Soft Comput (2016) 20:1521–1535
DOI 10.1007/s00500-015-1603-y

METHODOLOGIES AND APPLICATION

Evolutionary computation solutions to the circle packing problem

Juan J. Flores · Jose Martínez · Felix Calderón

Published online: 3 February 2015

Abstract In this work, we present an evolutionary omputation-based solution to the circle packing problem (ECPP). The circle packing problem consists of placing a set of circles into a larger containing circle without overlaps: a problem known to be NP-hard. Given the impossibility to solve this problem efficiently, traditional and heuristic methods have been proposed to solve it. A naïve representation for chromosomes in a population-based heuristic search leads to high probabilities of violation of the problem constraints, i.e., overlapping. To convert solutions that violate constraints into ones that do not (i.e., feasible solutions), in this paper we propose two repair mechanisms. The first one considers every circle as an elastic ring and overlaps create repulsion forces that lead the circles to positions where the overlaps are resolved. The second one forms a Delaunay triangulation with the circle centers and repairs the circles in each triangle at a time, making sure repaired triangles are not modified later on. Based on the proposed repair heuristics, we present the results of the solution to the CPP problem to a set of unit circle problems (whose exact optimal solutions are known). These benchmark problems are solved using genetic algorithms, evolutionary strategies, particle swarm optimization, and differential evolution. The performance of the solutions is compared to those known solutions based on the packing density. We then perform a series of experiments to determine the performance of ECPP with non-unitary circles. First, we compare ECPP's results to those of a public competition, which stand as the world record for that particular instance of the non-unitary CPP. On a second set of experiments, we control the variance of the size of the circles. In all experiments, ECPP yields satisfactory near-optimal solutions.

Keywords Optimization · Circle packing · Evolutionary computation · Genetic algorithms · Evolutionary strategies · Particle swarm optimization · Differential evolution

Communicated by V. Loia.

J. J. Flores (✉) · J. Martínez · F. Calderón
Universidad Michoacana, Morelia, Mexico
e-mail: juanf@umich.mx

J. Martinez
e-mail: jmart80@hotmail.com

F. Calderón
e-mail: calderon@umich.mx

1 Introduction

Given N circles, of given radii, the circle packing problem (CPP) is concerned with how to pack those circles into a circular container, without overlapping. CPP has a wide spectrum of applications; it is encountered in a variety of real-world applications, including production and packing for the textile, apparel, naval, automobile, aerospace, and food industries (Castillo et al. 2008).

Many optional features exist on this problem; e.g., the container can be a circle, rectangle, or polygon, and the objects can be a circular, rectangular, or irregular. This paper addresses CPP, where the objects and container are circles. This problem has been proven to be NP-hard (Demaine et al. 2010). So, heuristic search methods are generally proposed to solve this problem.

Packing circular objects is a challenge in discrete and computational geometry (Szabó et al. 2006). With a large number of circular objects to pack, the optimal solution is very difficult to find. An optimal solution may be rotated, reflected, or the circular objects reordered; hence, the number of equivalent optimal solutions blows up as the number of circular objects increases (Hifi and M'Hallah 2009). In addition, one or more of the circular objects may be moved slightly without affecting the optimal solution. In fact, there exists a continuum of optimal solutions (Hifi and M'Hallah 2009).

Circle packing problem has many important applications in manufacturing, logistics, networks, facility layout, and materials science (Castillo et al. 2008). For example, in the automobile industry, design engineers have to estimate the size of the hole to be drilled on the body of the car and through which they plan to pass a bundle of wires that connect car's sensors to the display board (Sugihara et al. 2004). The hole has to be large enough to allow all wires to pass, but as small as possible to avoid unnecessarily weakening the body (Sugihara et al. 2004). CPP is also encountered in the manufacturing of sprockets for the motorcycle industry (Dowsland et al. 2007). Similarly, it is of interest to the telecommunication, electrical, oil companies, and refineries, which have to pass bundles of different types of cables, pipes, and insulated pipes through cylindrical shapes over very long distances. The smaller the diameters of the cylinders, the cheaper is the cost. Finally, CPP emerges in material science where it is used to interpret topological relationships encountered when analyzing the normal grain growth in two dimensions (Nordbakke et al. 2004) and to model certain absorption patterns of molecules (Harary et al. 1996).

Last, there is the issue of computational accuracy. The goal is to search for the best packing of the N circles inside the container c_0, where the best packing minimizes unused space.

In this article, we are presenting an approach to get a feasible solution through evolutionary computation. This approach is called EC-CPP (for evolutionary computation CPP). Perhaps the first chromosome representation that comes to mind is a vector containing the coordinates of the center of each circle. Most evolutionary computation methods start with a random population, and then successively apply perturbations (i.e., genetic operators) to members of the population, until the best possible solution is reached. When generating new solutions, both randomly or by perturbations, the probability of generating overlaps is high.

