Expert Systems With Applications 202 (2022) 117185

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Coverage area maximization with parallel simulated annealing ☆

Narcís Coll *, Marta Fort, Moisès Saus

Graphics and Imaging Laboratory, Universitat de Girona, Campus Montilivi, Girona, 17003, Spain

ARTICLE INFO

Keywords:
Facilities placement
Service coverage
Multiple services
Covered area
Simulated annealing
Parallel GPU processing

ABSTRACT

This study provides a system that determines where to locate k disks-like services of radius r so that they globally cover as much as possible a region of demand. It is an NP-hard problem with notorious applications in the facility location field when locating multiple warning sirens, cellular towers, radio stations, or pollution sensors covering as much area as possible of a city or geographical region. The region of demand is assumed to be delimited by a general polygonal domain, and the resolution strategy relies on a parallel simulated annealing optimization technique based on a suitable perturbation strategy and a probabilistic estimation of the area of the polygonal region covered by the k disks in $O(k^2)$ time. The system provides a good enough location for the disks starting from an arbitrary initial solution with very reasonable running times. The proposal is experimentally tested by visualizing the solutions, analyzing and contrasting their quality, and studying the computational efficiency of the entire strategy.

1. Introduction

One of the main objectives of private companies and public entities is to offer their services to as many citizens as possible with the minimum investment. That is why it is essential to know where to place their headquarters or service centers so that their locations optimize the coverage they offer.

People seek to be *covered* in the broadest sense of the expression. Indeed, this coverage may refer to mobile phones, electric charging points, warning sirens, radio stations, pollution sensors, delivery systems, weather forecasts or medical coverage. Moreover, citizens expect this coverage at home and wherever we do our daily or sporadic life. It results in the necessity of covering large geographical areas. And, as mentioned, both public and private entities have limited resources and locating them strategically so that few services cover as much area as possible is a clue element.

The study of facility location issues is also called location analysis. It is a field of both operations research and computational geometry that aims to locate several facilities in a given region in a way that maximizes or minimizes some objective function related to the type of facilities considered. Most of these problems consider that a demand point is covered by a facility if the distance or travel time between them is less than a given predetermined value, generally called the coverage radius. As we previously mentioned, the coverage concept frequently appears when planning the facilities' location for both the public and private sectors. Optimally, it would be desirable that all demand points become covered by at least one facility. But often, full coverage of all demand in a region is not possible due to budgetary limitations in the number of facilities to locate. Therefore, they aim at covering the greatest possible extent of the demand region by efficiently managing a limited set of resources.

Next, we present the background on the maximal coverage problem together with the existing challenges in Section 1.1, specify the problem solved in this paper in Section 1.2, explain the novelty of our contribution in Section 1.3, and finally describe the organization of the paper in Section 1.4.

1.1. Background

The problem of locating a specific number of facilities to maximize the amount of covered demand-region within an acceptable service distance was formalized and introduced by Church and ReVelle (1974). It was called the Maximal Covering Location Problem (MCLP) and to solve it they consider two sets of discrete locations representing the demand and the possible facility sites. Consequently, they convert the MCLP to an integer linear program that allows some variations. Variations come from: using different ways to determine the coverage area of a potential facility or assigning weights to the sites according to several factors and parameters. These problems are typically called the maximal covering problems.

While the standard form of MCLP considers a finite set of candidate locations for the facilities and a set of discrete points represent the

☆ Research supported by PID2019-106426RB-C31 of the Spanish Government.
* Corresponding author.
E-mail addresses: coll@imae.udg.edu (N. Coll), mfort@imae.udg.edu (M. Fort), moises_saus@hotmail.com (M. Saus).

https://doi.org/10.1016/j.eswa.2022.117185
Received 12 July 2021; Received in revised form 10 January 2022; Accepted 1 April 2022
Available online 13 April 2022

demand, its generalization, called the planar MCLP (Church, 1984), allows both the facilities and the demand to be located anywhere in the continuous plane. Solving exactly a maximal covering problem assuming that the demand to cover exists anywhere in a continuous region and that facilities can also be placed arbitrarily in a domain is not feasible due to the NP-hard nature of the problem (Hochbaum, 1996; Megiddo et al., 1983). Therefore, several heuristic techniques trying to obtain acceptable solutions have been proposed. For instance, there exists a greedy algorithm for MCLP that chooses, at each step, the location containing the highest number of uncovered elements. Hochbaum (1996) prove that this greedy algorithm achieves an approximation ratio of $1 - 1/e$ and that it is the best-possible polynomial-time approximation algorithm.

Other approaches start by partitioning the demand into several small regions and distinguish between completely covered or uncovered regions (Church, 1984; Murray, O'Kelly et al., 2008; Murray & Tong, 2007; Tong & Murray, 2009). They assume that a facility covers a demand region if it covers the whole region. Partially covered regions are considered uncovered regions. It is typically called *binary coverage* in contrast with *partial coverage*. Considering binary coverage allows, first, obtaining a finite set of potential facility sites, referred to as polygon intersection point set (PIPS), that contains an optimal solution and, second, treating the problem as an integer linear program. However, although binary coverage makes planar MCLP tractable, this approach introduces significant errors because all the partially covered regions, which may have a significant amount of covered area, are not taken into account and ignored in the final solution (Wei & Murray, 2014).

Considering partial coverage is not trivial. That is why, several papers consider the one-single facility case under partial coverage. Matisziw and Murray (2009) develop a technique for addressing the location problem of siting a circular shape service area in a polygonal region based on computing the medial axis of the region. Coll et al. (2019) approximate the optimal position of a single circular shape service area with bounded error in an unconnected domain with holes bounded by linear or circular segments while considering partial coverage. They provide an efficient algorithm to exactly compute the overlap area of a circle and a polygonal domain and introduce the covering area-map concept, see Section 2.1 for further details. Bansal and Kianfar (2017) also study partial coverage concretely with rectangular service zones. Circular service zones are studied by Murray, Matisziw et al. (2008). They avoid binary-covering by perturbing an initial solution strategically relocating the disks one-by-one. They use a computationally prohibitive geometry-based approach that, according to Wei and Murray (2014), guarantees optimally only for some limited cases because they have no control of the area covered by the disks.

The computational geometry field also studies similar problems. For instance, Mount et al. (1996) provide an algorithm to compute the maximum overlap between two simple polygons P and Q with n and m vertices in $O(n^2m^2)$ time. The maximum overlap between two polygons can be approximately found, with high probability, in near quadratic time with respect $\max(m, n)$ with the algorithm presented by Cheong et al. (2007) which uses random sampling techniques. Cheng and Lam (2013) extend this approach to approximate the maximum overlap between two polygons with multiple holes.

Therefore, the challenge is developing a fast and robust algorithm to solve the maximum continuous coverage problem of the disk-shaped facilities. It should not be based on a demand partition, nor on using a discrete set of possible locations, nor on a local optimization heuristic that does not lead to a global optimum. That is optimally locating k-disks in a continuous domain that globally maximizes the area they cover, which implies finding a way to measure the area covered by a set of k-disks within a region.

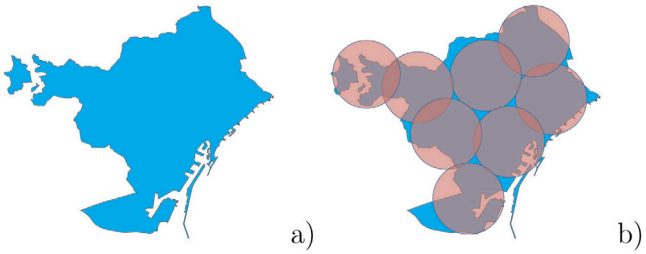

Fig. 1. (a) Polygonal domain to be covered; (b) Domain partially covered by eight facilities.

1.2. Problem definition

In this paper, we focus on a multiple-facility location problem. We assume that demand is uniformly distributed on a polygonal domain, $\mathcal{P}$, which may be defined by several connected components that may have holes. Holes may represent already covered areas or inaccessible regions. We consider k facilities with coverage radius r represented by k disk-like service areas of the same radius r. Facilities can be located anywhere in the plane whenever they intersect $\mathcal{P}$. Domain $\mathcal{P}$ may not fully contain their service areas, and their union may not entirely cover $\mathcal{P}$. See Fig. 1 for an example.

We are interested in locating the k disks so that their location maximizes the area of the parts of $\mathcal{P}$ covered by the disks or, equivalently, minimizes the not-covered area of $\mathcal{P}$. Indeed, if we denote by $c_1, \dots, c_k$ the k centers and by $D(c_i, r)$ the disk of center c_i and radius r, we aim to determine the k centers $c_1, \dots, c_k$ that maximize the area of the overlap between $\mathcal{P}$ and the union of the k disks. Hence, the value of the objective function of our covering problem is given by the exact value of

$$Area(\mathcal{P} \cap \bigcup_{i=1}^{k} D(c_i, r)).$$

Both, determining the optimal location of the k-centers, and obtaining the exact value of the mentioned area are NP-hard problems (see Sections 2 and 3).

Hence, our aims are:

(i) to provide a practical objective function that estimates the overlapped area of $\mathcal{P}$
(ii) to present an heuristic method to obtain an approximate optimal location for the k-centers that globally optimizes the used objective function.

1.3. Our contribution

In this section, we provide the contributions of this paper while highlighting the novelties of the solution.

We develop a fast and robust algorithm to solve the planar maximal continuous coverage problem of k-disk-shaped facilities. It globally maximizes the area they cover using parallelism and Simulated annealing (SA). SA is a powerful technique widely used to find the global maximum/minimum of a function. Since finding the optimal is a computationally demanding task requiring a large volume of tests and operations, it is a good option using parallelism. Programmable graphic processing units (GPU) have high computational rates and notorious parallel processing capability, making them powerful platforms for accelerating these processes.

Concretely, the contributions of the paper are:

- An $O(k^2)$-time probabilistic estimation of the area of the polygonal region of demand $\mathcal{P}$ covered by k disks. This estimation does not rely on a discretization of $\mathcal{P}$, i.e., it does not use binary-covering.

- A parallel heuristic algorithm, based on the SA technique, to locate k disks of radius r on a continuous set of possible positions so that their union covers $\mathcal{P}$ as much as possible. That is, the estimation of the covered area has to reach its global maximum at the obtained locations of the disks.
- A strategy for generating random perturbations of the k disks that ensures that each new disk intersects $\mathcal{P}$.
- An experimental results collection showing the obtained results and analyzing their quality.
- An experimental and theoretical time complexity analysis of the proposed strategy demonstrating its efficiency.

1.4. Paper organization

The paper is structured as follows. Section 2 summarizes the previous work used in our strategy. In Section 3, we discuss the overlapped area computation and provide a usable formula that estimates its value in $O(k^2)$ time. We present and theoretically analyze our proposal in Section 4. In Section 5, a modest interface developed to deal with the problem and test our strategy is presented together with a complete collection of experimental results showing the quality, scalability and efficiency of the algorithm. Finally, in Section 6 some conclusions and further comments are provided.

2. Related work

In this section we summarize the previous works with special importance to understand the proposed method. Section 2.1 contains the main ideas of the overlap-area map computation for one single disk presented by Coll et al. (2019), that is the basis of part of our strategy. Section 2.2 provides the main issues of the simulated annealing technique used in our proposal to obtain an approximation of the global optimum of our overlap area function.

2.1. Overlap-area map computation

As mentioned before, Coll et al. (2019) approximate the optimal position of a single circular shape service area with a ε bounded error by using ε-overlap-area maps. They introduce the concept of uniform and nonuniform overlap-area maps. A (uniform) overlap-area map is a discrete graph that maps each center c of a uniform grid of square cells covering $\mathcal{P}$ to the overlap area $D(c,r) \cap \mathcal{P}$. Nonuniform overlap-area maps use grids obtained with a global grid refinement method. The ε guarantees that the obtained solutions were ε approximations of the optimal solution.

