



University of Salerno
DEPARTMENT OF MATHEMATICS

A Hybrid Exact Approach for Maximizing Lifetime in Sensor Networks with Complete and Partial Coverage Constraints

Francesco Carrabs

Department of Mathematics, University of Salerno. E-mail: fcarrabs@unisa.it

Raffaele Cerulli

Department of Mathematics, University of Salerno. E-mail: raffaele@unisa.it

Ciriaco D'Ambrosio

Department of Computer Science, University of Salerno. E-mail: cdambrosio@unisa.it

Andrea Raiconi

Department of Mathematics, University of Salerno. E-mail: araiconi@unisa.it

Technical Report N° 50147
27/8/2015

A Hybrid Exact Approach for Maximizing Lifetime in Sensor Networks with Complete and Partial Coverage Constraints

Francesco Carrabs · Raffaele Cerulli ·
Ciriaco D'Ambrosio · Andrea Raiconi

Received: date / Accepted: date

Abstract In this paper we face the problem of maximizing the amount of time over which a set of target points, located in a given geographic region, can be monitored by means of a wireless sensor network. The problem is well known in the literature as Maximum Network Lifetime Problem (MLP). In the last few years the problem and a number of variants have been tackled with success by means of different resolution approaches, including exact approaches based on column generation techniques. In this work we propose an exact approach which combines a column generation approach with a genetic algorithm aimed at solving efficiently its separation problem. The genetic algorithm is specifically aimed at the Maximum Network α -Lifetime Problem (α -MLP), a variant of MLP in which a given fraction of targets is allowed to be left uncovered at all times; however, since α -MLP is a generalization of MLP, it can be used to solve the classical problem as well. The computational results, obtained on the benchmark instances, show that our approach overcomes the algorithms, available in literature, to solve both MLP and α -MLP.

Keywords Maximum Lifetime · Wireless Sensor Network · Column Generation · Genetic Algorithm

Francesco Carrabs
Department of Mathematics, University of Salerno.
E-mail: fcarrabs@unisa.it

Raffaele Cerulli
Department of Mathematics, University of Salerno.
E-mail: raffaele@unisa.it

Ciriaco D'Ambrosio
Department of Computer Science, University of Salerno.
E-mail: cdambrosio@unisa.it

Andrea Raiconi
Department of Mathematics, University of Salerno.
E-mail: araiconi@unisa.it

1 Introduction

Wireless Sensor Networks (WSNs) are usually composed of a large amount of sensing devices (*sensors*) scattered over a region of interest. Each sensor is generally capable of monitoring a certain portion of the space around itself (usually called its *sensing area*, and defined by the *sensing range* of the sensor). While each individual device has obvious limits in terms of range extension and battery lifetime, a coordinated use of multiple sensors together allows to perform complex monitoring activities in possibly large areas, in fields as diverse as environmental control, military and health care applications, among others (see, for example, [1], [14], [16]).

Given the limited power of the batteries that usually keep sensing devices operational, an issue which has generated intense research interest in the last years is related to the optimization of battery consumption. In particular, the problem of appropriately use sensors to monitor a set of specific points of interests located inside the area (known as *targets*) for as long as possible has been widely studied; the problem is usually known as Maximum Network Lifetime Problem (MLP). It has been mainly approached with methods aimed at finding several, possibly overlapping sets of sensors (defined *covers*) which can individually provide coverage for all the targets, as well as an activation time for each of them, such that the sum of the activation times of the covers in which each sensor appears is not greater than the lifetime provided by its battery. The idea is then to activate the covers one by one, where by activating a cover we intend to turn on all the sensors which belong to it, while keeping all other sensors turned off.

In [4] the authors showed that MLP can bring improvements with respect to previous approaches in which sensors were divided into disjoint sets (that is, each sensor could only belong to a single cover). They also proved the problem to be NP-Complete and they proposed an approximation algorithm to solve it.

A Column Generation approach aimed at solving the MLP was proposed in [12]. In this work the authors propose a hybrid approach where the separation problem of the Column Generation technique is either solved heuristically or optimally by means of an appropriate ILP formulation. More details about this technique are given in Section 3. For a survey on hybrid algorithms, including the embedding of heuristics and metaheuristics into Column Generation frameworks, the reader may refer to [3].

Several variants of MLP have been proposed as well, in order to adapt the problem to different contexts. Some of the proposed variants take into account cover connectivity ([2], [7], [8], [15], [19]) or reliability issues ([10]), or consider sensors with adjustable sensing ranges ([5], [9], [17]). For many of these variants, efficient algorithms based on Column Generation have been proposed ([2], [6], [7], [8], [9], [10], [15], [17], [18]).

Another interesting variant of the problem is the Maximum Network α -Lifetime Problem (α -MLP), which was proposed and studied in [13]. In such a variant, a predefined portion of the overall number of the targets is allowed to

be neglected in each cover. As will be better investigated in Section 2, α -MLP generalizes MLP and therefore each method aimed at solving this problem can also be used to face the original one. In [13] the authors presented both a heuristic algorithm and an exact one, showing that large improvements in terms of overall network lifetime can generally already be achieved by neglecting a small percentage of targets in each cover. Furthermore, the authors also showed that most of the advantage is usually retained when some additional regularity conditions are taken into account in order to guarantee a minimum global coverage level to each target.

In this work we propose an hybrid exact approach for the α -MLP problem, named GCG. While the overall structure of the algorithm is again based on the Column Generation technique, the main contribution of this work consists in the proposal of an appropriately designed genetic metaheuristic which is used to solve its separation problem. As will be shown in the discussion of our computational tests our algorithm is proven to be highly efficient in terms of requested computational time with respect to both the algorithms presented in [12] for MLP and in [13] for α -MLP.

The rest of the work is organized as follows. Section 2 formally introduces the problems and a mathematical formulation to describe them. Section 3 resumes the approaches presented in [12] and [13] to solve MLP and α -MLP. Section 4 describes our proposed genetic algorithm, while Section 5 presents the results of our computational experiments. Finally, Section 6 presents some final remarks.

2 Problems Definition and Mathematical Formulation

Let $N = (T, S)$ be a wireless sensor network, with $T = \{t_1, \dots, t_n\}$ being the set of the targets and $S = \{s_1, \dots, s_m\}$ being the set of sensors. As previously introduced, each sensor is assumed to have a given sensing range and to be powered by a battery that can keep it activated for a limited amount of time. In this paper we assume each sensor to be identical, therefore each of them has a sensing range of the same size, and all battery durations are normalized to 1. In Figure 1(a) a sensor network with sensors s_1, \dots, s_6 , targets t_1, \dots, t_6 and sensing ranges represented by circles is shown.