Two methods have been extensively explored and used to enable metaheuristic optimization to produce solutions to constrained problems. One is to apply a penalty function to individuals that violate constraints. A penalty function is normally expressed as a term or coefficient in the fitness function that makes violating individuals the least fit, so they are discarded in the evolutionary process. This approach has two main drawbacks. The first one is that the design of the penalty function is domain dependent and non-trivial, and the second one is the time spent in processing a very large number of violating individuals which end up being discarded. The second approach to deal with constrained optimization, known as repairing, uses a function that takes an individual that violates the constraints and returns a different individual that does not violate the constraints (the closer to the original, the better). This is the approach we use in this paper.

EC-CPP implements two repair heuristics and use them in conjunction with several metaheuristic search methods. We evaluate the results that EC-CPP produces in terms of the obtained density; i.e., the relationship between the area of the circumcircle and the sum of the areas of the circles inside it. We compare those results with the existing benchmarks (Specht 1999). Experimental results show that our approach has a good performance in terms of the solutions' densities. CPP can be divided into two variations of the same problem. The first one known as the unit circle packing problem (UCPP) considers all circles of the same size. The second and more general one considers all circles of arbitrary sizes. We have empirically tested EC-CPP with many instances of both variations of the CPP. Nevertheless, our evaluations focus on the UCPP, given that there are theoretical solutions for the problem up to 1,104 circles.

The rest of the paper is organized as follows. In Sect. 2, we present the problem definition and formulation of the circle packing problem. In Sect. 3, we give a literature survey of work related to the circle packing problem. In Sect. 4, we describe the proposed algorithms. Computational results are presented in Sect. 5. Finally, in Sect. 7, we present our conclusions.

2 Problem definition

2.1 The circle packing problem

Consider the set C of N circles; each circle $c_i \in C$, $i \in [1, 2, \ldots, N]$ and has the structure (P_i, r_i) where $P_i = (x_i, y_i) \in \mathbb{R}^2$ and $r_i \in \mathbb{R}$.

CPP consists of providing the locations of the circles' centers $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$, that minimize the radius r_0 of the containing circle c_0, subject to:

$$(x_0 - x_i)^2 + (y_0 - y_i)^2 \leq (r_0 - r_i)^2, \quad i \in \{1, 2, \ldots, N\} \tag{1}$$

$$(x_i - x_j)^2 + (y_i - y_j)^2 \geq (r_i + r_j)^2, \quad i \neq j, \forall (i, j) \in \{1, 2, \ldots, N\}^2 \tag{2}$$

where constraint (1) establishes that all circles must be inside the container circle c_0 and constraint (2) establishes that circles must not overlap.

It is important to note that the smaller the radius of the container circle, the less unused space, therefore, the greater is the packing density. Although the problem has been stated as a minimization one, where the objective function is directly the radius of the enclosing circle, it could be formulated as a maximization problem, where the objective function is the packing density. Both optimization problems are equivalent.

2.2 Levels of abstraction

When solving CPP using a population-based metaheuristic, individuals are produced either at random or by perturbations generated by (random) combination of information encoded in other individuals. Under those circumstances, the probability of generating individuals that do not comply with the constraints (i.e., at least a pair of circles overlap) is very high. We distinguish two types of individuals: those that do not violate the constraints form the feasible search space, or feasible space. The set of all possible individuals, regardless of whether or not they violate the constraints, forms the general search space, or search space.

To solve CPP using a population-based metaheuristic, we need to explore the search space and determine an individual that does not violate the constraints and produces the smallest possible radius of the enclosing circle (the fitness or objective function).

There are two general approaches to deal with individuals that represent unfeasible solutions: penalty functions and repair functions.

A penalty method discourages unfeasible solutions by giving penalties so that feasible solutions are preferred to unfeasible solutions. Michalewicz (1996) demonstrated a good summary on constraint handling methods for evolutionary algorithms, and most of the existing methods are based on penalty functions. Each method is different in the amount of penalty assigned to unfeasible individuals. These variations become problem dependent for better performance.

Repair functions modify unfeasible solutions to produce feasible ones. This mapping preferably has to find the closest feasible solution to the individual to be repaired: a repaired solution substitutes the unfeasible solution and can be used for the search process. There are no general guidelines on how to repair unfeasible solutions. Most of the repair heuristics are problem dependent.

At the feasible space, metaheuristics from the field of evolutionary computation such as genetic algorithms (GA), evolutionary strategies (ES), differential evolution (DE), and particle swarm optimization (PSO) can be applied to optimize the objective function, i.e., find the smallest possible container circle. These metaheuristics do not always guarantee an optimal solution. However, in most cases they give a near optimal solution with less effort and time than the mathematical methods.

In this paper, we propose the use of two different repair functions. Repair functions allow us to view the search process at two levels of abstraction. At the lower level, we have the search space; at the higher level, we have the feasible space (see Fig. 1). Some individuals in the search space can be mapped to the feasible space, not all of them. The initial population will be composed, in its great majority, of individuals that do not comply to the non-overlapping constraints.