Computing the overlapped area between a disk $D(c,r)$ and the polygonal domain $\mathcal{P}$ is not trivial when partial coverage is taken into account and was a clue element of their algorithm. They developed an exact algorithm that computes the intersection area between a disk and a piecewise circular domain. Piecewise circular domains may have holes and unconnected regions delimited by line segments or circular sectors defining the edges connecting one vertex with the next one. The algorithm to compute the overlap area traverses the edges of $\mathcal{P}$ per order, considering one component of $\mathcal{P}$ after the other. They detect whether the analyzed edge intersects the boundary of $D(c,r)$. At these intersecting edges (or the ending edge of a component), they compute the area of $D(c,r) \cap \mathcal{P}$ covered by the sequence of analyzed edges since the previously intersecting edge (or the starting edge of the component). The exact overlapped area is obtained in $O(n)$ time with few calculations done at each of the n edges defining $\mathcal{P}$ whenever we can traverse $\mathcal{P}$ via adjacent edges.

Moreover, they describe a GPU implementation of the overlap area computation. Indeed, they compute the overlap-area maps in parallel in the GPU and experimentally and theoretically prove that it is much more efficient than the correspondent sequential version. To obtain a $H \times W$ uniform overlap-area map, they use a kernel that has as input: (i) the coordinates of the bottom-left corner of the region covered by the grid defining the map; (ii) the size of the grid cells; and (iii) the domain $\mathcal{P}$ stored in two arrays, one with the vertices coordinates and the other describing its components and holes. This kernel is executed by a $H \times W$ grid of threads and each thread is identified by a two dimensional integer index $(id_x, id_y) \in [0,H) \times [0,W)$ that directly associates it to a cell of the considered grid. Each thread computes the center c of the cell it represents and then computes the area of $\mathcal{P}$ covered by the disk centered in c. The area is computed with the before-mentioned algorithm and then stored in a $H \cdot W$ array associated with the grid (linearized in a row first fashion). The array of areas is the kernel output. Note that the grid cell centers are computed on the fly and not stored for not being further needed.

2.2. Simulated annealing

Simulated annealing (SA) is a metaheuristic stochastic optimization technique for approximating the global optimum of an objective function $\overline{\mathcal{O}}(x)$. Usually approximates the minimal of an objective function that is called *energy*. Kirkpatrick et al. (1983) argue that this technique tries to imitate the annealing process used in metallurgy. Since then, it has been used to solve many different facility location problems (Allahyari & Azab, 2018; Davari et al., 2011; Nasab & Mobasheri, 2013).

At each step, the SA technique considers some new state x', close to the current state x, and probabilistically decides between staying at x or moving to x'. The process performs, at most, a predefined maximum number of steps but can terminate earlier if the energy of the current state is low enough. The probability of transition from x to x' depends on their energy, $\overline{\mathcal{O}}(x)$, $\overline{\mathcal{O}}(x')$ and a global parameter T called temperature that decreases while the number of steps increases. Therefore, the sequence of changes between states can be viewed as a Markov chain because the probability of transition between states only depends on the current state and not on the previous ones. Certainly, the relevant aspects of a SA process are the three following ones:

1. **The election of the initial temperature T_0 and its decreasing model**
   There is no general way to determine the best option for the starting temperature T_0. The general recommendation is that it has to be large enough. Often, an approximation of the difference between the maximum and minimum values of the objective function is used. Regarding how it should decrease, some papers use a multiplicative scheme in which the temperature follows a geometric progression and some others a logarithmic scheme. For the multiplicative one, the formula to obtain T_j, the temperature of step j, is $T_j = \alpha T_{j-1}$, with $\alpha \in [0.9, 1)$. However, for the logarithmic schedule $T_j = \frac{T_0}{\log(j+1)}$. According to Hajek (1988), the logarithmic scheme converges to the global minimum if T_0 is greater than or equal to the suitably defined depth of the deepest local, but not global, minimum state.
2. **The random generation of the new state x'** No general indications are given. In some papers adaptive neighborhoods are considered (Tavares et al., 2011).
3. **The acceptance criterion of the new state x'** If x' improves x, its energy $\overline{\mathcal{O}}(x')$ is lower than $\overline{\mathcal{O}}(x)$, x' is accepted. Otherwise, if x' deteriorates x and $\overline{\mathcal{O}}(x) < \overline{\mathcal{O}}(x')$, x' is accepted with probability equal to $\exp\left(-\frac{\overline{\mathcal{O}}(x')-\overline{\mathcal{O}}(x)}{T_j}\right)$.

Indeed, this process can also be used to find a global maximum of an objective function. In this case, the probability to accept x' when it deteriorates x should be $\exp\left(\frac{\overline{\mathcal{O}}(x')-\overline{\mathcal{O}}(x)}{T_j}\right)$.

The SA process leads to a global optimal, but it is a computationally demanding technique. Usually, the generation of x', and the computation of its energy, are time-consuming processes. Moreover, the

number of steps performed to obtain good enough solutions tends to be considerable. Therefore, the literature contains several approaches to parallelize the SA. We categorize them into the four following types:

1. **Domain decomposition** (Zhang et al., 2019): they subdivide the domain and associate each subdomain to a different processor. Then, each processor searches for an optimal in its subdomain to finally obtain the global minimum from the minimums obtained by all processors.
2. **Energy evaluation** (Kravitz & Rutenbar, 1987): they distribute the computation of the objective function among the set of processors.
3. **Asynchronous Markov chains** (Ferreiro et al., 2013; Lee & Lee, 1996; Onbaoğlu & Ozdamar, 2001): each processor launches a Markov chain and obtains the global minimum from the minimums obtained by all processors.
4. **Synchronous Markov chains** (Chu et al., 1999; Czech et al., 2010; Ferreiro et al., 2013; Lee & Lee, 1996; Onbaoğlu & Ozdamar, 2001; Ram et al., 1996): each processor launches its own Markov chain, processors exchange information every certain number of steps, and finally obtain the global minimum from the minimums of all processors.

Several of these works take advantage of the capabilities of the Graphic Processing Units (GPUs) to use them for general-purpose programming. Concretely (Fabris & Krohling, 2012; Ferreiro et al., 2013; Sonuç et al., 2017, 2018; Wei et al., 2015) have been devoted to the parallelization of the SA for Nvidia GPUs in CUDA and to solve several optimization problems.

3. Overlapped area

In this section, we determine a way to compute the overlapped area that allows us to define the objective function of our problem. We start setting some notation and discussing the overlapped area computation. Section 3.1 contains a probabilistic approach to estimate the overlapped area that defines the formula of our objective function. Finally, Section 3.1.1 explains how to evaluate this objective function in $O(k^2)$ time.

We denote by k the number of disk-like services to locate, r the disk radius, $D(c,r)$ the disk of center c and radius r, $\mathcal{P}$ the polygonal domain to be covered, $I(c,r)$ the part of $\mathcal{P}$ contained in $D(c,r)$, $\mathcal{B}$ the bounding box of $\mathcal{P}$, $\overline{\mathcal{P}}$ the polygonal domain $\mathcal{P}$ amplified with an offset of size r, and $\overline{\mathcal{B}}$ the bounding box of $\overline{\mathcal{P}}$. According to this notation disks $D(c,r)$ with $c \notin \overline{\mathcal{P}} \subset \overline{\mathcal{B}}$ will not intersect $\mathcal{P}$, i.e. for all $c \notin \overline{\mathcal{B}}$ $Area(I(c,r)) = 0$.

In our proposal, we consider configurations, x, of k centers in $\overline{\mathcal{B}}$, i.e. $x = (x_1, \dots, x_k) \in \overline{\mathcal{B}}^k$. To simplify the notation, given a configuration x, for every $1 \le i \le k$, we denote by D_i the disk $D(x_i, r)$ and by I_i the overlap region $I(x_i, r)$.

The problem of finding the optimal location of k disks can be stated as the problem of finding the configuration $x = (x_1, \dots, x_k)$ that maximizes the function area of overlap between $\mathcal{P}$ and the union of the disks:

$$\mathcal{O}(x) = Area(\mathcal{P} \cap \bigcup_{i=1}^{k} D_i)$$

By applying the inclusion–exclusion principle, the area of overlap can be computed as:

$$\mathcal{O}(x) = \sum_{1 \le i_1 \le k} Area(I_{i_1}) - \sum_{1 \le i_1 < i_2 \le k} Area(I_{i_1} \cap I_{i_2}) + \sum_{1 \le i_1 < i_2 < i_3 \le k} Area(I_{i_1} \cap I_{i_2} \cap I_{i_3}) - \cdots (-1)^k Area(I_1 \cap \cdots \cap I_k)$$

The evaluation of this objective function is an NP-hard problem. It requires the computation of $O(2^k)$ intersections between disks and the polygonal domain $\mathcal{P}$. Thus, we present an alternative objective function that estimates, in polynomial time, the overlap area between $\mathcal{P}$ and the union of the disks. The maximization of this function with the heuristic method presented in Section 4 will provide an approximation of the optimal location.

3.1. Estimated overlapped area

Aiming at estimating the overlapped area in polynomial time, in this section, we propose to use an objective function that approximates the overlapped area of $\mathcal{P}$ considering the area covered by every single disk and by each pair of disks.

The objective function we propose is formulated as follows:

$$\overline{\mathcal{O}}(x) = \sum_{1 \le i_1 \le k} Area(I_{i_1}) - \sum_{1 \le i_1 < i_2 \le k} \overline{Area}(I_{i_1} \cap I_{i_2}),$$

where $\overline{Area}(I_{i_1} \cap I_{i_2})$ is a probabilistic estimation of the area of the triple overlap $\mathcal{P} \cap D_{i_1} \cap D_{i_2}$. The estimation is obtained from the disk–disk overlap area $A_o = Area(D_{i_1} \cap D_{i_2})$, as the sum of the part of A_o that is know to be contained in $\mathcal{P}$ plus a probabilistic estimation of the remaining part of A_o.

The contribution of each of these parts can be deduced with the following reasoning, in which we denote by $A_M = \max(Area(I_{i_1}), Area(I_{i_2}))$ and $A_m = \min(Area(I_{i_1}), Area(I_{i_2}))$.

1. If the overlap area of a disk with $\mathcal{P}$, A_M, is bigger than the part of the disk not overlapped with the other disk, whose area is $\pi r^2 - A_o$, the disks intersection surely overlaps $\mathcal{P}$. That is, when $A_M > \pi r^2 - A_o$, the difference $A_M - (\pi r^2 - A_o)$ is necessarily in the triple-overlap $D_{i_1} \cap D_{i_2} \cap \mathcal{P}$. The portion of the remaining disks-intersection area, $A_o - (A_M - (\pi r^2 - A_o)) = \pi r^2 - A_M$, that may potentially be contained in $\mathcal{P}$ will probabilistically estimated.
2. When the inequality $A_M > \pi r^2 - A_o$ does not hold, we know nothing about the portion of A_o contained in $\mathcal{P}$. Hence, the probabilistic estimation affects the whole area A_o.
3. Let A be the area of overlap of a disk with $\mathcal{P}$, then, the probability that a portion of the disk overlaps $\mathcal{P}$ is $\frac{A}{\pi r^2}$. Hence, the probability that a portion of the disks is in the triple overlap is $\frac{A_M}{\pi r^2}\frac{A_m}{\pi r^2}$.

Taking into account these observations, the triple overlap area, $\overline{Area}(I_{i_1} \cap I_{i_2})$, is estimated with the following formula:

$$\overline{Area}(I_{i_1} \cap I_{i_2}) = \begin{cases} A_M - (\pi r^2 - A_o) + \frac{A_M}{\pi r^2}\frac{A_m}{\pi r^2}(\pi r^2 - A_M), & A_M > \pi r^2 - A_o \\ \frac{A_M}{\pi r^2}\frac{A_m}{\pi r^2} A_o, & \text{otherwise.} \end{cases}$$

Our aim is, therefore, to heuristically determine the location $x_{opt} \in \overline{\mathcal{B}}^k$ that maximizes the objective function $\overline{\mathcal{O}}$. This process considers several configurations x and the objective function is evaluated for each of them. Hence, it is important to determine $\overline{\mathcal{O}}(x)$ as fast as possible.

3.1.1. Objective function evaluation

The time needed to evaluate the objective function is one of the clue elements of the SA heuristic algorithms. Hence, it is worth analyzing its computational cost and trying to find a way to reduce it. This section analyzes this matter.