For each target $t_k \in T$ and sensor $s_i \in S$, let δ_{ki} be a binary parameter equal to 1 if t_k is positioned within the sensing range of s_i (it is *covered* by the sensor), 0 otherwise. For a subset of sensors $S' \subseteq S$ and $t_k \in T$, let $\Delta_{kS'}$ be another binary parameter equal to 1 if $\delta_{ki} = 1$ for a given $s_i \in S'$, 0 otherwise.

Given a value $\alpha \in (0, 1]$, we define $C \subseteq S$ to be a *feasible cover* (or simply a cover) for the network if its sensors cover at least $T_\alpha = \alpha \times n$ targets, that is, $\sum_{t_k \in T} \Delta_{kC} \geq T_\alpha$. Furthermore, we define a cover to be *non-redundant* if it does not contain another cover as a proper subset.

The Maximum Network α -Lifetime Problem (α -MLP) consists then in finding a collection of pairs (C_j, w_j) where each $C_j \subseteq S$ is a feasible cover and each $w_j \geq 0$ is an activation time, such that the sum of the activation times

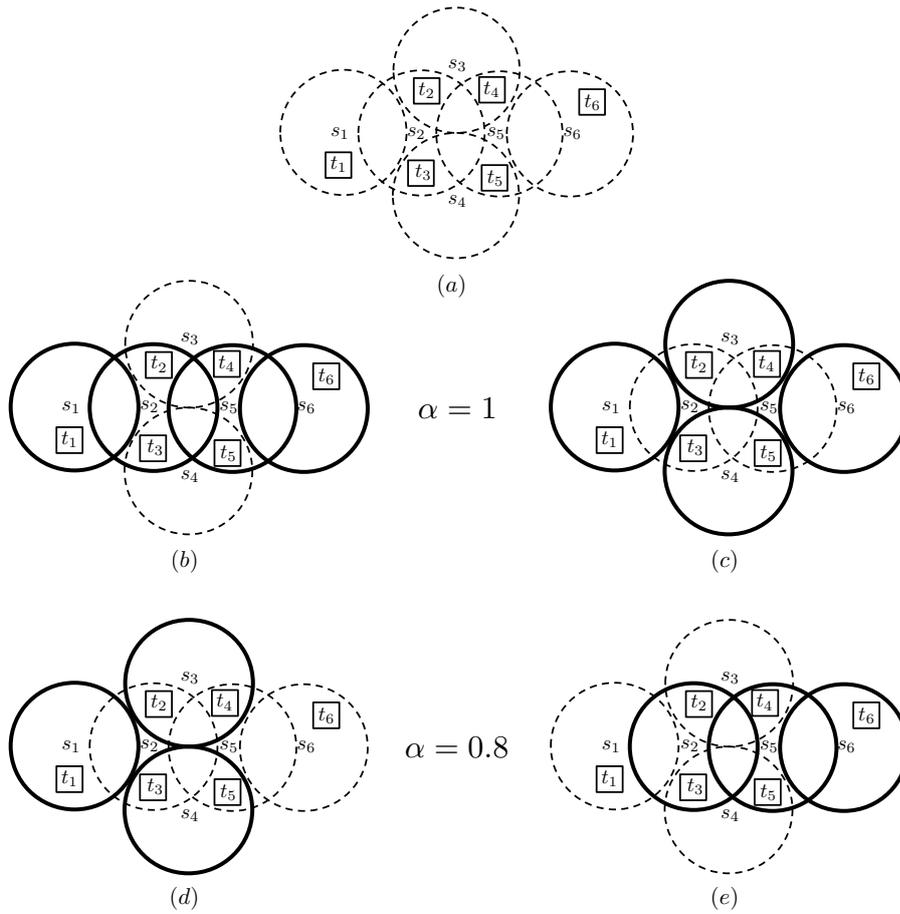


Fig. 1 Example network

is maximized and each sensor is used for an amount of time that does not exceed its normalized battery duration. It is easy to understand that an optimal solution can always be found by only considering non-redundant covers.

Assuming to be able to compute the whole set of feasible covers C_1, \dots, C_ℓ in advance, α -MLP could then be represented using the following Linear Programming formulation, where the binary parameter $a_{ij} = 1$ if $s_i \in C_j$, 0 otherwise:

$$[\mathbf{P}] \max \sum_{j=1}^{\ell} w_j \quad (1)$$

s.t.

$$\sum_{j=1}^{\ell} a_{ij} w_j \leq 1 \quad \forall i = 1, \dots, m \quad (2)$$

$$w_j \geq 0 \quad \forall j = 1, \dots, \ell \quad (3)$$

Objective function (1) maximizes the sum of the activation times, while constraints (2) enforce the respect of the lifetime constraints for each sensor.

In the classical Maximum Network Lifetime Problem (MLP), each cover has to provide information on the whole set of targets in order to be feasible; therefore, MLP corresponds to the α -MLP with $\alpha = 1$ (and hence $T_\alpha = n$). Under these assumptions, the problem definition and the **[P]** formulation presented above represent the classical problem as well.

It is interesting to observe that, on the same wireless sensor network, the maximum lifetime for the α -MLP is often greater than the maximum lifetime for the MLP. For instance, let us consider again the network in Figure 1(a). It is easy to see that the only two non-redundant feasible covers for MLP would be $\{s_1, s_2, s_5, s_6\}$ (Figure 1(b)) and $\{s_1, s_3, s_4, s_6\}$ (Figure 1(c)). In this case, it is possible to obtain a network lifetime equal to 1 time unit by activating them for any couple of activation times $w_1, w_2 \geq 0$ such that $w_1 + w_2 = 1$. However, after this operation, the batteries of sensors s_1 and s_6 are exhausted, and no more feasible covers can be obtained by using the remaining sensors, therefore the final solution is equal to 1 as well. Let us consider now on the same network an α -MLP problem with $\alpha = 0.8$, that is 1 out of 6 targets can be neglected. In this case there are four non-redundant covers $\{s_1, s_3, s_4\}$ (Figure 1(d)), $\{s_2, s_5, s_6\}$ (Figure 1(e)), $\{s_1, s_2, s_5\}$ and $\{s_3, s_4, s_6\}$, and we can easily obtain a lifetime equal to 2 time unit by activating in sequence the covers $\{s_1, s_3, s_4\}$ and $\{s_2, s_5, s_6\}$, for 1 time unit.