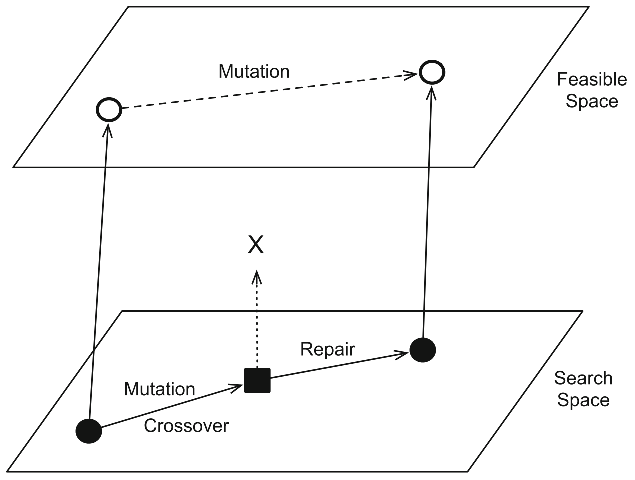

Fig. 1 Feasible space

Those individuals are repaired, so that the process starts with a population of individuals that can be mapped to the feasible space. The application of perturbations (a.k.a. genetic operators) to feasible individuals will most likely produce unfeasible individuals. Offsprings are also repaired.

2.3 CPP definition revisited

Given the above, we can view the evolutionary process as occurring at the feasible space as an unconstrained optimization problem. Now, from the feasible space level of abstraction, CPP can be formally stated as follows:

Given a set of circles C, determine these circles' positions (x, y), such that minimize the objective function

$$r_0^* = \min_{r_0} f(C) \tag{3}$$

where r_0, the radius of the container circle c_0, is computed using Algorithm 4 of Sect. 4.4.

3 Related work

Circle packing problem is concerned with the arrangement of a finite number of circles inside a circular container without overlap. The mathematical model that represents CPP consists mainly of two types of constraints: the container boundary and the non-overlapping constraints. The container boundary constraint ensures that all circles lie inside the container and the non-overlapping constraint ensures that for any two given circles, the distance between their centers is at least the sum of their radii. When both sets of constraints are satisfied, we have a feasible configuration for the packing problem. This section distinguishes between two main approaches to solve this problem: traditional optimization and evolutionary computation.

3.1 Traditional optimization

Mladenovic et al. (2005) apply a general reformulation descent heuristic (RD) to the problem of identifying the largest radius of identical circles that can be packed into a unit containing circle. RD iterates switching from solving CPP expressed in Cartesian coordinates to solving it expressed in polar coordinates and viceversa until no further improvement is obtained.

Wenqi and Yan (2004) formulate CPP as a potential energy function by simulating a system of elastic solids. They position all circles randomly inside the containing circle. If this configuration has no overlapping circles, a feasible solution is at hand. Otherwise, the elastic repulsion forces generated by the overlaps drive the overlapping circles to restore their shape and size. The circles move along straight lines, colliding with each other and with the containing circle until the composition of elastic forces is decreased to zero. If the amount of overlap is also decreased to zero, then the process stops with a feasible solution. Otherwise, the process restarts.

Zhang and Deng (2005) adopt the model of Wang et al. (2002), and use a hybrid approach consisting of simulated annealing to explore the neighborhood of the current solution, and tabu search to implement the jumps. When exploring the neighborhood of the current solution, one of the circles whose position is infeasible is translated and the degree of infeasibility of the neighbor is computed. A neighboring solution that reduces the degree of infeasibility becomes the incumbent solution whereas a non-improving solution is accepted with a given probability, which decreases as the search becomes more selective.

Pintér and Kampas (2006) present numerical results obtained using Lipschitz global optimizer (LGO). Castillo et al. (2008) applies various off-the-shelf generic global optimization techniques, and compare their performance. They further improve the results of the generic solvers by implementing a posteriori strategy that, given a near-optimal initial arrangement, swaps all pairs of adjacent-sized circles until no possible improvement exists.

Addis et al. (2008) present a strategy for optimally placing circles in a smallest circle. They mix standard local optimization routines with local moves between minima, while reinforcing solution dissimilarity but reducing the solution space. The resulting approach obtains the best-known solution for problems of up to 50 circles and $r_i = i, i = 1, \ldots, N$.

Al-Modahka et al. (2011) present an adaptive hybrid algorithm that addresses the combinatorial structure of CPP via a Tabu search (TS), and its continuous optimization aspects via a combination of nested partitioning (NP) and nonlinear optimization. The hybrid TS/NP algorithm exploits the advantages of TS to undertake a local search aimed at identifying a good permutation of the circles, whereas NP undertakes a global search to identify their respective best positions. The provided results are further modified/improved using some diversification strategies.

Francesco et al. (2014) propose an algorithm that, by applying a strength along a selected direction on each circle, simulates the shifting of circles on the plane and tries to reduce the radius of the circular container during this movement. The algorithm is based on a multistart technique where the starting solutions are produced by a tabu search heuristic that uses also the current best solution.

3.2 Evolutionary computation

Evolutionary computation for circle packing problem (ECCPP) relies on the parameter values that encode solutions, which are initially randomly distributed between lower and upper bounds. Non-overlapping constraints are easily broken; it is not easy to avoid producing unfeasible solutions during the initialization process. The modifications of valid solutions through crossover and mutation in genetic algorithms, for example, may produce unfeasible solutions as well as feasible.