Given configuration a x, evaluating the objective function requires computing $Area(I_{i_1})$ for each of the k disks and $Area(D_{i_1} \cap D_{i_2})$ for each of the $O(k^2)$ pairs of disks. Each of the $O(k^2)$ latter values can be obtained in constant time, hence obtaining all of them takes $O(k^2)$ time. However, the computation of each of the k former ones requires $O(n)$ time, being n the number of edges of $\mathcal{P}$, according to Coll et al. (2019). Therefore, evaluating $\overline{\mathcal{O}}(x)$ requires $O(k(n+k))$ time.

To reduce its computation time and since the objective function is an estimation of the desired value, we reject obtaining the $Area(I_{x_i})$ exactly, and use, instead, an approximated value obtained in $O(1)$ time.

With this aim, we compute a uniform overlap area map covering $\overline{\mathcal{B}}$ in a preprocessing stage. The map defines a $H \times W$ uniform grid and,

for every cell, it stores the $Area(I(c,r))$ where c denotes the cell center. See Section 2.1 for further details. Now, given a configuration x, each of the k areas $Area(I_{i_1})$ can be approximated in $O(1)$ time. First, the disk center x_{i_1} is located in the uniform grid in $O(1)$ time. Then, the value $Area(I_{i_1})$ is obtained in $O(1)$ time by bilinear interpolation using the area values stored in the cell containing x_i and in its neighboring cells.

Overall, the overlap area map and the probabilistic estimation of the triple-overlapped region area, provided in Section 3.1, allows us to evaluate $\overline{\mathcal{O}}(x)$ in $O(k^2)$ time. Latter, in Section 4.1.3 we provide further details on how the objective function is evaluated in parallel in the GPU.

4. Proposal description

This section contains the description of our approach based on the parallel simulated annealing to determine the global maximum of the objective function. First, we provide a global overview of the approach, Section 4.1 details the SA algorithm with the generation of the Markov chains (Section 4.1.1), the new configuration generation (Section 4.1.2), the objective function evaluation (Section 4.1.3) and the parameter values determination (Section 4.1.4). Finally, Section 4.2 provides the complexity analysis of the proposed strategy.

The heuristic method we propose to maximize the objective function and obtain an approximation of the optimal location uses the GPU parallelism and the simulated annealing probabilistic technique. It does not require manual parameters tuning nor initial manual facilities location and consists of the following stages.

I. **Preprocessing stage:** The overlap-area map, $\mathcal{M}$, defined on a $H \times W$ grid covering $\overline{\mathcal{B}}$ is parallelly computed in the GPU by using the algorithm of Coll *et al.* described in Section 2.1.
II. **Starting location:** An optimal cell center, c, of $\mathcal{M}$, with maximal area value, is determined. The k facilities are, all, initially centered at this cell center c. The starting configuration is set to be $x_{ini} = (c, \dots, c)$.
III. **Dispersive stage:** A parallel simulated annealing algorithm to maximize $\overline{\mathcal{O}}$ run in the GPU. The parameter values guarantee that it disperses the k facilities throughout the polygonal domain amplified with an r-offset, $\overline{\mathcal{P}}$. It automatically determines an appropriate initial configuration x_0 starting from x_{ini}.
IV. **Refining stage:** The simulated annealing algorithm runs again in the GPU starting with configuration x_0 and with more conservative parameter values. We assume that the solution obtained when the algorithm converges accurately approximates x_{opt}.

Stages I, III, and IV rely on two algorithms, the algorithm to compute the overlap area map (Stage I) and the simulated annealing algorithm (Stages III and IV), that run mainly in the GPU. Indeed, the overlap area map of the preprocessing Stage I computed in the GPU is then transferred to the CPU to determine x_{ini} in Stage II. Stage III starts by determining several initial parameter values for the SA algorithm in the CPU and with the initial configuration x_{ini} in the CPU. Then, this information is transferred to the GPU. In there, x_{ini} is perturbed with a dispersive SA algorithm until a stopping criterion holds and x_0 is determined. Stage III ends by transferring x_0 to the CPU. Stage IV starts computing the parameter values for the more conservative SA algorithm in the CPU from the already obtained x_0. Again, the parameter values are transferred to the GPU, where the SA algorithm runs again until x_{opt} is obtained and then transferred back to the CPU.

Section 4.1 provides the details of the SA algorithm applied in Stages III and IV.

4.1. Simulated annealing algorithm

The dispersive and refining stages (Stages III and IV) are the clue elements of our algorithm because is where the SA algorithm takes place. This section describes them in detail.

Our SA algorithm has as input parameters: the number of disk-like services k, their radius r, the overlap area map $\mathcal{M}$, a maximal number of steps S to be done, and an initial configuration $\widetilde{x}$ ($\widetilde{x} = x_{ini}$ in Stage III, $\widetilde{x} = x_0$ in Stage IV) determining k disks with nonempty intersection with $\mathcal{P}$, i.e. $\widetilde{x} \in \overline{\mathcal{P}}^k$. The configuration x_{ini} obtained in Stage II fulfills this condition, and, as we will see, the algorithm guarantees it for x_0.

Starting from $\widetilde{x}$, a CUDA kernel executed by N threads computes in parallel N Markov independent chains of the SA of length L. Then, after the L iterations, the threads are synchronized to extract the best already obtained configuration among the N threads used to update $\widetilde{x}$. The kernel repeats this procedure at most S time-steps and stops as soon as a stopping criterion holds after $s \leq S$ synchronization steps. The obtained configuration $\widetilde{x}$ is postulated as the desired configuration, i.e. x_0 in Stage III or x_{opt} in Stage IV. To be able to generate the Markov chains, the kernel receives several data from the CPU such as the initial temperature T_0, an initial perturbation radius R_0, the radius-decreasing scheme, the temperature-decreasing scheme, the maximal number of steps S, the length L of the Markov chains and the minimal desired factor $\mathcal{F}$ of area covered. Moreover, it also receives the overlap area map directly from the GPU.

The rest of the section is organized as follows. We explain how are the Markov chains generated in CUDA in Section 4.1.1, how configurations are perturbed in Section 4.1.2, how the objective function is evaluated in the GPU in Section 4.1.3, and, finally, how is the initial temperature T_0 and perturbation radius R_0 are elected in Section 4.1.4.

4.1.1. Markov chains generation

Markov chains are generated in a per-thread level as it is explained in this section. Each thread generates several Markov chains of $L > 0$ iterations, each of them starting with a global optimal configuration $\widetilde{x}$ stored in shared memory. To generate the Markov chains, each thread identified by an integer id locally stores its current best solution $\widetilde{x}_{id}$, a current candidate x_{id} and their objective function values $\widetilde{o}_{id} = \overline{\mathcal{O}}(\widetilde{x}_{id})$ and $o_{id} = \overline{\mathcal{O}}(x_{id})$. Each Markov chain starts with $\widetilde{x}_{id} = \widetilde{x}$ and $x_{id} = \widetilde{x}$. At each iteration, the current configuration x_{id} is perturbed defining a new configuration x'_{id}, for which $o'_{id} = \overline{\mathcal{O}}(x'_{id})$ is computed. If x'_{id} is better than the current best solution $\widetilde{x}_{id}$, i.e. $o'_{id} > \widetilde{o}_{id}$, $\widetilde{x}_{id}$ and $\widetilde{o}_{id}$ are updated and set to x'_{id} and o'_{id}, respectively. Moreover, x_{id} is also set to x'_{id} if it is probabilistically accepted even though it does not improve x_{id}. A worst solution x'_{id} is accepted if a generated random value $p \in (0,1)$ is smaller than $\exp((o'_{id} - o_{id})/T)$. Parameter T is the SA temperature which starts being T_0 (see Section 4.1.4) and diminished at each iteration by using a multiplicative scheme in Stage III and a logarithmic scheme in Stage IV:

$$\text{Stage III: } T = \alpha_T T = \alpha_T^{n_{it}} T_0 \qquad \text{Stage IV: } T = \frac{T_0}{\log(n_{it}+1)},$$

where n_{it} represents the total number of iterations already done, i.e. at the iteration $l \leq L$ of the synchronization step s it is calculated as $n_{it} = (s-1)L + l$. Therefore, the more iterations are done the lower is the probability to accept a worse solution.

Every L iterations, the threads are synchronized to compute new Markov chains starting from the best solution obtained among all of them. The information generated in each Markov chain is stored locally in each thread and is not accessible for the others threads. Only variables stored in global and shared memory are accessible for all the threads. Hence, we use a real variable $\widetilde{o}$ stored in shared memory to maintain the maximum among all the $\widetilde{o}_{id}$ values locally stored in the threads. CUDA has a read–write atomic maximum operation that allows obtaining a global maximum. Read–write atomic operations guarantee that once one thread reads the value stored in the variable, no other

thread reads it until the first one has finished writing in that variable. Therefore, once a thread has finished its L iterations of the current step, it performs an atomic maximum operation in $\widetilde{o}$. Then, it waits until all the threads have finished doing this action by using a synchronization point. Once all the threads have finished, $\widetilde{o}$ stores the best $\widetilde{o}_{id}$ among all the threads. Now, we have to determine which configuration $\widetilde{x}_{id}$ leads to $\widetilde{o}$ and update $\widetilde{x}$ accordingly. With this aim, each thread checks whether its $\widetilde{o}_{id}$ coincides with $\widetilde{o}$. If it is so, it stores its id, in another variable $\widetilde{id}$ also stored in shared memory, and waits until the rest of the threads have finished doing it. When all the threads have done, $\widetilde{id}$ stores a thread id whose $\widetilde{o}_{id}$ equals $\widetilde{o}$. Then, this thread id stores its $\widetilde{x}_{id}$ in $\widetilde{x}$ and the rest await it. At this point, it is checked, according to a stopping criterion, whether a new step is needed. When no further steps are required, the desired solution is assumed to be $\widetilde{x}$.

The process ends when S steps, with L iterations each, have already been done. Or when the SA algorithm has obtained a good enough solution. That is when the area covered by the current configuration reaches a certain level concerning the maximal potential area to cover $k\pi r^2$ specified by the minimal desired factor of area covered $\mathcal{F}$. Indeed, the *stopping criterion* can be mathematically formulated as follows:

$$s = S \;\text{ or }\; \widetilde{o}_s \geq \mathcal{F} \cdot k\pi r^2,$$

where $s \geq 1$ and $\widetilde{o}_s$ denotes the value of $\widetilde{o}$ once the sth step has finished. Accordingly, $\widetilde{o}_0$ is $\overline{\mathcal{O}}(x_{ini})$ in Stage III and $\overline{\mathcal{O}}(x_0)$ in Stage IV.

The initial configuration and its objective function value, initially stored in $\widetilde{x}$ and $\widetilde{o}$, are input parameters of the kernel. And indeed, the final values of $\widetilde{x}$ and $\widetilde{o}$ are the outputs of the kernel. Hence the initial and final values stored in $\widetilde{x}$ and $\widetilde{o}$ have to reachable from the CPU. With this aim, they are stored in global variables, concretely, in a global memory real array $\widetilde{x}_g$ of size $2k$ and a real global variable $\widetilde{o}_g$. The initial values of $\widetilde{x}_g$ and $\widetilde{o}_g$ are set from the CPU and transferred to shared memory in $\widetilde{x}$ and $\widetilde{o}$ by using the standard strategy used to transfer information from global to shared memory. Finally, the final values of $\widetilde{x}$ and $\widetilde{o}$ are similarly transferred to global memory into $\widetilde{x}_g$ and $\widetilde{o}_g$ and finally read from the CPU.

Finally we should mention that the strategy used to obtain $\widetilde{o}$ forces calling the kernel by using a single CUDA block. The threads of a CUDA kernel are grouped in blocks according to the programmer's indications. The threads in the same block run in the same core and can be synchronized in a kernel execution. However, there is no way to synchronize threads of different blocks within a kernel. The programmer cannot control how blocks are distributed among the cores or the sequence they are processed. Therefore, to be properly obtained $\widetilde{x}$, all the threads must belong to the same CUDA block.

4.1.2. New configuration generation

Generating a new configuration is a clue aspect of the SA algorithms. An ad hoc process is required to obtain a fast and robust SA algorithm. In this section, we explain how to create new configurations.