On instances of real-world size, formulation **[P]** can not be expected to be directly applicable due to the high (potentially exponential) number of covers. This can be especially true for lower values of α ; indeed, it is straightforward to observe that given $(\alpha_1, \alpha_2) \in (0, 1]^2$ with $\alpha_2 < \alpha_1$, each cover for α_1 -MLP is also feasible for α_2 -MLP. For this reason, it is necessary to apply different approaches, such as Column Generation which was proposed by [12] for MLP and by [13] for α -MLP. We use the same type of approach in this work, however we focus our attention on solving efficiently the subproblem, since it is a key component to obtain an effective algorithm. To this end, we design a fast genetic metaheuristic whose main characteristic is the ability to return several good covers at once and, as we will see in Section 5, this feature is able to bring significant improvements in terms of computational time, with respect to the previous algorithms.

3 Column Generation Approaches for α -MLP and MLP

Given a Linear Programming formulation with a large number of variables (in our case, formulation **[P]**), the Column Generation (CG) technique starts by considering a version of the formulation which only uses a subset of those

variables (in our case, a subset of feasible covers) in the so-called *Master Problem*, and by solving it to optimality. The optimal solution of the Master Problem is clearly a feasible solution for $[\mathbf{P}]$. The CG then considers a specific optimization problem (defined *separation problem* or *subproblem*) which either produces an attractive cover to be added for a new iteration of the Master Problem, or certifies that the last solution found by it (that we denote as *incumbent solution* from now on) is indeed optimal for $[\mathbf{P}]$. The procedure iterates until the above described optimality condition is met. In this way, the CG approach allows to implicitly discard most of the variables that will be nonbasic in the optimal solution.

An *attractive cover* is a feasible cover corresponding to a nonbasic variable with a negative reduced cost, which could therefore improve the incumbent solution if introduced in the master problem. Conversely, such a solution can not be improved if the reduced cost associated with the nonbasic variables are all non negative. More in detail, given the dual prices π_i associated to each constraint of the Master Problem, that is, to each sensor, the incumbent solution is optimal if $\sum_{i:s_i \in C_j} \pi_i - c_j \geq 0$ for each nonbasic cover C_j , which can be rewritten as $\sum_{i:s_i \in C_j} \pi_i \geq 1$ since the coefficients in the objective function (1) of the original LP formulation are all equal to 1.

We can therefore define as subproblem the following formulation $[\mathbf{SP}]$, where objective function (4) minimizes the sum of the dual prices in the sensors chosen to be part of the newly produced cover, while constraints (5)-(8) define a feasible cover:

$$[\mathbf{SP}] \quad \min \sum_{i=1}^m \pi_i x_i \quad (4)$$

s.t.

$$\sum_{i=1}^m \delta_{ki} x_i \geq y_k \quad \forall k = 1, \dots, n \quad (5)$$

$$\sum_{k=1}^n y_k \geq T_\alpha \quad (6)$$

$$x_i \in \{0, 1\} \quad \forall i = 1, \dots, m \quad (7)$$

$$y_k \in \{0, 1\} \quad \forall k = 1, \dots, n \quad (8)$$

For each sensor s_i , the binary variable x_i represents the choice on including it in the new cover, while, for each target t_k , the variable y_k represents whether the target is monitored in the cover. Constraints (5) make sure that each y_k can have value 1 only if at least one of the sensors that cover the target has been added, while constraints (6) impose that at least T_α targets are covered. The incumbent solution is then optimal if the value of objective function (4) is greater or equal than 1, otherwise the new attractive cover is added to the master problem.

When $\alpha = 1$, that is we are considering the MLP problem, constraints (5) reduce to $\sum_{i=1}^m \delta_{ki} x_i \geq 1 \forall k = 1, \dots, n$, and constraints (6) as well as variables y_k are not necessary.

3.1 Heuristics to enhance CG

The main disadvantage of the column generation approach presented above is that **[SP]** is strongly NP-hard, being a specialization of the Set Covering problem. For this reason, it is advisable to limit as much as possible the number of times in which it is required to be solved. In [12] the author faces the problem by introducing a constructive heuristic to quickly solve the subproblem. This heuristic iteratively builds a cover by first choosing, in random way, an uncovered target and then by selecting the sensor that can cover it, with the minimal dual price value. This process is repeated until a complete coverage has been obtained. The author introduces three column generation approaches named Exact, Heur and Mixed, respectively. The first algorithm solves the subproblem in an exact way, while the second one solves the subproblem by invoking the above described constructive heuristic. When the heuristic does not find attractive covers, Heur stops without certifying the optimality of the incumbent solution. For this reason, the solutions provided by Heur can be suboptimal. Finally, in the Mixed algorithm the attractive covers are provided by constructive heuristic and, when it fails, by solving the exact subproblem, which is also used to prove the optimality of the solution in the last iteration of the algorithm.

In [13], the authors propose instead a heuristic meant to independently produce a complete solution for α -MLP (that is, a collection of covers and activation times). Each cover in this approach is again built iteratively, adopting some heuristic criteria to favor the coverage of sensors which has been covered for fewer amounts of time so far in the partial solution. Each newly produced cover is assigned a predefined amount of time, and the algorithm ends when the residual energy in the sensors do not allow to produce a new feasible one. Finally, the set of produced covers is used as initial restricted set for the master problem.

In this work, we attempt to heuristically solve **[SP]** at each iteration, by using a genetic metaheuristic instead of a simple constructive heuristic as the one proposed in [12]. As in the Mixed algorithm, the exact subproblem formulation is used when the genetic algorithm fails in order to guarantee that an exact solution is always found. We define this hybrid exact approach GCG. As we show in the following sections, our column generation approach is able to significantly outperform the previous algorithms for MLP and α -MLP proposed in literature.

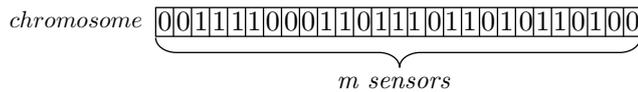


Fig. 2 The chromosome representation.

4 A Genetic Algorithm to address the [SP] Subproblem

A genetic algorithm is a naturally randomized technique that emulates the typical steps of the biological evolution based on the concept of *natural selection*, *crossover* and *mutation*. Each problem solution is expressed by an element, named *chromosome*, that represents the structure of an individual. Given a starting population P of chromosomes, the genetic algorithm iteratively produces new chromosomes by means of the crossover operator which combines, in a probabilistic manner, the genetic information of typically two or more randomly selected elements of the population. On each newly generated chromosome, a mutation operator is applied in order to provide a perturbation of the solution to add diversity. The natural selection process and the *fitness* function, which is used to rank each solution, aim at introducing in the population new chromosomes which are better adapted to the environment. Genetic algorithms take typically into account stop conditions, which could be for instance a maximum number of iterations, a time limit, a lack of improvements in the fitness function of the best individual for a given number of iteration, or a combination of some of the above. For a complete and detailed description of genetic algorithms and their characteristics, the reader can refer to [11].