Zhi-Qin et al. (2001) propose a human–computer interactive genetic algorithm for solving the two-dimensional constrained layout optimization problem. The algorithm composes chromosomes with artificial individuals (AIs) and divides the population into subgroups. Each subgroup has different values of crossover and mutation probabilities. After copy, crossover, and mutation, the best individual in each subgroup is transferred to adjacent subgroups. New AIs are determined based on the value of the fitness function. These new AIs are copied to ensure they play an important role in chromosome population. Then, they are placed into the chromosome population to replace the worse individuals. The steps mentioned above are repeated until the human expert finds a satisfactory solution.

Xu et al. (2007) present a novel order-based positioning technique for the layout optimization problem. A permutation $(1, 2, \ldots, N)$ can yield a layout by specifying the order in which circles are placed. As there exist $N!$ possible permutations for N circles, the GA is an appropriate technique to use to search such a large space. The GA is used to evolve the placement order of each circle.

Shi et al. (2010) propose an improved evolution strategy with crossover operator (ESCO) to tackle the constrained circle packing problem. The proposed ESCO extends a canonical ES to deal with combinatorial optimization by employing the crossover operator from genetic algorithms, aiming to exchange the location of circles for obtaining a better packing scheme. They aim to solve the general CPP; they measure the quality of the packing by the size of the container and the weighted average pair-wise distance between circles.

Yan-Jun et al. (2012) present a layout pattern-based particle swarm optimization algorithm (LPPSO) for solving the two-dimensional packing problem with constraints. In the optimizing process of LPPSO, some individuals are constructed according to non-isomorphic layout patterns and these individuals are added into the current population of the PSO algorithm to replace the worst individuals; the new population is created as a result. A non-isomorphic layout pattern is constructed based on an exact boundary-line approach (the distance between two circles) to avoid premature convergence and improve the computational efficiency.

To the best of the knowledge of the authors, there does not exist a satisfactory E.C. approach to solving CPP. There are not even previous publications available to compare our results with other approaches.

4 Population-based solution

A population is a set of individuals where each individual represents a prospect solution to the problem. A population-based solution spreads through a good search region, instead of only a good point, in the search space. In addition to representing a potentially good search region, the variance in the population members provides information about the extent of the potential search region. If new solutions are created in proportion to the variance of an existing population of points, a self-adaptive search procedure can be developed. In the start of such an algorithm with a randomly picked initial population of solutions, the variance in the population members is expected to be large, thereby ensuring a thorough exploration of the entire search space. On the other hand, during later iterations when the population of points have covered near the optimum, the variance in population members is expected to be small, thereby ensuring a focused search near the optimum. Without an external guidance, such a population-based algorithm can widen or narrow down its search power adaptively (Deb 2004).

These algorithms are typically applied to hard problems with a large search space, where the presence of multiple solutions is exploited to find better search regions. The presence of multiple solutions in a search process allows diversity to be maintained and this can be beneficial in handling constrained optimization problems. A population-based solution usually yields the best values of the variables or the best scenarios which are an approximation to the optimum solution. A solution is considered optimum with respect to the best performance or best fitness in terms of the objective function.

4.1 A general evolutionary computation algorithm

Evolutionary computation or population-based algorithms are direct search methods, which use only objective function values to drive the search and employ more than one solution at each iteration. These algorithms work on an encoding of the parameters set, search from a population of individuals, use an objective function, use probabilistic transition rules, and can provide a number of potential solutions to a given problem. Population-based algorithms allow direct generation of possible solutions in a single run.

Evolutionary computation is an ambitious name for a simple idea: use the theory of evolution as an algorithm. Any program that uses the fundamental structure shown in Algorithm 1 could be termed Evolutionary Computation. In an evolutionary algorithm, the first step is to create a population of individuals. The structures that describe those individuals are filled in at random. An objective function (fitness function for EAs) is used to decide which solutions deserve further attention. In the main loop of the algorithm, we pick solutions so that on average better solutions are chosen. This process is known as selection. The selected solutions are then subjected to variation. This variation can be in the form of random tweaks to a single structure or exchange of genetic material between structures. Changing a single structure is called unary variation or mutation. Exchanging material between structures is called an n-ary variation or crossover.

The main loop iterates the process of population updating via selection and variation. In accordance with the general theory of evolution, this should move the population toward fitter structures. This continues until you reach an optimum in the space of solutions, defined by your fitness function, or until a specified number of generations has been reached. This optimum may be the best possible place in the entire fitness space, or it may merely be better than all structures nearby in the search space. Adopting the language of optimization, we call these two possibilities global and local optima. Unlike many other types of optimizers, an evolutionary algorithm can jump from one optimum to another.

Algorithm 1 shows the basic structure of an evolutionary algorithm; in our case, we add a step that tests the feasibility of each individual. To maintain our perspective of a high level of abstraction (i.e., working at the feasible space), after an individual is generated (steps 1 and 5) we apply a repairing process. This process repeats until computes the fitness of each individual (i.e., determines the radius of the enclosing circle) reaching an specified number of generations.

Algorithm 1 GenericEA

```
1: Generate a random population of individuals
2: repeat
3:     Test the individuals for quality
4:     Select individuals to be perturbed
5:     Produce new variations of selected individuals
6:     Replace old individuals with new ones
7: until termination criterion is satisfied
```

4.2 Repulsion-based repair

A configuration $P = ((x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N))$ represents fixed positions for the N circles. The repairing process can be stated as finding coordinates (x_i, y_i) of every circle c_i so that the whole configuration does not violate the non-overlapping constraint.