The new configurations are generated trying to avoid overlapping pairs of disks because they are undesirable according to our purpose and the defined objective function. It is reasonable because we aim at maximizing the area of $\mathcal{P}$ covered by k disk-like services by using an objective function that sums the area of $\mathcal{P}$ covered by each disk and subtracts a pretty accurate estimation of the area of $\mathcal{P}$ covered by each pair of disks. Therefore, given a configuration x, the perturbation scheme attempts to separate the overlapping disks of x while accepts only disks intersecting $\mathcal{P}$, that is, centered at a point of $\overline{\mathcal{P}}$.

To compute a perturbation of x, a random point $x'_i \in D(x_i, R_i) \cap \overline{\mathcal{P}}$ is generated for each center x_i of x. Devroye (1986) (page 234) explain how to obtain a uniformly distributed random point in $D(x_i, R_i)$. Accordingly, two real random values v and v' in $(0, 1)$ are generated by using *cuRAND*[1] CUDA-library and the perturbed center x'_i becomes:

$$x'_i = x_i + (\rho\cos\alpha, \rho\sin\alpha) \qquad \text{where} \qquad \alpha = 2\pi v \quad \text{and} \quad \rho = R_i\sqrt{v'}.$$

[1] https://developer.nvidia.com/curand.

Once $x'_i \in D(x_i, R_i)$ has been obtained, it is checked whether $D(x'_i, r)$ intersects $\mathcal{P}$, i.e. $x'_i \in \overline{\mathcal{P}}$. It is done by using the overlap area map according to the details given in Section 4.1.3. If the area obtained is not 0, x'_i is accepted. But, if it is 0, it is assumed that $D(x'_i, r)$ does not intersect $\mathcal{P}$, the perturbation is rejected and x_i is not updated.

Value R_i depends on x_i and is obtained as the product $\beta_i R$, where R is the SA *perturbation radius*, and β_i is a *refactoring value* that is obtained from an approximation of the area of P contained in the disk D_i and not contained in any other disk D_j. The refactoring value is used to determine how good is the disk D_i. That is, the better D_i, the less it has to be perturbed. Next, we explain how R and β_i are computed.

- The perturbation radius R is initialized to the CPU-given value R_0 (see Section 4.1.4). Then, it diminishes at each synchronization step s of the SA algorithm by using a multiplicative scheme in Stage III and a logarithmic scheme in Stage IV:

  $$\text{Stage III: } R = \alpha_R R = \alpha_R^s R_0 \qquad\qquad \text{Stage IV: } R = \frac{R_0}{\log(s+1)}.$$

  Hence, the perturbation radius diminishes exponentially in Stage III and rapidly tens to 0. Whereas in Stage IV, it decreases faster at the very beginning but tends to 0 much slower.
- To determine β_i we use $ownArea(D_i)$, the area of P contained in the disk D_i and not contained in any other disk $D_{i'}$. This area can be roughly approximated by:

  $$ownArea(D_i) \approx Area(I_i) - \sum_{\substack{i'=1 \\ i'\neq i}}^{k} \overline{Area}(I_i \cap I_{i'})$$

  Note that the principle of inclusion–exclusion should be applied using the intersections of three or more disks to calculate it accurately. These intersections, for time-consuming reasons, are obviated in the approximation. Then, the $ownArea(D_i)$ value is not necessarily positive. In fact, $ownArea(D_i) \in [-(k-1)\pi r^2, \pi r^2]$. This value is used to compute the refactoring value $\beta_i \in [0.01, 2]$ by using the formula:

  $$\beta_i = \max\left(0.01, \min\left(2, 1 - \frac{ownArea(D_i)}{\pi r^2}\right)\right).$$

  Observe that when the minimum is 0, disk D_i is fully contained in $\mathcal{P}$ and does not overlap any other disk D_j hence, it covers the maximum area and, in principle, does not needs to be perturbed. But to let the SA continue, a small perturbation is still allowed.

4.1.3. Objective function evaluation

Threads evaluate the objective function $\overline{\mathcal{O}}$ for a specific configuration $x = (x_1, \ldots, x_k)$ by taking into account the aspects mentioned in this section.

Each thread extracts the k values approximating $A_i = Area(I_i)$ from the overlap area map, and exactly computes the area of overlap of the $k(k-1)/2$ pairs of disks. To obtain the k values A_i, as fast as possible, we make use of the texture fetching CUDA-functions after binding the overlap area map computed in Stage I to a texture. We set the texture fetching function so that when the value of a given center x_i is fetched, it returns:

- 0 if x_i does not fall in the texture boundaries, i.e. $x_i \notin \overline{B}$
- the bilinear interpolation of the values stored in the neighborhood of x_i, otherwise.

This texture is fetched during the objective function evaluation and also when a center is perturbed to check whether the new disks intersect $\mathcal{P}$ or not. During the computation of $\overline{\mathcal{O}}(x)$, the fetched values A_i are accumulated in a local real variable *areaSum* and also stored in the ith position of a local real array of size k, a, i.e. $a[i] = A_i$.

Thereafter, the area of overlap $\overline{Area}(I_{i_1} \cap I_{i_2})$ is computed for each pair of disks D_{i_1}, D_{i_2} satisfying $1 \leq i_1 < i_2 \leq k$. These overlapping areas are subtracted from *areaSum* and from the i_1th and i_2th positions of the

a array. Once all the pairs have been processed, $areaSum = \overline{\mathcal{O}}(x)$ and $a[i]$ contains the $ownArea(D_i)$ value needed to generate a new configuration as explained in Section 4.1.2.

Note that the threads store this area vector for each of the configurations they locally maintains while generate a Markov chain, i.e. they maintain $\tilde{a}_{id}$, a_{id} and a'_{id}. Moreover, when the threads are synchronized at after L iterations, the $\tilde{a}_{id}$ vector of the thread defining the new value of $\tilde{x}$ has to be also transferred to a shared memory float vector, $\widetilde{a}$, to be used at the next step. When the SA ends, this vector is transferred to a global memory array and then to the CPU. The values obtained at the end of Stage III are used to initialize the vector at Stage IV. At Stage III it is initialized to 0.

4.1.4. Parameter values determination

SA algorithms lead to globally optimal solutions, but they need to use a suitable perturbation strategy and the proper parameter values. In this section, we explain how to set the parameter values.

The initial values T_0 and R_0 for the temperature and perturbation radius respectively of the SA are set automatically in the CPU as follows:

Commonly, the initial temperature T_0 of a SA algorithm is an approximation of the difference between the maximum and the minimum values of the objective function, see Section 2.2. In our case, this difference corresponds to the minimum between $k\pi r^2$ and the area of $\mathcal{P}$. But, taking into account that, in standard cases, the value $k\pi r^2$ is not much greater than the area of $\mathcal{P}$, we initialize T_0 to $k\pi r^2$ in both stages.

The radius R plays a crucial role in SA algorithms, but there is no general indication to determine it. What is known is that to have a dispersive SA, R must be much bigger than when a more conservative SA is desired. In general, the worst a configuration x is, the bigger R should be. The dispersive stage, Stage III, starts from x_{ini}, which has all the disks centered at the same point, and it is expected to scatter the disks all over the domain. Hence, we make the initial radius proportional to the longest edge of the bounding-box $\mathcal{P}$. However, it cannot be too big because disks have to move in a somehow controlled manner. Therefore, in the dispersive stage, R_0 is set by $0.1\mathcal{L}$, being $\mathcal{L}$ the length of the longest edge of the bounding-box of $\mathcal{P}$. Meanwhile, the refining stage, Stage IV, starts from the configuration obtained at the dispersive stage x_0 and only refines it. In this case, R_0 is still set proportional to $\mathcal{L}$, but also to the improvement possibilities of x_0, which are execution-dependent. The improvement possibilities of x_0 are complementary to the quality of x_0 which can be quantified as the ratio of the configuration objective-function value divided by the maximum of the objective function. Accordingly, in the refining stage, the perturbation radius R_0 is initialized by $0.1\mathcal{L}(1 - \overline{\mathcal{O}}(x_0)/(k\pi r^2))$. In Summary, we initialize R_0 as follows:

$$\text{Stage III:} \quad R_0 = 0.1\mathcal{L} \qquad \text{Stage IV:} \quad R_0 = 0.1\mathcal{L}\left(1 - \frac{\overline{\mathcal{O}}(x_0)}{k\pi r^2}\right).$$

4.2. Complexity analysis

One of our aims was to obtain a fast algorithm, in this section, we provide the theoretical complexity analysis of the algorithm to demonstrate, theoretically, that we succeed.

To provide the complexity analysis of our approach, we have to analyze its four stages. Coll et al. (2019) study the complexity of Stage I. Stage II only stores and traverses the overlap area map generated in Stage I, and, hence, requires $O(HW)$ space and time. In stages III and IV, the CPU only transfers the configuration information to and from the GPU and calls the kernel. Its CPU time and space complexity is $O(k)$. Then, the kernel performs its work in the GPU. Therefore, along this section, we provide the space and time complexity analysis of the kernel called in Stages III and IV.

Let us start with the space complexity. During the kernel execution, the GPU stores: the $H \times W$ real overlap area map in texture memory, T_0, R_0, k, r, the $2k$ real values defining the global optimal solution $\widetilde{x}_g$, its objective function $\widetilde{o}_g$ and also its own area vector $\widetilde{a}_g$ which is defined by k real values. The $3k + 1$ values associated to the configuration are stored in global memory, meanwhile the constant values are stored in constant memory. Indeed, the kernel also needs $3k + 1$ float values in shared memory to store $\widetilde{x}$, $\widetilde{o}$ and $\widetilde{a}$. Finally, each of the N thread stores its current best solution $\widetilde{x}_{id}$, its current candidate x_{id}, the new perturbation configuration x'_{id} and their own area vectors and objective function values, i.e. $\widetilde{a}_{id}$, a_{id}, a'_{id}, $\widetilde{o}_{id}$, o_{id} and o'_{id}. They use $9k + 3$ float values, $3k+1$ values for each configuration. Moreover, each thread also needs some other auxiliary variables such as those needed to compute the two-disks overlap area, the threshold evaluation, the random values extraction or the R and T values, which represent $O(1)$ values. Hence, the GPU memory requirements of this algorithm are

$$O(HW_t + k_g + k_s + k_l + 1_c),$$

where subindices t, g, s, l and c indicate texture, global, shared, local and constant memory, respectively.

Concerning the time complexity, there are several factors to analyze. An important aspect to consider is the number of accesses to the global memory to be done. In our case, most of the information is in shared, local, texture, or constant memory and have fast accesses. Threads only read and write to global memory at the very beginning and the end of the kernel call when $\widetilde{x}_g$, $\widetilde{a}_g$ and $\widetilde{o}_g$ are transferred from global to shared memory first, and from shared memory to global memory when the SA algorithm ends. Afterwards, threads read information from shared or local memory.

The most time-expensive issue of our algorithm is obtaining the best solution found up at each step, every L iterations. At this point, the kernel performs k atomic operations at $\widetilde{o}$. In principle, atomic operations can make the algorithm behave sequentially. However, this not always happens because threads may reach the atomic operation at different times. As we show in the experimental results, Section 5, this is our case. The synchronization is worth it and does not affect the running time.

On the other hand, our algorithm performs $O(k)$ reads and writes to global memory to transfer the information from and to the shared memory. Shared memory is read S times by each of the N threads when $\widetilde{x}$, $\widetilde{a}$ and $\widetilde{o}$ are read at the beginning of each step. Meanwhile, at each step, only one thread overwrites its $3k+1$ values. It gives $O(kSN)$ reads and $O(kS)$ writes to shared memory. Whereas, at each iteration, each thread does one atomic operation to shared memory, which gives $O(SN)$ atomic operations to shared memory. Finally, concerning local memory, at each iteration each thread generates a new configuration with its associated information x'_{id}, o'_{id} and a'_{id}. Determining it takes $O(k^2)$ time and requires $O(k^2)$ reads and writes to local memory apart from $O(k)$ fetches to texture and $O(k)$ reads to constant memory. Since, in the worst-case, LS iterations are done, this per-thread analysis leads to $O(k^2LS)$ accesses to local and constant memory and $O(kLS)$ to texture memory. Moreover, the N threads perform $O(k^2LSN)$ accesses to local memory and $O(kLSN)$ to constant or texture memory. Therefore, the time complexity of the kernel used in Stages III and IV is

$$O(k\tau_g + kSN\tau_s + SN\tau_s^a + k^2LSN\tau_l + k^2LSN\tau_c + kLSN\tau_t)$$

where: τ_g, τ_s, τ_l, τ_c and τ_t represent the time needed to read or write to global, shared, local, constant or texture memory, respectively, and τ_s^a to perform an atomic operation into shared memory.