In order to overcome the hardness of the [SP] problem, we decided to solve the subproblem heuristically through the design of a specific genetic algorithm, defined GA from now on.

The aim of GA is to quickly find attractive covers, and return them to the master problem. An interesting feature of our approach is its ability to potentially produce several attractive covers at once, reducing dramatically the number of required iterations. If GA fails in finding any attractive cover, GCG solves the [SP] formulation instead, in order to either find a new attractive cover or prove the optimality of the current solution. It follows that the greater is the effectiveness of GA, the better are the performances of the whole GCG framework. As will be shown in Section 5, GA appears to be very effective since on the consider set of benchmark instances it often fails only once, i.e. when the optimal solution is found.

Sections 4.1-4.5 describe in detail the different components of GA, while Section 4.6 presents a general overview of the procedure.

4.1 Chromosome Representation and Fitness Function

In GA, the binary vector representation shown in Figure 2 is used for the chromosomes. Each chromosome contains $m = |S|$ positions, which are associ-

ated to the sensors of the network. A chromosome represents a feasible cover, meaning that each position i ($i = 1, \dots, m$) is equal to 1 if the related sensor s_i belongs to the cover (in which case the sensor is said to be *active*), and 0 otherwise. It can be observed that the value of position i corresponds in GA to the value assigned to binary variable x_i in the [SP] formulation. Analogously to covers, a chromosome is defined to be redundant if it is possible to switch off at least one of its active sensors, and the related cover remains feasible. Since, as already mentioned, an optimal solution can always be found by only considering non-redundant feasible covers, during the GA execution we only allow non-redundant chromosomes to be part of the population.

The fitness function for a given chromosome is equal to the dot product of the binary chromosome vector and the dual prices vector deriving from the last iteration of the Master Problem (and therefore corresponds to objective function (4) for [SP]). At the end of the GA procedure, each chromosome with a fitness lower than 1 will be included in the Master Problem as a new column.

4.2 Crossover

One of the main aspects that influence the effectiveness of a genetic algorithm is the crossover operator. This operator allows the creation of new chromosomes starting from previous individuals in the population. In particular, the crossover usually selects two chromosomes of the population (defined *parents*), and generates a new one starting from them (the *child*), which hopefully inherits good features from them. During the evolution process of a genetic algorithm, special care should be taken in order to avoid the case in which several identical chromosomes exist in the population; indeed, in that case the crossover operator has failed to create offspring that is different from their parents. This situation penalizes the effectiveness of the algorithm and therefore the quality of the final solutions.

In our crossover, the selection of the parents is carried out through a typical binary tournament. To this end, the chromosomes of the population are sorted, in ascending order, according to their fitness values. Subsequently, two chromosomes are selected randomly, and the one with the best fitness function among them is chosen as first parent. The second parent is chosen in the same way, avoiding the first parent to be chosen among the participants of the second tournament.

Our crossover operator works exactly like the bitwise AND logical operator. Figure 3 shows, on the left, the AND truth table and, on the right, two sample parent chromosomes *parent1* and *parent2*, as well as the building process of the *child* chromosome starting from them. This type of operator is meant to bring to the new child the genetic heritage which is common to the two parents.

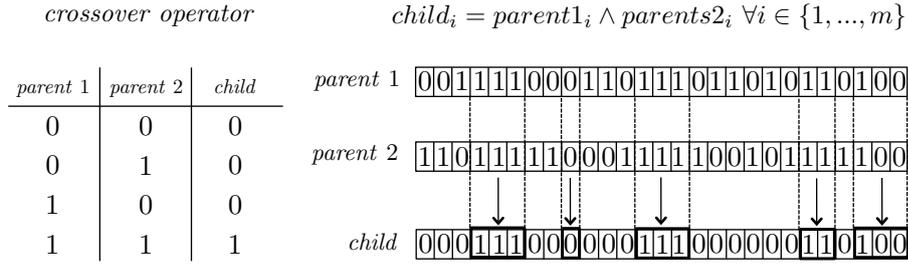


Fig. 3 *The crossover operator.*

4.3 Mutation

The mutation operator alters one or more genes in a chromosome in order to introduce some perturbation, and thus providing diversification in the new generated chromosomes.

As previously introduced, in our genetic algorithm no duplicated chromosomes are allowed in the population. This means that building a duplicate chromosome is a waste of computational time since it would be rejected. Since we do not build new chromosomes by taking into account all the previous ones in the population, we try to differentiate each child from at least both its parents, if possible. If two parent chromosomes have mostly identical genes, which could be a common case especially towards the end of the procedure, the child will be very similar to them as well, and therefore it is not uncommon that the final operations carried out on it in order to guarantee feasibility and remove redundancy (see Section 4.4) could make it exactly identical to one of its parents. In order to face this problem, we use mutation to change the value of one of a random single gene in the child whose value is identical into its parents, if it exists, in order to differentiate it from both of them. This gene will be switched back only if strictly needed by the feasibility or redundancy operator described in the next section.

4.4 Feasibility and Redundancy Operators

It is easy to see that the chromosome produced by the crossover and mutation operators could be not feasible, since it is not guaranteed that the T_α coverage is satisfied. For this reason, it is necessary to apply another operator, which we call *feasibility* operator, in order to restore feasibility. To this end, the feasibility operator selects randomly one of the genes in the child with a value equal to zero whose related sensor could cover some new targets, and switches its value to one (thus activating the sensor). This process is repeated until the threshold T_α is satisfied. Algorithm 1 shows the pseudocode of this operator. The while loop of line 1 is repeated until the threshold is reached. The procedure individuates the set of uncovered targets \hat{T} (line 2), and randomly selects one of them, t (line 3). Then it randomly selects and activates a

sensor s which can cover t (line 4). Finally, in the last two lines the operator updates the child chromosome and the related set of covered targets.