In this section, we propose an algorithm inspired on a physical analogy. We imagine the set of circles as elastic rings. Under this metaphor, circles that are too close together (i.e., overlapping circles) start deforming and exerting a restoring force that pushes them apart. We allow these forces to push circles to the point where they touch, which is the point where the overlap disappears. Where first is necessary, calculate the center of gravity P_g of the given circles to sort them by proximity to this center.

Given two circles c_i and c_j where $i \neq j$ in the given configuration. If $(x_i - x_j)^2 + (y_i - y_j)^2 < (r_i + r_j)^2$, then c_i and c_j overlap, then the overlapping depth d_{ij} between them is

$$d_{ij} = r_i + r_j - \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}, \quad (i, j) \in \{1, \ldots, N\}^2 \tag{4}$$

$$d_{ij} \begin{cases} <0 & \text{distant circles} \\ =0 & \text{tangent circles} \\ >0 & \text{overlapping circles} \end{cases} \tag{5}$$

Since the initial configuration is generated randomly, there most likely exist overlaps among the circles. According to the physical analogy, these circles are considered as elastic rings. If there exists overlap between objects, they must have elastic forces acting on them, so they will move with the action of those forces.

Figure 2 shows two circles c_i and c_j that overlap; c_j will move along the direction from i to j by the reaction of elastic forces. How far it moves depends on its embedding depth d_{ij} (Eq. 4), c_j moves away from c_i a distance d_{ij} until it does not overlap with c_i. According to the following equations, the coordinates of c_j are modified.

$$dx_j = \frac{x_j - x_i}{D_{ij}} d_{ij} \tag{6}$$

$$dy_j = \frac{y_j - y_i}{D_{ij}} d_{ij} \tag{7}$$

where D_{ij} denotes the distance from the centers of c_i and c_j, d_{ij} is the same as the previous definition, dx_j is the projection of d_{ij} in the horizontal axis x and dy_j is the projection of d_{ij} in the vertical axis y. Therefore, new position is

$$x'_j = x_j + dx_j \tag{8}$$

$$y'_j = y_j + dy_j \tag{9}$$

Equations (4) to (9) solve the overlapping problem for two circles. If we have more than two circles, we need to define the order in which their positions are corrected, so that it guarantees that, at the end, no overlaps exist and that corrected circles are not moved in subsequent corrections (otherwise the algorithm could fall into an infinite cycle of corrections).

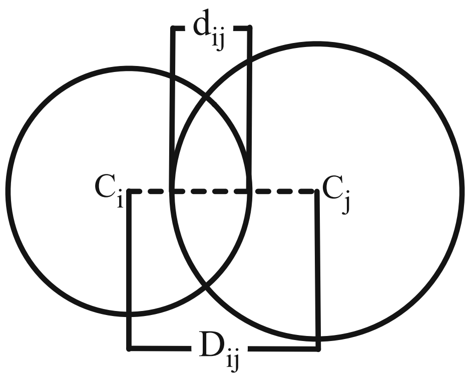

Fig. 2 Two overlapping circles

Algorithm 2 proposes such an order, guaranteeing both properties mentioned in the previous paragraph. Let us define P_g as the center of gravity of the set of circles. $P_g = \{\overline{x}, \overline{y}\}$, where $\overline{x}$ and $\overline{y}$ are the means of the coordinates of the centers of all circles. The main idea is to compute the center of gravity of the set of circles, and correct all possible overlaps from P_g, outwards.

If the previous configuration is $P = ((x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N))$, then the new configuration is $P' = ((x_1, y_1), (x'_2, y'_2), \ldots, (x'_N, y'_N))$ represents the center coordinates of circles without overlapping. Note that c_1 is the closest circle to P_g, therefore, it does not change with the execution of Algorithm 2.

It is easy to prove that the computational complexity of Algorithm 2 is $O(N^2)$. Since Algorithm 2 does not have the means to find the closest neighbors of each circle, it compares every circle with every other remaining one. Therefore, it is important to seek the implementation of repairs that are faster and consume less computational resources. An alternative approach is proposed in the next section.

Algorithm 2 RBRepair (C)

```
1: $C$: set of $Circles$
2: $P_g = (x_g, y_g) = Center\ of\ Gravity\ of\ Circles$
3: Sort $C$ by proximity to $P_g$
4: for $i = 1$ to $|C| - 1$ do
5:     for $j = i + 1$ to $|C|$ do
6:         if $c_j$ overlaps $c_i$ then
7:             $c_j \leftarrow Repair\ c_j$ to avoid overlapping, according to Eqs. 6 – 9
8:         end if
9:     end for
10: end for
11: return $C$ (without overlapping)
```

4.3 Delaunay triangulation-based repair

A configuration P represents fixed positions of the centers of the N circles. The repairing process can be stated as finding coordinates P_i' of every circle c_i so that constraint (2) is not violated.

The set $T = \{T_1, \ldots, T_{N_t}\}$ represents the Delaunay triangulation formed by P. Each triangle $T_k = (P_{k_1}, P_{k_2}, P_{k_3})$ of T is a triplet of circle identifiers.