Note that the local memory requirements of our algorithm are considerably large. When local memory is not sufficient, global memory is automatically used, notoriously slowing the algorithm performance. Hence, using too many threads may slow the algorithm running time because the local memory requirements depend on the number of threads, N. If N is too big, part of the variables that should use local memory will use global memory. Consequently, the correspondent part of the $O(k^2LSN)$ accesses to local memory will directly be transformed

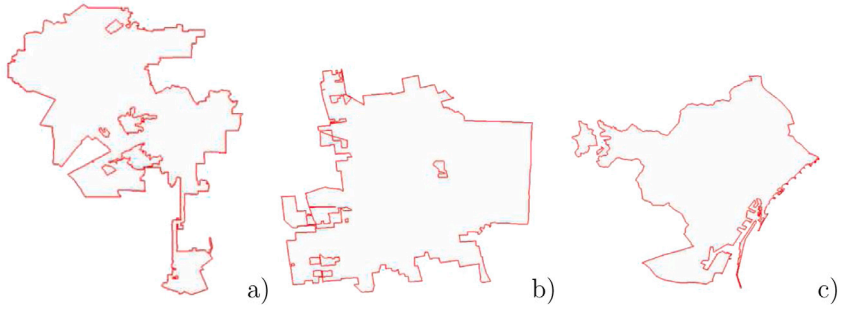

Fig. 2. Polygonal domain.

to global memory accesses, and the running time of our algorithm will notably increase.

In conclusion, the number of operations done by the algorithm depends linearly on the number of threads N, the length of the Markov chains L and the number of steps done S, and it is quadratic on the number of disks k.

5. Experimental results

In this section, we analyze the performance of our proposal, not only in terms of running times but also the quality of the solutions. We provide the experimental settings in Section 5.1. Next, we present the interface developed to deal with the problem, test the method and visually inspect the obtained solutions in Section 5.2. In Section 5.3, we discuss the influence of the SA algorithm parameters on the running times and the obtained solution. In Section 5.4, we analyze the influence of the original problem parameters such as the polygonal domain or the number or radius of the disk-like facilities considered. Finally, in Section 5.5 we evaluate the goodness of solutions obtained.

5.1. Experimental settings

The implementation language used to program the algorithm presented has been C++ with Cuda C for the parallel parts, and OpenGL for the visualization process. The algorithm has run in an Intel Core i7-7700 CPU @3.6 GHz with 32 GB of RAM and a GPU Nvidia GTX 1080 6 GB.

During this experimental analysis, we considered three real-polygonal chains[2] representing the cities of Los Angeles, the second-largest city in the United States, Dublin from Ohio, and Barcelona from Catalunya-Spain, see Fig. 2 (a). Los Angeles (**LA**) has a long-thin corridor connecting a big upper part to a south smaller one. The polygon boundary contains 7943 edges determining one single component with 8 holes. Dublin (**Dublin**) has 946 edges delimiting a single component with 15 holes. Barcelona (**Bcn**) has no holes but 6 unconnected regions delimited by 1801 edges, 1506 of them define the biggest one. All the polygonal domains are scaled and translated to the $[-1,1]\times[-1,1]$ square.

In this section, we consider disk-like services of scaled radius equal to $0.05, 0.1, 0.15, 0.2, 0.25$ and 0.3. The difficulty of obtaining an optimal location for k disks depends on many different factors, but one of the most relevant ones is the portion of $\mathcal{P}$ that the disks can cover. Generally speaking, if the area covered by the k disks is much smaller than the area of the domain $\mathcal{P}$, there usually exist many different optimal locations for the k disks. Contrarily, it is extremely complicated to obtain an optimal configuration for the k disks when their area is close to, or larger than, the area of $\mathcal{P}$. Hence, in most of our experiments, we have not directly prefixed the number of circles k. We computed k so that the area of the k disks represents a prefixed percentage, $\%p$, of the area of the domain $\mathcal{P}$. That is, k is computed as

$$k = \left\lceil \frac{\%pArea(\mathcal{P})}{\pi r^2} \right\rceil.$$

We considered the following percentages $\%p$, that cover a wide range of scenarios: $15\%, 45\%, 60\%, 75\%, 90\%, 100\%$ and 115%. Taking into account the mentioned radius and percentages, the number of disks covering LA varies from 1 to 186, Dublin from 1 to 316 and Bcn from 1 to 205. Nowadays, Bcn contains 346 petrol stations, 106 pharmacies, 21 hospitals, and 888 cell towers of 4 different companies (the company with more cell towers owns 374 towers). Therefore, the number of considered disks cover a wide range of real problems. Moreover, we also provide some time and experimental results for settings with up to 2000 disks of radius 0.01.

Regarding the parallelization issues, we have run our algorithm with the number of threads N equal to 512, 256 and 128. As synchronization frequency, L, we considered $25, 50, 100$ and 150. The maximum number of steps, S, allowed in Stage III (denoted by S_{III} in this section) varies among $15, 30, 45, 60, 75$ and 90, and in Stage IV, S_{IV}, among 50, 150, 250, 350, 450, 550 and 650. The maximum expected area for the termination criterion of Stages III and IV is the 99% of the area of the k disks. In all the execution the multiplicative factor used in the decreasing-scheme for the temperature and the perturbation radius in Stage III are $\alpha_T = 0.99$ and $\alpha_R = 0.975$. Finally, the regular grid of the overlap area covering $\overline{B}$ has the cell edges of size equal to 0.0025.

To further validate the goodness of the algorithm, in Section 5.5, we run it with the unit square domain and considering 40, 50, 70 and 100 disks of radius 0.105466, 0.093089, 0.078427, 0.064813, respectively. Stoyan and Patsuk (2010) prove that these radii are the minimal radius needed to cover the unit disk with the amounts of disks considered.

5.2. Interface

To visually inspect the obtained solution and test the presented method, we have developed the interface shown in Fig. 3. It allows reading the polygonal domain from a file, specifying some basic parameters related to the problem settings and tuning, if desired, some advanced parameters related to the SA algorithm.

As basic parameters, the user can specify the number of disk-like services, their radius and the minimum percentage of the area of the k disks to achieve before stopping the SA algorithm. Concerning the more advanced parameters related to the SA, the user can specify: the number of steps done in the dispersive Stage III, the ones done in the refining Stage IV, the synchronization frequency, the number of threads used, and, if desired, the initial perturbation radius R_0 of each stage.

After selecting the polygonal domain and setting the basic parameters, the user presses the *Start* button, and the algorithm runs. When the SA algorithm ends, the right-bottom corner shows some relevant information, such as the execution time, the total area covered, the polygonal domain area, the objective function value and the centers

[2] Obtained from OpenStreetMap! https://www.openstreetmap.org.

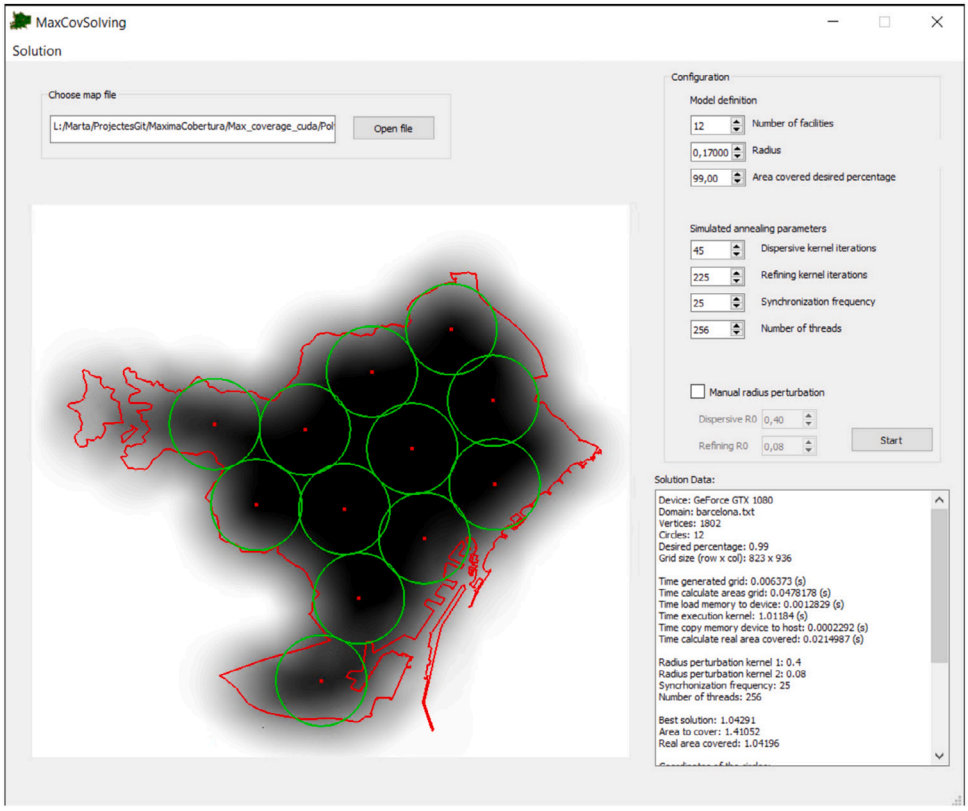

Fig. 3. Interface used to visually inspect the solutions and test the methods.

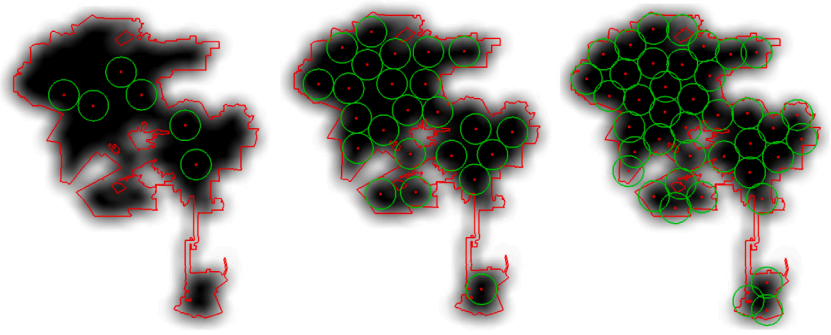

Fig. 4. LA domain covered by disks of radius 0.1.

defining the optimal configuration. Meanwhile, the main widget shows the obtained solution by painting a red polygonal chain representing the boundary of the polygonal domain, green circumferences delimiting the disk-like services, and red squares representing their centers. The image background contains the overlap area map obtained by painting its cell centers on a grayscale. The darker a point, the bigger the overlapped area between the disk centered at the point and $\mathcal{P}$ is. Hence, white points are centers of disks that do no overlap $\mathcal{P}$.

The interface also allows moving around the domain and zooming in and out to inspect the obtained solution or export the disk centers to a file by using the *Solution* bar menu (top-left corner).

5.3. Parallel SA issues

In this section, we comment on the influence of the number of threads N, the synchronization frequency L, and the maximum number of steps S done in the kernels run in Stages III and IV. We have run the tests with all the maps and radii, but here we present the results obtained with LA Domain considering disk-like services of radius 0.1 whose total area represents from the 15% to the 115% of LA. The results obtained with the other domains and radii do not differ significantly from the ones discussed in this section and represented from Figs. 4 to 6.

Fig. 4 exemplifies the considered settings showing LA covered by disk-like services of radius 0.1. Concretely it is first covered with six circles which cover the 15% of LA area and then by 24 and 40 disks whose global area corresponds to the 60% and 100% of LA, respectively. In the right-most figure, the area of the disks coincides with that of LA, but the disks overlap and parts of LA remain uncovered.