Algorithm 1: feasibilityOperator

Input: unfeasible *Child* chromosome;
Output: feasible *Child* chromosome;

```

1 while  $T_{covered} < T_\alpha$  do
2    $\hat{T} \leftarrow T \setminus T_{covered}$ ;
3    $t \leftarrow randomSelect(\hat{T})$ ;
4    $s \leftarrow randomSelect(S, t)$ ;
5    $update(Child, s)$ ;
6    $update(T_{covered})$ ;
7 return Child;
```

The application of the *feasibility* operator can produce a redundant chromosome. Therefore we apply a final operator called *redundancy*. The operator checks whether each sensor in the chromosome could be switched off without compromising feasibility, thus producing a list of redundant sensors. If the list is not empty, it switches off a randomly chosen element of it. The chromosome is updated, the list is rebuilt and the procedure iterates until the list is equal to the empty set.

4.5 Building the Initial Population

Each individual of the initial population P is randomly built by applying in sequence the *feasibility* and *redundancy* operator starting from a chromosome whose positions are all set to zero.

As soon as a feasible chromosome is obtained, it is added to the population if it is not already present in it and is rejected otherwise, until a fixed desired number $Size_P$ of different chromosomes is obtained. In order to avoid the procedure to iterate indefinitely, a *maxinitDB* threshold is taken into account. If the number of rejected chromosomes reaches the threshold, the procedure stops and $Size_P$ is updated to be equal to the current value of $|P|$.

4.6 GA Structure and GCG initialization

This section describes the overall structure of GA. The pseudocode is listed in Algorithm 2. The input consists of a wireless sensor network (S, T) , where S is the set of sensors and T is the set of targets, as well as a vector of dual prices DP coming from the last iteration of the current Restricted Master Problem. The GA first generates a starting population P of feasible solutions and identifies an initial best chromosome through the computation of a best initial fitness, named *BestFit*. During the evolution process the *BestFit* value

Algorithm 2: WSNGenetic

Input: $(S, T), DP$;
Output: a subset of chromosomes(i.e. columns) for the MasterProblem;

```

1  $P \leftarrow \text{Init}P()$ ;
2  $\text{BestFit} \leftarrow \text{bestFitness}(P, DP)$ ;
3  $\text{criteria} \leftarrow \text{setCriterion}(\text{MaxIT}, \text{MaxDB})$ ;
4 while  $\text{check}(\text{criteria})$  do
5    $(p_1, p_2) \leftarrow \text{tournament}(P)$ ;
6    $C \leftarrow \text{Crossover}(p_1, p_2)$ ;
7    $C \leftarrow \text{Mutation}(C)$ ;
8    $C \leftarrow \text{feasibilityOperator}(C)$ ;
9    $C \leftarrow \text{redundancyOperator}(C)$ ;
10  if  $C \notin \text{Pop}$  then
11     $\text{Insert}(C, P)$ ;
12    if  $\text{fitness}(C) \geq \text{BestFit}$  then
13       $\text{update}(\text{criteria})$ ;
14    else
15       $\text{BestFit} \leftarrow \text{fitness}(C)$ ;
16  else
17     $\text{update}(\text{criteria})$ ;
18  $\text{Chromos} \leftarrow$  chromosomes with fitness  $\leq 1$ ;
19 return  $\text{Chromos}$ ;
```

stores the value of the *incumbent* solution and it is used as a comparison parameter through the overall procedure. The population of individuals has a fixed size, named Size_P , throughout the algorithm execution, and it is initialized as described in Section 4.5. The genetic algorithm builds iteratively new chromosomes one by one, by executing the steps reported in Sections 4.2-4.4. The new child produced at each iteration is inserted in current population P only if it does not already belong to it and in this case it replaces an individual which is chosen randomly among the $\lfloor P/2 \rfloor$ individuals with the worst fitness values.

The procedure iterates until one of two stopping criteria is reached. The first criterion is based on a *MaxIT* parameter, representing the maximum number of iterations without improvements with respect to the *BestFit* value, and the second one is the maximum number of consecutive duplicate chromosomes, named *MaxDB*.

The chromosomes in the final population P whose fitness value is less than 1 are then introduced in the Master Problem as new columns.

The GA algorithm was also used in our tests to provide the initial set of columns which is required by the first step of the master problem. In this case, however, the vector of dual prices which is used to evaluate the chromosomes is not available. For this reason, in this first iteration a random positive value is used as dual price for each sensor. The whole set of Size_P individuals is returned to the master problem in this case.

5 Computational Results

The purpose of the computational experience presented in this section is to study the performance of our algorithm (GCG) with respect to column generation approaches proposed in literature by [12] for the case $\alpha = 1$ and by [13], named GR from now on, for the α -coverage problem. The computational tests are carried out on the same set of instances used in these two papers. Our algorithm was coded in C++ on a (SUSE) Linux platform running on a Intel Core2 Duo 2.4GHz processor with 4GB RAM (single thread mode). Mathematical formulations within the GCG framework were solved using the Concert library of IBM ILOG CPLEX 12.5.

We ran a preliminary tuning test phase to determine the values used by the GA parameters. The $Size_{pop}$ population size was set to be equal to 50. The population initialization threshold $maxinitDB$ was chosen equal to 100. Finally, the two stopping criteria $MaxDB$ and $MaxIT$ were set to 100 and 2000, respectively.

Sensors	Targets	Lifetime	Time	Inv	Col	Flr
50	30	3.80	0.21	1.0	0.0	1.0
	60	3.00	0.31	1.0	0.0	1.0
	90	2.80	0.40	1.0	0.0	1.0
	120	2.70	0.51	1.0	0.0	1.0
100	30	8.70	0.44	1.6	10.1	1.0
	60	7.20	0.65	1.4	6.1	1.0
	90	6.90	1.11	1.6	8.5	1.0
	120	6.70	1.57	1.5	7.4	1.0
150	30	14.70	0.80	2.6	20.4	1.0
	60	12.30	1.41	2.4	18.8	1.0
	90	11.80	2.40	2.3	19.6	1.0
	120	11.30	3.38	2.3	19.9	1.0
200	30	19.60	1.24	2.9	24.4	1.0
	60	17.30	2.39	2.6	23.2	1.0
	90	16.60	4.10	3.0	24.5	1.0
	120	15.50	5.14	2.7	24.4	1.0
Avg				1.93	12.96	1.0

Table 1 Results obtained by the GCG algorithm on the benchmark instances proposed in [12].