The repairing process described in this subsection consists on removing all overlaps using the Delaunay triangulation. The Delaunay triangulation is a triangulation such that the circumcircle of each triangle does not contain another point in its interior.

Consider a set P of points in the Euclidean plane, where $P_i = (x_i, y_i)$ and $|P| \geq 3$. Assume that these points are not all collinear, and that no four points are cocircular. Let $d(P_i, P_j)$ denote the Euclidean distance between points P_i and P_j. In Lee and Schachter (1980), a T triangulation of a set of N points P is considered Delaunay triangulation of P if either of the following lemmas hold:

Lemma 1 *Given a set P of N points, any triangulation $T(P)$ has the same number of triangles,*

$$N_t = 2(N-1) - N_h \tag{10}$$

and the same number of edges,

$$N_e = 3(N-1) - N_h \tag{11}$$

where N_h is the number of points on the convex hull of P.

Lemma 2 *Given a set P of points, any edge (P_i, P_j) is a Delaunay edge of $DT(P)$ if and only if there exists a point x such that the circle centered at x and passing through P_i and P_j does not contain in its interior any other point of P.*

Lemma 3 *Given a set P of points, a triangle $T_k = (P_{k_1}, P_{k_2}, P_{k_3}) \in DT(P)$ if and only if its circumcircle does not contain any other point of P in its interior.*

Among the possible triangulations of a set of points, Delaunay Triangulations are an interesting alternative as they tend to generate triangles that maximize the minimum angle of all the triangles in the triangulation. Additionally, the performance of Delaunay triangulations is acceptable in practice; the algorithm has a complexity time of $O(N \log N)$. The repulsion-based repair algorithm does not use information about neighborhood of circle. The DT-based repulsion algorithm uses the neighborhood information provided by the DT to repair triplets of triangles at once, not having to verify overlaps with any other circle.

For CPP the set P of Eq. 12 is formed by the coordinates (x, y) of the centers of the circles.

$$P = \{P_1, \ldots, P_N\}, \quad P_i = (x_i, y_i),\ i \in [1, 2, \ldots, N] \tag{12}$$

Constraint 13 represents the conditions to apply by the repair process based on the Delaunay triangulation, i.e., the three circles after the repair process must be as close as possible to each other, and at last one of the three must be tangent with the other two.

$$\begin{aligned}&\sqrt{(x_{k_1} - x_{k_3})^2 + (y_{k_1} - y_{k_3})^2} = r_{k_1} + r_{k_3},\\ &\sqrt{(x_{k_2} - x_{k_3})^2 + (y_{k_2} - y_{k_3})^2} = r_{k_2} + r_{k_3},\\ &\quad k_1 \neq k_2 \neq k_3, k_1, k_2, k_3 \in [1, 2, \ldots, N]\end{aligned} \tag{13}$$

The repair process starts with $k = 1$ for $T_k = (P_{k_1}, P_{k_2}, P_{k_3})$, the first triangle in T, where P_{k_i}, corresponds to circle c_{k_i} of T_k, for $1 \leq i \leq 3$. Circle c_{k_i} has coordinates P_{k_i}. The Delaunay triangulation-based repair process first repairs P_{k_1} and P_{k_2} of T_k. We establish that P_{k_1} will not be moved and P_{k_2} is attracted to or repelled from P_{k_1} to make their circles tangent. P_{k_2}' is the repaired position of P_{k_2}. Circle c_{k_2} is moved along the line that connects P_{k_1} with P_{k_2} to a place where Eq. (14) is satisfied

$$\sqrt{(x_{k_1} - x'_{k_2})^2 + (y_{k_1} - y'_{k_2})^2} = r_{k_1} + r_{k_2} \tag{14}$$

Similarly proceed to repair circle c_{k_3}, until Eq. (15) is satisfied

$$\begin{aligned}&\sqrt{(x_{k_1} - x'_{k_3})^2 + (y_{k_1} - y'_{k_3})^2} = r_{k_1} + r_{k_3},\\ &\sqrt{(x'_{k_2} - x'_{k_3})^2 + (y'_{k_2} - y'_{k_3})^2} = r_{k_2} + r_{k_3}.\end{aligned} \tag{15}$$

I.e., given three circles c_{k_1}, c_{k_2}, and c_{k_3}, c_{k_2} is attracted to or repelled from c_{k_1}, x_{k_2}' and y_{k_2}' are calculated with Eqs. 6–9. The third circle c_{k_3} is attracted to or repelled from c_{k_1} and c_{k_2}', its new coordinates x_{k_3}' and y_{k_3}' are calculated solving the system of quadratic equations (16).

$$\begin{aligned}&x'_{k_3}(x'_{k_3} - 2x_{k_1}) + y'_{k_3}(y'_{k_3} - 2y_{k_1}) = (r_{k_1} + r_{k_3})^2\\ &\quad -x_{k_1}^2 - y_{k_1}^2,\\ &x'_{k_3}(x'_{k_3} - 2x'_{k_2}) + y'_{k_3}(y'_{k_3} - 2y'_{k_2}) = (r_{k_2} + r_{k_3})^2\\ &\quad -x'^2_{k_2} - y'^2_{k_2}.\end{aligned} \tag{16}$$

