87%
			50	0	0	0	*t.l.*	*t.l.*	*t.l.*	2.84%	2.65%	2.20%
		1000	20	0	0	0	*t.l.*	*t.l.*	*t.l.*	21.51%	17.90%	15.67%
			50	0	0	0	*t.l.*	*t.l.*	*t.l.*	3.93%	3.81%	3.26%
		2000	20	0	0	0	*t.l.*	*t.l.*	*t.l.*	23.63%	20.07%	17.75%
			50	0	0	0	*t.l.*	*t.l.*	*t.l.*	4.76%	4.63%	4.16%
				33	60	74	398.46	93.58	42.83	6.55%	4.68%	2.94%
all				121	153	193	349.42	148.37	83.79	5.05%	3.96%	2.53%

Table 2: Detailed results over all subsets of models.

It is thus interesting to compare our specialized methods against a direct application of a black box MIP solver, which of course requires much less effort. We present the comparison for the subset of locations models—similar results can be obtained for the class of weighted coverage models—and we compare the best of our specialized methods, `bc+`, against CPLEX defaults (`CPLEX`) and the automatic Benders decomposition of CPLEX (`Benders`)—Benders being a viable option assuming variables y_{ij} are relaxed to continuous. Aggregated results are given in Table 3, whose structure is identical to Table 1, and the corresponding performance profile is given in Figure 2. The comparison allows us to put the performance of `bc+` into perspective: while significantly faster than what we started from (`base`), it is still no match for a black box MIP solver like CPLEX, which is able to solve 50% more models, and is overall twice as fast (and up to $10\times$ faster as the models get harder). The automatic Benders decomposition is even faster, solving additional models and being quite faster than `CPLEX`.

In hindsight, this is not unexpected: being able to express the full model as a mixed integer program allows for a lot of sophisticated techniques to be employed that can take advantage of a global view on the problem. Unsurprisingly, encoding the strcuture of f directly into the model results in being a better option than writing a model that can access f only through a black box oracle. On the other hand, a direct MIP formulation is not always a viable option, e.g. in the bipartite inference case, and this justifies the effort to make the general submodular framework more efficient. In addition, there is clearly still room for further research and improvements.

5 Conclusions and future research

In the present paper, we presented a modern implementation of the MIP model of Nemhauser and Wolsey for submodular function maximization, based on lazy constraint generation. We also developed some algorithmic improvements, namely a GRASP heuristic and two (heuristic) procedures for separating submodular cuts from fractional solutions. A computational evaluation on three classes of submodular functions showed that the developed methods significantly improve over the basic constraint generation model by the book. A comparison with a direct MIP formulation for one class of models also showed that, when available, this is usually a preferable option, being not only far easier to implement but also quite faster than the MIP-based framework based on the general model. Still, the general framework is in its computational infancy, and further research is needed in many areas. Among others, more effective (and possibly exact) separation procedures over the family of cuts (9)—or even (6)—and custom branching rules based on f could make the method more efficient in practice.

References

1. Tobias Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, 2007.

instances	bc+ #S	bc+ #T	bc+ time (s)	CPLEX #S	CPLEX #T	CPLEX tQ	Benders #S	Benders #T	Benders tQ
all	45	75	269.63	66	54	0.48	73	47	0.39
$[0, 3600\}$	45	28	50.13	66	7	0.28	73	0	0.20
$[1, 3600\}$	16	28	623.30	37	7	0.13	44	0	0.07
$[10, 3600\}$	10	28	1551.18	31	7	0.10	38	0	0.05
$[100, 3600\}$	5	28	2952.66	26	7	0.10	33	0	0.05

Table 3: Aggregated results on location models, comparing `bc+` to CPLEX defaults and CPLEX automatic Benders.

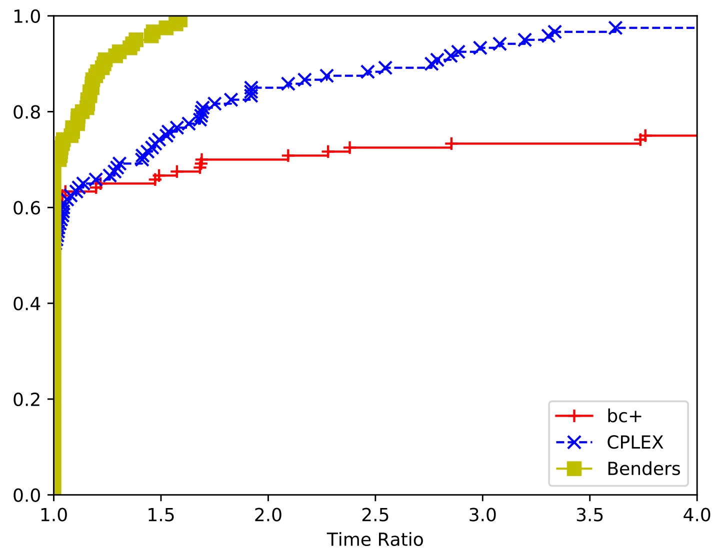

Fig. 2: Performance profile on location models, comparing `bc+` to CPLEX defaults and CPLEX automatic Benders.

2. David L. Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook. Finding cuts in the TSP (A preliminary report). Technical Report 95-05, DIMACS, 1995.
3. David L. Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, USA, 2007.
4. J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.
5. A. Chambolle. Total variation minimization and a class of binary MRF models. In *EMMCVPR*, pages 136–152, 2005.
6. Wenlin Chen, Yixin Chen, and Kilian Q. Weinberger. Filtered search for submodular maximization with controllable approximation bounds. In *AISTATS*, volume 38, pages 156–164, 2015.
7. E. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002.
8. Delbert Dueck and Brendan J. Frey. Non-metric affinity propagation for unsupervised image categorization. In *ICCV*, pages 1–8. IEEE Computer Society, 2007.
9. Thomas A. Feo and Mauricio G. C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989.
10. IBM. ILOG CPLEX 12.8 User's Manual, 2018.
11. A. Jović, K. Brkić, and N. Bogunović. A review of feature selection methods with applications. In *MIPRO*, pages 1200–1205, 2015.
12. Yoshinobu Kawahara, Kiyohito Nagano, Koji Tsuda, and Jeff A. Bilmes. Submodularity cuts and applications. In Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. I. Williams, and Aron Culotta, editors, *NIPS*, pages 916–924, 2009.
13. David Kempe, Jon Kleinberg, and Eva Tardos. Maximizing the spread of influence in a social network. *KDD*, pages 137–146, 2003.
14. Andreas Krause and Daniel Golovin. *Submodular Function Maximization*, pages 71–104. Cambridge University Press, 2014.
15. László Lovász. Submodular functions and convexity. In Achim Bachem, Martin Grötschel, and Bernhard Korte, editors, *Mathematical Programming — The State of the Art*, pages 235–257. Springer, 1983.
16. Michel Minoux. Accelerated greedy algorithms for maximizing submodular set functions. In *Optimization Techniques*, pages 234–243, 1978.
17. George L. Nemhauser and Laurence A. Wolsey. Maximizing submodular set functions: Formulations and analysis of algorithms. In P. Hansen, editor, *Studies on Graphs and Discrete Programming*, pages 279–301, 1981.
18. George L. Nemhauser, Laurence A. Wolsey, and Marshall L. Fisher. An analysis of approximations for maximizing submodular set functions - I. *Mathematical Programming*, 14(1):265–294, 1978.
19. Shinsaku Sakaue and Masakazu Ishihata. Accelerated best-first search with upper-bound computation for submodular function maximization. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *AAAI*, pages 1413–1421, 2018.