Let us start our comparison from the benchmark instances proposed in [12]. In Table 1 the results of GCG are reported. Each line in the table represents a *scenario* composed of 10 instances with the same characteristics but different topologies. Therefore, the results reported in each line are the average values on these 10 instances. For a detailed description of the characteristics of these scenarios see [12]. The first two columns (*Sensors* and *Targets*) report the

number of sensors and targets into the scenarios. The columns *Lifetime* and *Time* report the solution values and the CPU times, in seconds. The last three columns *Inv*, *Col* and *Flr* report how many times the genetic algorithm is invoked by the restricted master problem after the initialization phase, the average number of columns (i.e. attractive covers) returned by the genetic algorithm at each invocation (again excluding the starting one), and how many times the genetic algorithm returns zero columns (i.e. the number of failures), respectively. Finally, the last line of the table reports the average values of the last three columns.

The values of the *Time* column show that GCG is very fast on the considered instances, with a running time that is always lower than 6 seconds on all the scenarios. A more accurate analysis of the GCG performance will be carried on while discussing the results reported in the Table 2. However, let us first analyze the impact of the genetic algorithm within column generation approach. To this end, we focus on the values reported in the last three columns of Table 1.

The values of the column *Inv* show that the genetic algorithm is invoked very few times, with an overall average equal to 1.93. In particular, on the scenarios with 50 sensors, it is invoked just once and this means that the starting columns, provided by genetic algorithm during the initialization phase, already contain the columns of the optimal solution. Indeed, a single invocation after the initialization means that the GA failed and the exact subproblem certified that an optimal solution was indeed reached, otherwise GA would have been invoked again in the following iteration.

On the other scenarios, the average number of invocations slowly increases up to 3 (in the scenario with 200 sensors and 90 targets). The number of invocations is small since the genetic algorithm returns a significant number of attractive covers, which is on average equal to 12.96 columns, with a peak of 24.5, which brings the columns needed to reach an optimal solution to be quickly added to the master problem. In particular, on the largest instances with 200 sensors the average number of returned columns is above 24, that is, almost 50% of the chromosomes in the final population are attractive covers for the restricted master problem. The more interesting results are, however, those reported into the column *Flr* which measure the effectiveness of the genetic algorithm. Remarkably, on all the scenarios provided by Deschinkel the number of GA failures is equal to 1, meaning that we need to solve the exact subproblem only once for each instance, in order to certify the optimality of the current incumbent solution.

In order to verify the competitiveness of our approach with respect to those proposed in the literature, the computational times of GCG and those of the *Exact*, *Heur* and *Mixed* algorithms described in [12], are reported in Table 2.

The first three columns show the characteristics of the scenarios, as already mentioned for Table 1. The subsequent four columns report the CPU time required by the four algorithms. The last three columns report the percentage gap, among GCG and the other three algorithms, computed as $100 \times (Alg - GCG)/Alg$ where $Alg \in \{Exact, Heur, Mixed\}$, and *GCG*, *Alg* refer to the

Sensors	Targets	Lifetime	Exact	Heur	Mixed	GCG	GAP		
			Time	Time	Time	Time	vs Exact	vs Heur	vs Mixed
50	30	3.80	0.25	0.30	0.12	0.21			
	60	3.00	1.03	0.53	0.52	0.31			
	90	2.80	2.95	0.82	1.55	0.40	86.42%		74.15%
	120	2.70	8.40	1.20	4.03	0.51	93.87%		87.22%
100	30	8.70	3.29	2.97	1.03	0.44	86.75%	85.32%	
	60	7.20	26.53	4.25	8.41	0.65	97.55%	84.71%	92.28%
	90	6.90	243.95	6.82	74.19	1.11	99.55%	83.77%	98.51%
	120	6.70	749.46	9.70	220.64	1.57	99.79%	83.79%	99.29%
150	30	14.70	17.17	14.51	4.94	0.80	95.37%	94.52%	83.89%
	60	12.30	315.66	22.21	48.96	1.41	99.55%	93.65%	97.12%
	90	11.80	2365.65	30.61	525.21	2.40	99.90%	92.17%	99.54%
	120	11.30	9249.81	48.15	1987.04	3.38	99.96%	92.98%	99.83%
200	30	19.60	38.80	34.85	9.50	1.24	96.80%	96.44%	86.93%
	60	17.30	750.40	56.34	126.39	2.39	99.68%	95.75%	98.11%
	90	16.60	8229.53	132.46	1297.82	4.10	99.95%	96.91%	99.68%
	120	15.50	28942.49	105.87	4393.04	5.14	99.98%	95.15%	99.88%
AVG			3184.09	29.47	543.96	1.63	96.79%	91.26%	93.57%

Table 2 Comparative of *GCG*, *Exact*, *Heur* and *Mixed* algorithms on the Deschinkel's benchmark instances.

computational time of the related procedure. Finally, the last line of the table reports the average values for the last seven columns. Note that when the CPU time gap between two algorithms is lower than 1 second, we do not report the percentage gap because we consider this gap to be negligible.

As previously mentioned, the results of the Time column for GCG show that it is able to find the optimal solution in less than 6 seconds on average whatever are the characteristics of the considered scenario. Therefore, the increment in terms of requested CPU time, as the size of scenarios grows, is bounded to few seconds. The situation appears to be completely different for the other three algorithms, that appear to be much slower, and whose computational times are significantly affected by the scenarios characteristics. More in details, from the average values of the last line it is clear that GCG is faster than Exact by three orders of magnitude, with a gap that is always greater than 86%. In particular, for the scenario containing the largest instances (that is, with 200 sensors and 120 targets) the Exact algorithm spends more than 8 hours to find the optimal solution while GCG requires less than 6 seconds. The Mixed algorithm results faster than the Exact algorithm, however when compared to the GCG algorithm it appears to be slower by two orders of magnitude. Moreover, the performance gap between these two algorithm is always greater than 83%. Finally, it is remarkable to note that GCG results to be 20 times faster than the heuristic approach *Heur*, with a percentage gap which is always greater than 74%.

It has to be highlighted that this comparison cannot be completely accurate since the algorithms proposed in [12] were run on a different hardware and the mathematical models were solved using GLPK. However, since the running time gap can be quantified in orders of magnitude, we believe that the comparison still provides solid evidence about the effectiveness of our approach.