Fig. 5 presents the results obtained varying the maximal number of steps S_{III} and S_{IV} permitted in Stages III and IV, respectively, and fixing L to 25 and N to 256. To analyze the influence of S_{III}, it varies

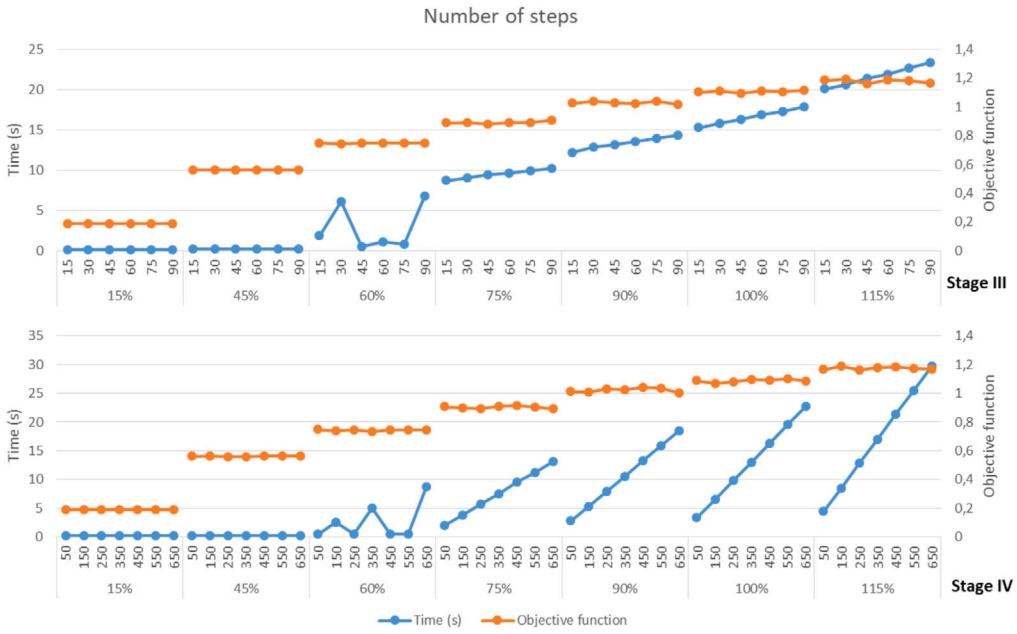

Fig. 5. Influence of the maximum number of steps in Stage III and Stage IV in LA with S_{III} from 15 to 90, S_{IV} from 50 to 650 and $r = 0.01$.

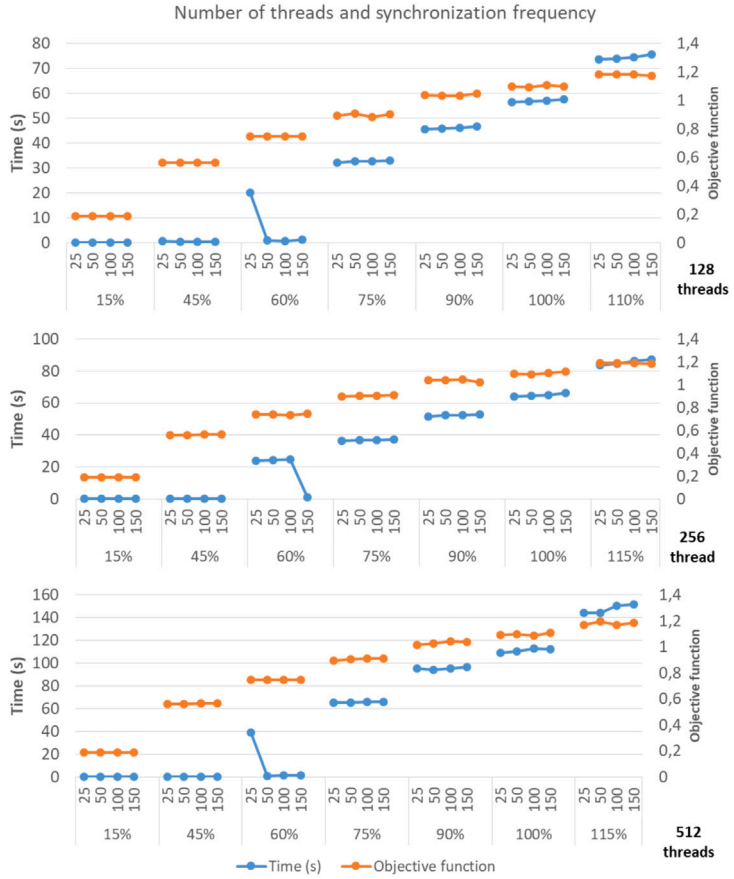

Fig. 6. Influence of the number of threads N and the synchronization frequency L on the running times and the objective function value, considering $N = 128$, 256 and 512, $L = 25$, 50, 100 and 150, LA domain, $r = 0.01$, $LS_{III} = 4525$ and $LS_{IV} = 27\,025$.

from 15 to 90 while maintaining $S_{IV} = 450$. Considering more than 90 steps in Stage III with a multiplicative scheme for temperature and perturbation radius makes no sense because the acceptance probability is almost zero. On the other hand, to analyze the influence of the number of steps in Stage IV, S_{IV} varies from 50 to 650 while fixing S_{III} to 45. Stage IV uses logarithmic schemes permitting much more iterations.

The figure exemplifies the two very different scenarios with a transition zone in between. When there exist many solutions to the problem, 15% or 45%, S_{III} and S_{IV} have no effect in the objective function

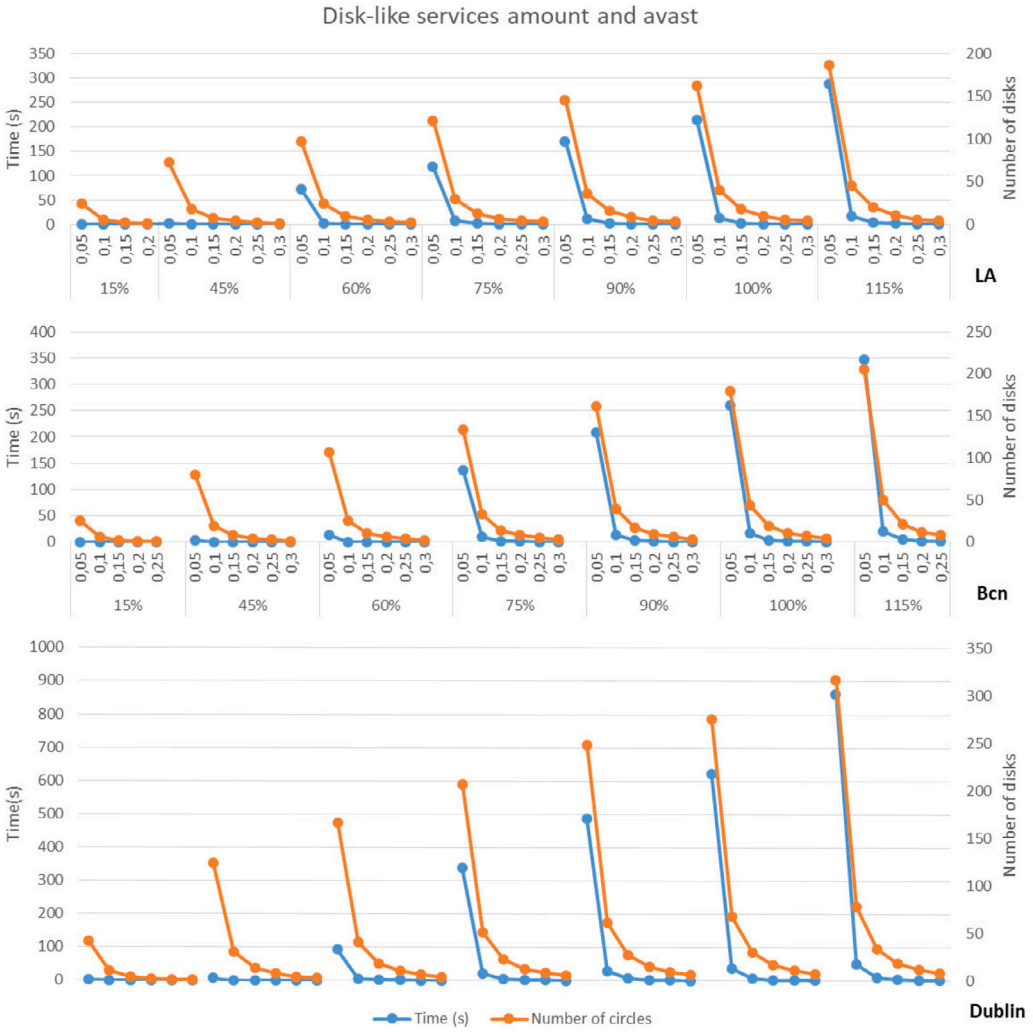

Fig. 7. Running times and number of disks used in LA, Bcn and Dublin for $r = 0.05$, 0.1, 0.15, 0.2, 0.25 and 0.3.

value nor the running times. The termination criterion rapidly stops the SA algorithm, and the problem is optimally solved. Things change when more disks exist. In the 60%, the running times are irregular because not all the solutions work. In this case, one circle has to be in the small southern part of the domain, see Fig. 4, and it depends on the execution that a disk is early thrown there or not. Adding more disks complicates obtaining a good enough solution, i.e. with an objective function value that reaches the 99% of the total area of the disks. Hence, the termination criterion of the SA actuates later or does never actuate. In these cases, the more steps the stage does, the bigger the time used. According to the obtained results, we consider 45 iterations a good starting point for Stage III and 450 for Stage IV. As we mentioned, the interface allows varying the number of iterations of each stage.

Fig. 6 presents, in different graphics, the running times and the objective function values obtained with $N = 128$, 256 and 512. In each graph, the number of circles and the synchronization frequency varies. Parameter L, varies among $25, 50, 100$ and 150 for each number of circles. To fair compare the results obtained with the different L values, we fixed the total amount of steps, LS_{III} to 4525 in Stage III and LS_{IV} to 27 025 in Stage IV. The values of S_{III} and S_{IV} are adjusted to achieve these values.

This figure reflects, again, the two different scenarios mentioned before with the transition zone in between. In this case, the running times reveal the scenarios. For the 15% and 45%, the running times do not exceed the 0.2 (s) for 128 or 256 threads, or 0.4 (s) for 512 threads. However, for the 60% with a fixed number of threads, the running time notoriously varies on the execution. It varies from 1 to 20 (s) for 128 or 256 threads or from 1 to 45 (s) for 512 threads. The reasons are the randomness of the algorithm and the specific characteristics of the optimal solution. For the 75%, the execution times are always bigger than 32, 36 or 65 (s) for 128, 256 or 512 threads, respectively. The more threads used, the greater the running time is. Concretely, using 512 threads has incremented the running time up to 76 (s), becoming the 109% of the time needed for the 128 threads. Meanwhile, using 256 threads has increased the running times at most 11 (s), the 18% of the running times obtained with 128 threads. These 76 or 11 (s) correspond to the 115% setting. Finally, concerning L, since SL is fix, the increment of time provides from the atomic operations and is not relevant. It represents at most 0.2 (s) for 128 threads, 1.7 (s) for 256 threads and up to 6 (s) for 512 threads. Concerning the value of the objective function, the variations do not exceed the 2% of its value, and the most important differences appear with 128 threads.

Hence, we can conclude that using 512 threads is not a good option, at least with our graphics card, probably because the shared memory needed exceeds the existent one and global memory supplies the excess. Considering 128 threads could be an option, but sometimes the objective function value is improvable. The regularity of the objective function and the small increment in the running times obtained for the 256 threads make considering $N = 256$ the best option. Concerning L, it is expectable and corroborated that the more often the threads are synchronized, the better the solutions become. Restarting the Markov chains from a better approximation more often leads faster to the optimal solution.