The remaining triangles will be repaired in an order such that every triangle being repaired is adjacent to a repaired triangle, according to the Delaunay triangulation. Under those circumstances, two of the circles of T_k (the triangle under repair) have already been fixed and will not be moved again. Assume $T_k = (P_{k_1}, P_{k_2}, P_{k_3})$, where P_{k_1} and P_{k_2} have been fixed; that leaves the only option of moving the remaining P_{k_3}, to repair triangle T_k. P_{k_3} will be moved to a place where it satisfies Eq. (17).

$$\sqrt{(x'_{k_1} - x'_{k_3})^2 + (y'_{k_1} - y'_{k_3})^2} = r_{k_1} + r_{k_3},$$
$$\sqrt{(x'_{k_2} - x'_{k_3})^2 + (y'_{k_2} - y'_{k_3})^2} = r_{k_2} + r_{k_3}.$$
$$k \in [2, 3, \dots, N_t] \tag{17}$$

The order in which circles are repaired is important. We need to produce an efficient algorithm that repairs every circle only once, and once it is fixed it will not be moved again. On the other hand, if the order is arbitrary, the repair process can get to a point where overlaps exist and no circles can be moved to resolve the situation. To avoid these problems and guarantee convergence of the repair process, we compute the center of gravity for the set of circles, then sort them by ascending distance to the center of gravity. Once we have the circles sorted, we compute the Delaunay triangulation, which yields the neighbors of each circle in triplets. We will repair these triplets attracting circles to the center of gravity. Algorithm 3, DTRepair, implements this process. DTRepair takes as input a set of circles and returns a set of repaired circles.

4.4 Enclosing

Once a configuration contains no overlaps, we need to determine the size of the smallest circle that contains all circles in the configuration. Enclosing finds the coordinates $\{x_0, y_0\}$ that minimize r_0, the radius of the containing circle c_0.

To compute the enclosing circle, Algorithm 4 sorts the circles by descending distance from their center of gravity. It takes the farthest circle as the starting enclosing circle, and takes each circle at a time, checking whether or not it is contained in the previous enclosing circle. If it is not, it computes the new enclosing circle.

Assume at a given iteration i, circle c_i has center coordinates (x_i, y_i), and container circle c_0 has coordinates (x_0, y_0). If $\sqrt{(x_0 - x_i)^2 + (y_0 - y_i)^2} + r_i < r_0$, then c_0 completely contains c_i, and there is nothing to do. Otherwise, we need to compute a new c_0 that encloses all of them. In that case, the radius of the new enclosing circle r'_0 is

$$r'_0 = \frac{\sqrt{(x_0 - x_i)^2 + (y_0 - y_i)^2} + r_0 + r_i}{2} \tag{18}$$

Algorithm 3 DTRepair (C)

```
1: $C$ : set of $Circles$
2: $P_g = (x_g, y_g) =$ Center of Gravity of $C$
3: Sort $C$ by ascending distance to $P_g$
4: $T \leftarrow$ Delaunay Triangulation of the centers in $Sorted\ C$
5: $Coordinates$ of $C(T_1) \leftarrow Repair\ C(T_1)$, according to Eqs. 14, 15

6: for $k = 2$ to $|T|$ do
7:     $Coordinates$ of $C(T_k) \leftarrow Repair\ C(T_k)$, according to Eq. 17
8: end for
9: return $C$ (without overlapping)
```

Algorithm 4 Enclosing (C)

```
1: $C$ : set of $Circles$
2: $P_g = (x_g, y_g) =$ Center of Gravity of $C$
3: $Sort\ C$ by descending distance to $P_g$
4: $c_0 \leftarrow c_1$
5: for $i = 2$ to $|C|$ do
6:     if $c_0$ does not enclose $c_i$ then
7:         $Generate$ a new $Container$ $c_0$, according to Eqs. 18 – 23
8:     end if
9: end for
10: return $c_0$
```

Once we know the new radius, we compute the changes in the x and y coordinates of c_0's center.

$$\theta_{0i} = \mathrm{ArcTan}\frac{x_0 - x_i}{y_0 - y_i} \tag{19}$$
$$dx_0 = (r'_0 - r_0)\mathrm{Cos}(\theta_{0i}) \tag{20}$$
$$dy_0 = (r'_0 - r_0)\mathrm{Sin}(\theta_{0i}) \tag{21}$$

where dx_0 and dy_0 are the changes to c_0's coordinates. Finally, the new position of c_0 is given by Eqs. 22 and 23.

$$x'_0 = x_0 + dx_0 \tag{22}$$
$$y'_0 = y_0 + dy_0 \tag{23}$$

Note that as Algorithm 4 proceeds towards the center of gravity, the likelihood to modifying the container circle decreases. When have revised that all circles are inside of the enclosing circle c_0, the process has finished and the container circle c_0 is returned.

4.5 Time complexity

It is well known that sorting and computing the Delaunay triangulation take time $O(N \log N)$ (Lee and Schachter 1980). The loop in lines 6–8 of Algorithm 4 repeats $O(N_t)$ times, and repair can be done in constant time, and since $N_t = O(N)$. The total times complexity of DTRepair is $O(N \log N)$.