The results of GCG, on one hand, confirm our expectations on the effectiveness and efficiency of our GA algorithm and, on the other hand, prove

Inst. Subgroup	Targets	T_α	LifeTime	Time	Inv	Col	Flr
Design	100	50	8.32	0.41	4.30	19.20	1.03
		75	5.42	0.77	9.90	13.63	2.00
		85	4.50	0.90	11.50	12.27	2.87
		93	3.65	0.52	6.97	14.03	1.57
		95	3.34	0.39	4.80	11.87	1.20
		97	3.04	0.26	2.13	7.43	1.00
		99	3.00	0.27	2.03	11.90	1.00
		100	3.00	0.29	2.37	13.63	1.00
Scattering	100	50	20.50	1.19	6.20	23.30	1.03
		75	13.36	9.14	39.77	10.03	8.07
		85	10.57	9.12	52.57	8.13	10.47
		93	7.73	2.35	16.10	17.77	2.10
		95	6.64	1.22	7.63	20.27	1.40
		97	5.37	0.74	3.37	17.00	1.03
		99	3.83	0.56	1.67	7.17	1.00
		100	3.00	0.48	1.00	0.00	1.00
Avg				10.77	12.98	2.36	

Table 3 Results obtained by the GCG algorithm on the Group 2 benchmark instances proposed in [13].

that a column generation approach, paired with a fast and effective method to generate new columns, results to be a very suitable approach for lifetime problems on sensor networks.

We now present the results of GCG when used to solve the Group 2 set of benchmark instances proposed in [13] for the α -coverage problem. This is the hardest set of instances considered in that paper, and therefore we considered the results on it to be more relevant and interesting. Nevertheless, we also tested our approach on the Group 1 dataset, and the related tables are contained in the Appendix. As will be shown, GCG performs well on all these instances as well.

The Group 2 instances contain 100 targets, while the number of sensors is not fixed *a priori*, but is rather computed assuring that each target is covered by at least 3 sensors. The instances are further divided in two subgroups, named *Scattering* and *Design* respectively. In the *Scattering* group sensors are added randomly until the desired coverage level is reached, while in the *Design* group, sensors are added only when needed to reach such coverage. For a detailed description of the characteristics of these instances see [13].

In Table 3, the results of GCG on the Scattering and Design scenarios are reported. The first two columns specify the type of instance and the number of targets present in it. The column T_α specifies the number of targets that must be covered, while the columns *Lifetime* and *Time* reports the solution value and the requested CPU time, respectively. Finally, the last 3 columns report for GA the same values which we already mentioned regarding Table 1, and the last line of the table reports the average values of these columns. Each line in the table represents a scenario composed by 30 instances with the same characteristics, therefore the results reported in each line are the average values on these 30 instances.

Inst. Subgroup	T_α	GR		GCG		GAP
		LifeTime	Time	LifeTime	Time	
Design	50	8.32	3.20	8.32	0.41	87.28%
	75	5.42	13.94	5.42	0.77	94.46%
	85	4.50	11.46	4.50	0.90	92.15%
	93	3.65	7.03	3.65	0.52	92.56%
	95	3.34	2.68	3.34	0.39	85.41%
	97	3.04	1.43	3.04	0.26	81.61%
	99	3.00	0.59	3.00	0.27	
	100	3.00	0.21	3.00	0.29	
Scattering	50	20.50	11.13	20.50	1.19	89.30%
	75	13.36	216.98	13.36	9.14	95.79%
	85	10.56**	302.91	10.57	9.12	96.99%
	93	7.38*	36.18	7.73	2.35	93.50%
	95	6.64	8.02	6.64	1.22	84.78%
	97	5.37	2.01	5.37	0.74	63.15%
	99	3.83	0.56	3.83	0.56	
	100	3.00	0.05	3.00	0.48	
AVG		38.65		1.79	88.08%	

Table 4 Computational results of GCG and GR algorithms for the α -coverage WSN problem.

The results of Table 3 show that for these instances the number of invocations is on average 10.77, the number of columns returned is approximately 12.98 and the number of average failures is 2.36. More in detail, on the *Design* scenarios we register a peak of GA invocations equal to 11.50 for the case $T_\alpha = 85$, which also corresponds to the peak of failures, equal to 2.87. The average number of columns returned for each iteration is greater than 10 in all cases except one, in the case $T_\alpha = 97$. The *Scattering* instances result to be harder to solve, with a peak of GA invocations and failures corresponding to 52.57 and 10.47, respectively (again in the case $T_\alpha = 85$). This can be explained considering the additional number of sensors, and therefore the higher amount of feasible covers which exists in such instances.

It can be noticed that also on this dataset GA only fails once for the highest values of T_α , and therefore the problem approaches the classical MLP. In particular, this happens for each instance with $T_\alpha \geq 97$ for the *Design* dataset and with $T_\alpha \geq 99$ for the *Scattering* one.

Despite the results appear to be less impressive than the ones presented in 1, the values in the Time column show that GCG is still very fast. Indeed, the algorithm finds the optimal solution in less than 1 second on average in all scenarios for the *Design* instances, and always in less than 10 seconds on average for the *Scattering* ones.

In Table 4 a performance comparison between GCG and the GR algorithm is performed. As mentioned above, we do not evaluate gaps when both procedures report a computational time which is below 1 second. On the *Design* scenarios, the GR algorithm finds all solution within the considered 1 hour

time limit. However, it is clear that GCG is generally much faster, with a percentage gap greater than 81% on the first 6 scenarios and a CPU Time always lower than a second. More interesting are the results on the *Scattering* scenarios, where some of its instances are not solved within the time limit by the GR algorithm. More in detail, it reaches the time limit on 2 instances of the scenario with $T_\alpha = 85$ and 3 instances of the scenario with $T_\alpha = 93$. The solution values of these scenarios are marked into the table with the symbols “*” and “**” to highlight that these values are averages evaluated only on the subset of instances which were solved to completion.

The values of column GAP show that GCG is at least 63% faster than GR with a peak equal to 97% and an average equal to 88%. The values reported in the last line show that GCG is faster than GR by an order of magnitude with a CPU time lower than 2 seconds with respect to the 38 seconds required by GR algorithm. These results certify that GCG is the fastest algorithm and that it is also more effective, since it can solve within 10 seconds at most on average all the considered scenarios.

6 Conclusion

In this work we addressed the maximum lifetime problem on wireless sensor networks, and more in particular we considered two variants in which either all sensors have to be covered, or a portion of them can be neglected at all times in order to increase the overall network lifetime. We presented an efficient genetic algorithm aimed at producing new covers, which can be embedded within a Column Generation framework. The obtained algorithm is shown to be highly efficient in terms of requested computational time, and to perform significantly better than the ones proposed in the literature.

Further research will involve the study of more complex problem variants, able to model aspects such as sensor-to-sensor communication.

Acknowledgements

The authors wish to thank K. Deschinkel, who provided the set of benchmark instances proposed in [12].