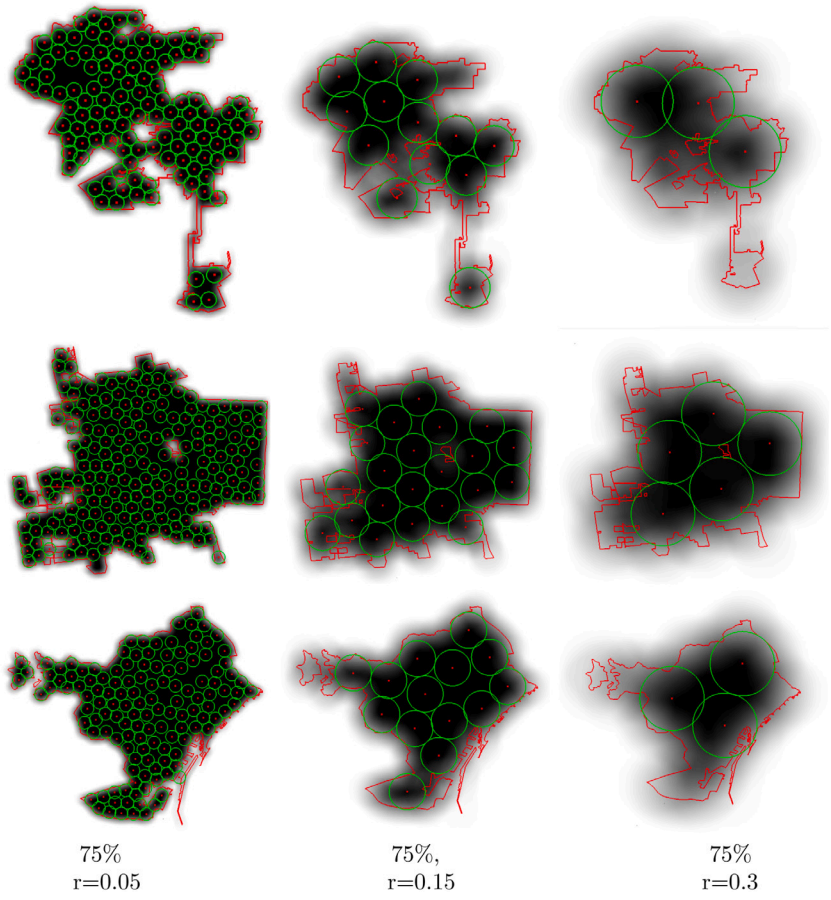

Fig. 8. Solution for LA, Dublin and Bcn considering disks whose area represents the 75% of the area of $\mathcal{P}$ for $r = 0.05, 0.15$ and 0.3.

5.4. Original problem parameters

In this section, we analyze the influence on the running times of the domain, the amount of the disk-like services, and their avast, i.e $\mathcal{P}$, r and k. First, we fix r and make that the area of the k disks equals a % of the area of $\mathcal{P}$, as in the previous section. These results are presented in Figs. 7 and 8. Then, we fix the number of disks k and the scaled-radius r and solve the same problem in each map, see Fig. 9 for the obtained results.

Fig. 7 presents, in a graphic per map, the running times (blue) and the number of disks (orange) used in each of the considered settings. The algorithm ran with $S_{III} = 45$, $S_{IV} = 450$, 256 threads and 25 iterations at each synchronization step.

Analyzing the figure, we can see how the number of disks needed to cover Bcn and LA is quite similar and the running times too. Neither Bcn nor LA is an easy domain to cover because they have long narrow parts that are difficult to handle. Meanwhile, the number of circles involved in Dublin are bigger and hence, even though the shape of Dublin makes things easier, the running times increase notoriously because the number of used disks is much bigger. The running times of Dublin become 860 (s) when 316 circles have to be located, in the case of Bcn with 205 circles the algorithm uses 346 (s), and LA with 186 disks requires 287 (s).

Fig. 8 shows solutions for LA, Dublin and Bcn corresponding to 75% for $r = 0.05$, 0.15 and 0.3.

Fig. 9 presents several solutions for LA, Dublin and Bcn with 10, 175, 275 and 2000 disks of radius 0.02, 0.05, 0.03 and 0.01, respectively.

The running times needed to obtain these solutions are presented in Fig. 10. Indeed, it presents the running times needed to cover LA, Dublin and Bcn with from 10 to 275 disks of radius varying from 0.3 to 0.03, respectively. The time needed for $k = 2000$ and $r = 0.01$ is of 214 (s) in Dublin, 207 (s) in LA and 24364 (s) in Bcn.

Looking at Fig. 10 we can see how the running times for the six first cases, i.e. for $k \leq 175$, are very similar. While, for the last two cases and $k = 2000$, the running times differ a lot. These differences on the running times is due to the fact that 200 disks with $r = 0.04$ have almost the 80% of the area of LA, the 70% of the area of Bcn, but only the 46% of the area of Dublin. The 275 disks with $r = 0.03$ have overall an area equal to the 60%, 55% and 36% of LA, Bcn and Dublin, respectively. The same happens with 2000 disks, some of the components of Bcn are completely covered, however, the disks do not intersect. Obtaining an optimal solution when the area of the disks covers an important part of the area of the domain is time costly. Hence, the obtained running times corroborate what we already mentioned: when the k disks left a lot of uncovered domain and there exist many different solutions. The running time can notoriously diminish because the termination criterion actuates soon. In all these cases, the algorithm run with $S_{III} = 45$, $S_{IV} = 450$, $L = 25$ and 256 threads except for the case of 2000 circles in which it uses 128 threads.

From these results, we can conclude that the algorithm is fast, scalable and that the running time mainly depends on the number of circles and on the part of the domain they can cover, but not on the domain.

5.5. Solutions goodness

This section proves the goodness of the obtained solutions. To evaluate the goodness of our objective function, we computed the area covered by the solution by using Monte Carlo's method. We generate random points and count how many are both in $\mathcal{P}$ and the union of the disks. Then we compared this area value obtained with the value of

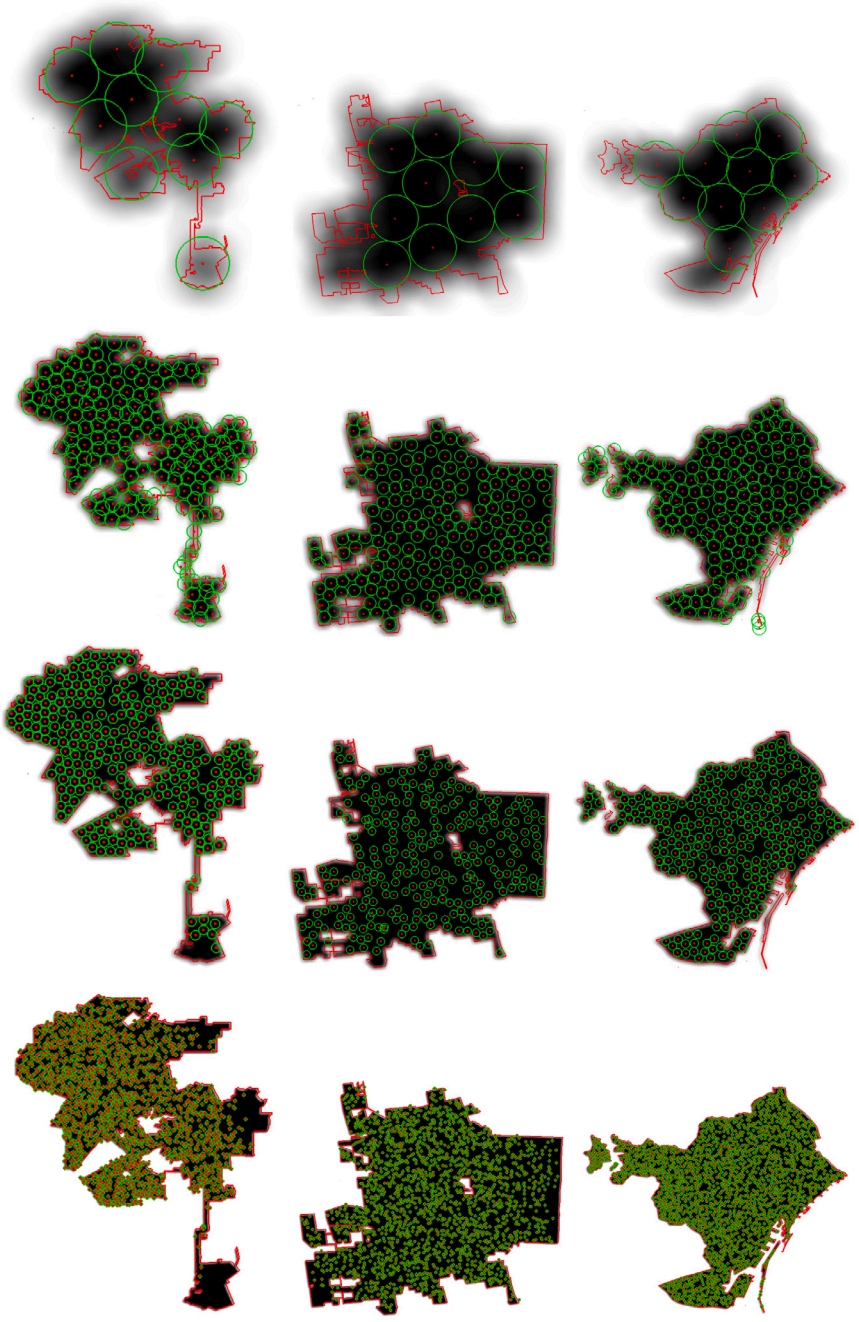

Fig. 9. LA, Dublin and Bcn covered by 10, 175, 275 and 2000 disks of $r = 0.2$, 0.05, 0,03 and 0.01, respectively.

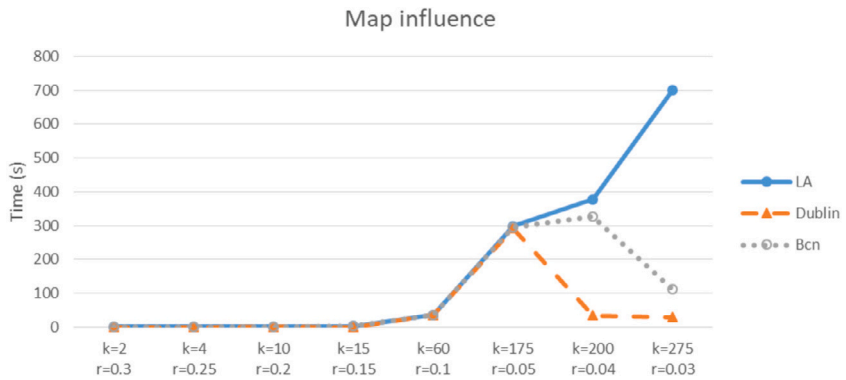

Fig. 10. Running times for LA, Dublin and Bcn with k and r fixed.