References

1. H. Alemdar and C. Ersoy. Wireless sensor networks for healthcare: a survey. *Computer Networks*, 54(15):2688–2710, 2010.
2. A. Alfieri, A. Bianco, P. Brandimarte, and C. F. Chiasserini. Maximizing system lifetime in wireless sensor networks. *European Journal of Operational Research*, 181(1):390–402, 2007.
3. C. Blum, M. J. Blesa Aguilera, A. Roli, and M. Sampels, editors. *Hybrid Metaheuristics - An Emerging Approach to Optimization*, volume 114 of *Studies in Computational Intelligence*. Springer-Verlag, Berlin/Heidelberg, 2008.

4. M. Cardei, M. T. Thai, Y. Li, and W. Wu. Energy-efficient target coverage in wireless sensor networks. In *Proceedings of the 24th conference of the IEEE Communications Society*, volume 3, pages 1976–1984, 2005.
5. M. Cardei, J. Wu, and M. Lu. Improving network lifetime using sensors with adjustable sensing ranges. *International Journal of Sensor Networks*, 1(1-2):41–49, 2006.
6. F. Carrabs, R. Cerulli, C. D’Ambrosio, M. Gentili, and A. Raiconi. Maximizing lifetime in wireless sensor networks with multiple sensor families. Submitted, 2014.
7. F. Castaño, E. Bourreau, N. Velasco, A. Rossi, and M. Sevaux. Exact approaches for lifetime maximization in connectivity constrained wireless multi-role sensor networks. *European Journal of Operational Research*, 241(1):28–38, 2015.
8. F. Castaño, A. Rossi, M. Sevaux, and N. Velasco. A column generation approach to extend lifetime in wireless sensor networks with coverage and connectivity constraints. *Computers & Operations Research*, 52(B):220–230, 2014.
9. R. Cerulli, R. De Donato, and A. Raiconi. Exact and heuristic methods to maximize network lifetime in wireless sensor networks with adjustable sensing ranges. *European Journal of Operational Research*, 220(1):58–66, 2012.
10. R. Cerulli, M. Gentili, and A. Raiconi. Maximizing lifetime and handling reliability in wireless sensor networks. *Networks*, 64(4):321–338, 2014.
11. L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.
12. K. Deschinkel. A column generation based heuristic for maximum lifetime coverage in wireless sensor networks. In *SENSORCOMM 11, 5th Int. Conf. on Sensor Technologies and Applications*, volume 4, pages 209 – 214, 2011.
13. M. Gentili and A. Raiconi. α -coverage to extend network lifetime on wireless sensor networks. *Optimization Letters*, 7(1):157–172, 2013.
14. M. Pejanovic Durisic, Z. Tafa, G. Dimic, and V. Milutinovic. A survey of military applications of wireless sensor networks. In *Proceedings of the Mediterranean Conference on Embedded Computing*, pages 196–199, 2012.
15. A. Raiconi and M. Gentili. Exact and metaheuristic approaches to extend lifetime and maintain connectivity in wireless sensors networks. In J. Pahl, T. Reiners, and S. Voss, editors, *Network Optimization*, volume 6701 of *Lecture Notes in Computer Science*, pages 607–619. Springer, Berlin/Heidelberg, 2011.
16. P. Rawat, K. D. Singh, H. Chaouchi, and J. M. Bonnin. Wireless sensor networks: a survey on recent developments and potential synergies. *The Journal of Supercomputing*, 68(1):1–48, 2014.
17. A. Rossi, A. Singh, and M. Sevaux. An exact approach for maximizing the lifetime of sensor networks with adjustable sensing ranges. *Computers & Operations Research*, 39(12):3166–3176, 2012.
18. A. Rossi, A. Singh, and M. Sevaux. Lifetime maximization in wireless directional sensor network. *European Journal of Operational Research*, 231(1):229–241, 2013.
19. Q. Zhao and M. Gurusamy. Lifetime maximization for connected target coverage in wireless sensor networks. *IEEE/ACM Transactions on Networking*, 16(6):1378–1391, 2008.

Appendix

Tables 5 and 6 contain the results related to the Group 1 instances proposed in [13]. Each instance in this group contain 15 targets, while the number of sensors for the different scenarios is specified by the *Sensor* heading in the tables. Each line in the tables contain averages over 5 different instances with the same characteristics. For a description of the table headings, refer to the description of Tables 3 and 4 in the paper.

Sensors	T_α	Lifetime	Time	Inv	Col	Flr
25	8	13.60	0.29	3.00	11.80	1.00
	11	10.40	0.26	3.40	16.40	1.00
	13	6.60	0.19	2.80	17.00	1.00
	15	3.60	0.19	2.00	19.60	1.00
50	8	27.23	0.59	4.60	19.40	1.00
	11	19.40	0.40	4.60	18.80	1.00
	13	13.93	0.33	3.60	21.00	1.00
	15	9.40	0.26	2.40	15.20	1.00
100	8	54.90	1.27	9.20	21.00	1.00
	11	41.49	1.25	11.00	23.20	1.20
	13	30.40	0.87	7.00	26.60	1.00
	15	15.40	0.52	3.00	21.80	1.00
150	8	87.60	2.39	11.80	22.00	1.00
	11	66.98	2.40	15.40	22.80	1.40
	13	51.72	1.97	12.20	27.60	1.00
	15	25.00	0.89	4.00	24.60	1.00
AVG				6.25	20.55	1.04

Table 5 Results obtained by the GCG algorithm on the Group 1 benchmark instances proposed in [13].

Sensors	T_α	GR		GCG		GAP
		LifeTime	Time	LifeTime	Time	
25	8	13.60	0.26	13.60	0.29	
	11	10.40	0.44	10.40	0.26	
	13	6.60	0.11	6.60	0.19	
	15	3.60	0.01	3.60	0.19	
50	8	27.23	1.11	27.23	0.59	
	11	19.40	0.68	19.40	0.40	
	13	13.93	0.39	13.93	0.33	
	15	9.40	0.01	9.40	0.26	
100	8	54.90	5.95	54.90	1.27	78.72%
	11	41.49	8.03	41.49	1.25	84.39%
	13	30.40	2.74	30.40	0.87	68.42%
	15	15.40	0.02	15.40	0.52	
150	8	87.60	15.24	87.60	2.39	84.31%
	11	66.98	13.90	66.98	2.40	82.74%
	13	51.72	9.79	51.72	1.97	79.86%
	15	25.00	0.02	25.00	0.89	
AVG		3.67		0.88	79.74%	

Table 6 Computational results of GCG and GR algorithms on the Group 1 benchmark instances proposed in [13].