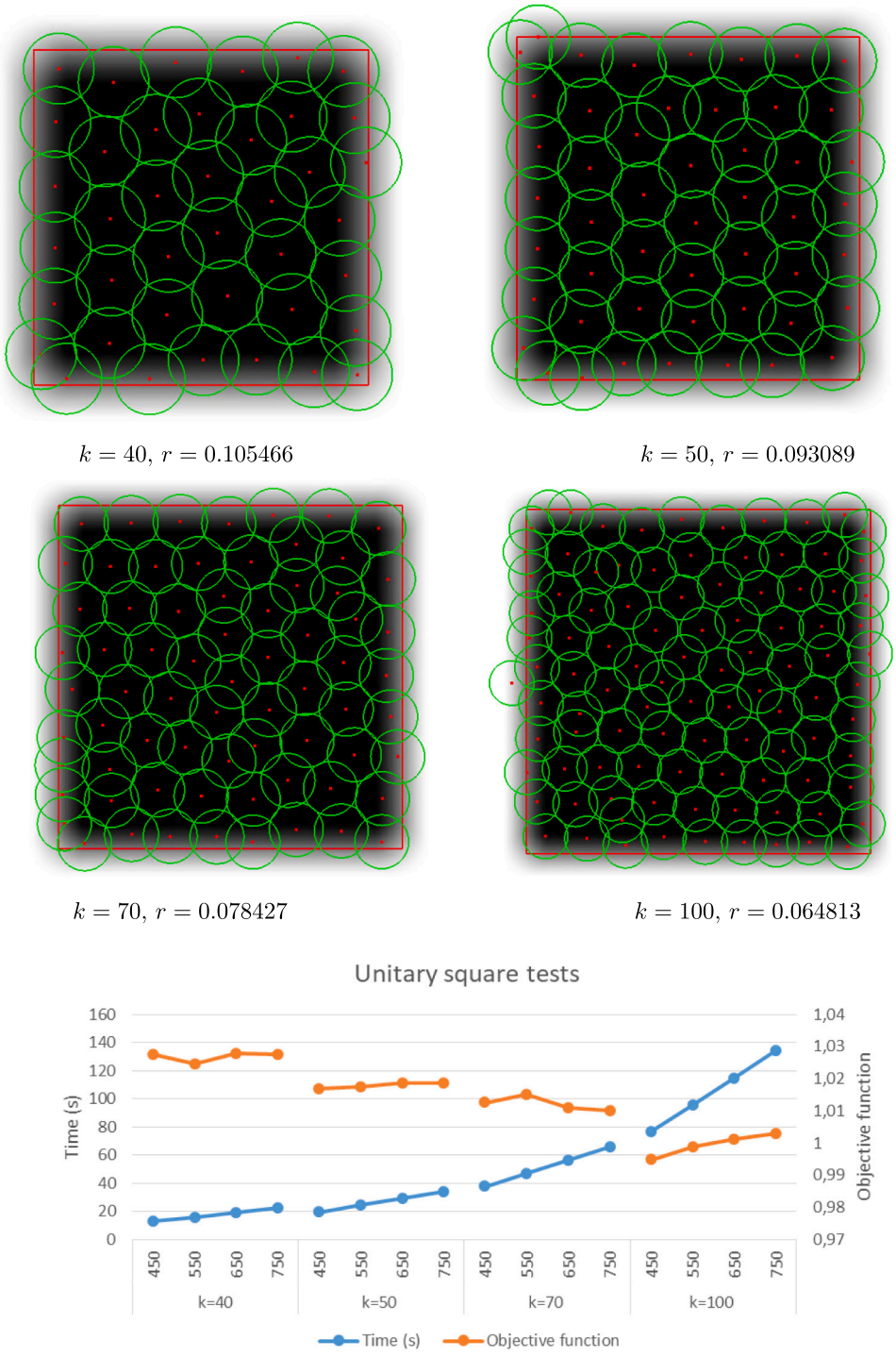

Fig. 11. Solutions obtained for the unit square, the running times and objective function value with S_{IV} from 450 to 750.

the objective function. The difference between the objective function and the area is 0.01 on average and the standard deviation 0.018. It corresponds to a 1.2% of the objective function value, on average, and the standard deviation is 2.6%.

Finally, to test the quality of our algorithm, we run the algorithm on a square unit domain with 30, 40, 70 and 100 circles. The radius considered in each case is the minimal radius needed so that the circles cover the unit square. The minimal radii are in Stoyan and Patsuk (2010) and the obtained results in Fig. 11. In this case, we disabled the termination criterion by asking to stop when the objective function achieves the 120% of the area of the k disks. It does never happen and, hence, all the steps are done. We ran the algorithm with $S_{III} = 45$ and S_{IV} varies from 450 to 750. From Fig. 11 we can see that the obtained solutions are very similar to the optimal ones provided in Stoyan and Patsuk (2010), the running times are good, and the value of the objective function varies between 0.99 to 1.03, on average is

1.01. The average of the obtained area is 0.98 and varies from 0.96 to 0.99.

6. Conclusions and further comments

We have designed an approximate fast and scalable algorithm to solve the problem of locating k disks of radius r so that their union covers as much area of a non connected polygonal domain $\mathcal{P}$ as possible. The algorithm parallelizes the SA technique by using synchronous Markov chains. The used objective function propitiates the overlap between the disks and the polygon and penalizes having areas covered by several disks. Moreover, the SA perturbation process also prevents having areas covered by several disks.

The objective function proposed approximates the area of the polygonal domain covered by a collection of k disks in $O(k)$ time. Indeed, it provides an approximation of the overlapped area of $\mathcal{P}$ with a disk by using an overlap area map and probabilistically estimating the area of $\mathcal{P}$ overlapped by any pair of disks. As demonstrated in the experimental results section, the objective function and the algorithm work well with a wide range of radii and numbers of circles. The algorithm is fast and provides good solutions.

As we mentioned in the introduction and show in the experimental results section, the strategy works with unconnected polygonal domains defined by several components. The dispersive Stage III scatters the initial disks through $\mathcal{P}$, already scaled and translated to $[-1,1]\times[-1,1]$, separating the perturbed center from the original one at most half of $\mathcal{L}$, the longest edge of the bounding box containing P, and locating the perturbed disk so that it intersects $\mathcal{P}$. Hence if the disconnected components are close enough so that a perturbed disk can reach them, there is no problem, and Stage III may distribute the sites correctly through $\mathcal{P}$. On the contrary, if there is one component farther from $0.1\mathcal{L}$ and $2r$ from the rest of $\mathcal{P}$, the Stage III dispersive SA will not be able to throw disks there. Hence, the disks will never cover this part of the domain. To avoid this, we can increase the $0.1\mathcal{L}$ maximal distance by using the interface or compact $\mathcal{P}$ approaching the isolated components to the rest of the domain in a preprocessing stage. The approached components must be kept at a distance bigger than $2r$ from the rest to avoid obtaining false overlapped areas. Hence, the obtained solutions are good approximations of the optimal solution and solve our challenge.

As future work, we plan to adapt the algorithm to locate k disks of different radii (prefixing the radii and number of disks of each radius) and use dynamic parallelism to evaluate the objective function.

CRediT authorship contribution statement

Narcís Coll: Conceptualization, Formal analysis, Writing – original draft, Writing – review & editing. **Marta Fort:** Conceptualization, Formal analysis, Writing – original draft, Writing – review & editing. **Moisès Saus:** Software.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Allahyari, M. Z., & Azab, A. (2018). Mathematical modeling and multi-start search simulated annealing for unequal-area facility layout problem. *Expert Systems with Applications*, *91*, 46–62.

Bansal, M., & Kianfar, K. (2017). Planar maximum coverage location problem with partial coverage and rectangular demand and service zones. *INFORMS Journal on Computing*, *29*(1), 152–169.

Cheng, S. W., & Lam, C. K. (2013). Shape matching under rigid motion. *Computational Geometry: Theory and Applications*, *46*(6), 591–603.

Cheong, O., Efrat, A., & Har-Peled, S. (2007). Finding a guard that sees most and a shop that sells most. *Discrete & Computational Geometry*, *37*(4), 545—563.

Chu, K.-W., Deng, Y., & Reinitz, J. (1999). Parallel simulated annealing by mixing of states. *Journal of Computational Physics*, *148*(2), 646–662.

Church, R. L. (1984). The planar maximal covering location problem. *Journal of Regional Science*, *24*(2), 185–201.

Church, R. L., & ReVelle, C. (1974). The maximal covering location problem. *Papers of the Regional Science Association*, *32*, 101–118.

Coll, N., Fort, M., & Sellarès, J. A. (2019). On the overlap area of a disk and a piecewise circular domain. *Computational and Operational Research*, *104*, 59–73.

Czech, Z. J., Mikanik, W., & Skinderowicz, R. (2010). Implementing a parallel simulated annealing algorithm. In R. Wyrzykowski, J. Dongarra, K. Karczewski, & J. Wasniewski (Eds.), *Parallel processing and applied mathematics* (pp. 146–155). Berlin, Heidelberg: Springer Berlin Heidelberg.

Davari, S., Fazel Zarandi, M. H., & Hemmati, A. (2011). Maximal covering location problem (MCLP) with fuzzy travel times. *Expert Systems with Applications*, *38*(12), 14535–14541.

Devroye, L. (1986). *Non-uniform random variate generation*. New York: Springer-Verlag.

Fabris, F., & Krohling, R. A. (2012). A co-evolutionary differential evolution algorithm for solving min–max optimization problems implemented on GPU using C-CUDA. *Expert Systems with Applications*, *39*(12), 10324–10333.

Ferreiro, A., García, J., López-Salas, J., & Vázquez, C. (2013). An efficient implementation of parallel simulated annealing algorithm in GPUs. *Journal of Global Optimization*, *57*(3), 863–890.

Hajek, B. (1988). Cooling schedules for optimal annealing. *Mathematics of Operations Research*, *13*(2), 311–329.

Hochbaum, D. S. (Ed.), (1996). Approximation algorithms for NP-hard problems. In *chapter Approximating Covering and Packing Problems: Set Cover, Vertex Cover, Independent Set, and Related Problems* (pp. 94–143). 20 Park Plaza Boston, MAUnited States: PWS Publishing Co..

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680.

Kravitz, S., & Rutenbar, R. (1987). Placement by simulated annealing on a multiprocessor. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *6*(4), 534–549.

Lee, S. Y., & Lee, K.-G. (1996). Synchronous and asynchronous parallel simulated annealing with multiple Markov chains. *IEEE Transactions on Parallel and Distributed Systems*, *7*(10), 993–1008.

Matisziw, T. C., & Murray, A. T. (2009). Siting a facility in continuous space to maximize coverage of a region. *Socio-Economic Planning Sciences*, *43*(2), 131—139.

Megiddo, N., Zemel, E., & Hakimi, S. (1983). The maximum coverage location problem. *SIAM Journal of Algebraic and Discrete Methods*, *4*(2), 253–261.

Mount, D. M., Silverman, R., & Wu, A. Y. (1996). On the area of overlap of translated polygons. *Computer Vision and Image Understanding*, *64*(1), 53–61.

Murray, A., Matisziw, T., Wei, H., & Tong, D. (2008). A geocomputational heuristic for coverage maximization in service facility siting. *Transactions in GIS*, *12*(6), 757–773.

Murray, A. T., O'Kelly, M. E., & Church, R. L. (2008). Regional service coverage modeling. *Computers and Operations Research*, *35*(2), 339–355.

Murray, A. T., & Tong, D. (2007). Coverage optimization in continuous space facility siting. *International Journal of Geographical Information Science*, *21*(7), 757–776.

Nasab, H. H., & Mobasheri, F. (2013). A simulated annealing heuristic for the facility location problem. *International Journal of Mathematical Modelling and Numerical Optimisation*, *4*(3), 210–224.

Onbaoğlu, E., & Ozdamar, L. (2001). Parallel simulated annealing algorithms in global optimization. *Journal of Global Optimization*, *19*(1), 27–50.

Ram, D. K., Sreenivas, T. H., & Subramaniam, K. G. (1996). Parallel simulated annealing algorithms. *Journal of Parallel and Distributed Computing*, *37*(2), 207–212.

Sonuç, E., Sen, B., & Bayir, S. (2017). A parallel simulated annealing algorithm for weapon-target assignment problem. *International Journal of Advanced Computer Science and Applications*, *8*(4), 87–92.

Sonuç, E., Sen, B., & Bayir, S. (2018). A cooperative GPU-based parallel multistart simulated annealing algorithm for quadratic assignment problem. *Engineering Science and Technology, An International Journal*, *21*(5), 843–849.

Stoyan, Y. G., & Patsuk, V. M. (2010). Covering a compact polygonal set by identical circles. *Computational Optimization and Applications*, *46*, 75–92.

Tavares, R. S., Martins, T. C., & Tsuzuki, M. S. G. (2011). Simulated annealing with adaptive neighborhood: A case study in off-line robot path planning. *Expert Systems with Applications*, *38*(4), 2951–2965.

Tong, D., & Murray, A. T. (2009). Maximising coverage of spatial demand for service. *Papers in Regional Science*, *88*(1), 85–97.

Wei, R., & Murray, A. T. (2014). Continuous space maximal coverage: Insights, advances and challenges. *Computers & Operations Research*, *62*, 325–336.

Wei, K.-C., Wu, C.-C., & Yu, H.-L. (2015). Mapping the simulated annealing algorithm onto CUDA GPUs. In *2015 10th International conference on intelligent systems and knowledge engineering (ISKE)* (pp. 358–365).

Zhang, L., Ye, Z., Xiao, K., & Jin, B. (2019). A parallel simulated annealing enhancement of the optimal-matching heuristic for ridesharing. In *2019 IEEE international conference on data mining (ICDM)* (pp. 906–915).