Enclosing also sorts the sets of circles, taking $O(N \log N)$ time. The loop in lines 5–9 of Enclosing repeats $O(N)$ times, and verification of enclosure and modifying the container can be done in constant time.

The total time complexity of Enclosing is $O(N \log N)$. In conclusion, given the chromosome of an individual in the evolutionary process, repairing it and measuring its fitness (i.e. determining the enclosing circle) can be done in $O(N \log N)$ time.

5 Results

ECPP was tested under different conditions, and the results were compared to known solutions (either exact or approximate) known in the field. The problem of packing circles in a circle has two known variants, the case where all circle sizes are equal (known as unit circle packing), and the case where the circles have different sizes (Castillo et al. 2008). ECPP was tested following this distinction.

The first set of experiments tests ECPP's performance on unit-circle problems. These tests were conducted in two orthogonal directions; we compare the performance of two metaheuristic search algorithms (namely, GA and DE)[1] using the repulsion based and the Delaunay triangulation repair heuristics. These results can be compared against the proven optimal analytical solutions.

The second set of experiments tests ECPP's performance on uneven circle problems. Since there are no known optimal results for these problems, we performed two experiments to test ECPP's performance: the first one compares our results with those published at Mathrec (Zimmermann 2006), the second one analyzes ECPP's performance for different values of variance of the radii of the circles. We analyze the algorithm's performance with respect to the solutions' packing density; when the variance of the circles' radii in a problem instance increases, large circles produce holes that can be occupied by small circles, thus increasing the density of the produced solution.

Those tests are reported in the following two subsections.

ECPP was implemented in Mathematica 9. We ran our experiments using a Mac computer with an Intel Core i7, 2 GHz, four cores, and 4 GB of RAM.

5.1 Unit circle tests

To measure the performance of the two repair mechanisms, we compare the numerical performance of four metaheuristics (genetic algorithms, evolutionary strategies, differential evolution, and PSO) applied to the circle packing problem. The most recent solutions to the circle packing problem are reported at the website Packomania (Specht 1999). This website started in 1999 reporting proven optimal solutions from 1 to 1,500 unit circles and some cases of uneven circles. Unfortunately, they do not provide computational time. To illustrate the effectiveness of our approaches, we tested problems of different sizes, from 3 circles to 100 circles. Only some of them are reported for the sake of brevity. We compare the results produced by our algorithms with the benchmarks presented in the Packomania web site (Specht 1999).

The metrics used in the comparison of results are radius and density. Radius is the best-known solution proven mathematically for a particular problem and density describes the ratio of total area occupied by the circles to container area, i.e., the relationship between the area of the container circle and the sum of areas of the circles that are inside it. Benchmarks are registered on Tables 1 and 2 in the column labeled Optimal, with their corresponding radii and densities.

The results of the two repair mechanisms (repulsion-based repair and DT-based repair) are presented in Sect. 5.1 and 5.2, respectively. These tables compare our approaches with the benchmarks. The results are shown for the size of the instances, the radius of the container circle and the density corresponding to the container circle of the mean solution obtained with genetic algorithms (GA) and differential evolution (DE).

5.1.1 Results using repulsion-based repair

Table 1 is composed of eight columns: the first column refers to the number of unit circles to be packed; the second and third columns refer to the radius and density, respectively, of the optimal solution; the fourth and fifth columns show the mean radius and density obtained applying GA, the sixth and seventh columns show the mean radius and density obtained applying DE, the eighth column shows the mean time required. The results displayed in Table 1 represent a series of instances that take values for $n = 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100$. The results produced by our approach were obtained by 30 independent executions with random initial populations. The table shows the mean of the 30 executions. The values highlighted in each row represent the best mean obtained.

From Table 1, we can see that DE outperforms the other metaheuristics. This result was at all expected; it is vox populi in the Evolutionary Computation community, that DE is one of the best metaheuristics for real-valued optimization problems. Table 1 also shows, in the last column, the computational time required to solve each problem size. Since both GA and DE were run with the same population size and number of generations, their computational times are basically the same.

Even though the mean results not seem to show clear superiority in most cases, we decided to perform a statistical comparison of the result for selected cases. For size 30, we compared the performance of GA and DE, for a statistical difference in means, for a significance level $\alpha = 0.05$, proving that indeed performance is superior to DE. For $n = 55$, we performed the same test, resulting in DE's performance proving superior to GA.

Figure 3 illustrates the plot formed by the mean values obtained for the solution for the four metaheuristics and the

[1] Although all experiments were also performed using genetic algorithms (GA), evolutionary strategies (ES), differential evolution (DE), and PSO, for the sake of brevity, only the GA and DE were included in the reports, since they were the ones that performed the best.

Table 1 Performance using repulsion-based repair

N	Optimal		GA		DE		Time (s)
	r_0	Density	r_0	Density	r_0	Density	
2	2.000000	0.500000	2.000000	0.500000	2.000000	0.500000	34
3	2.154700	0.646171	2.159630	0.643222	2.159420	0.643347	792
4	2.414213	0.686294	2.459890	0.661043	2.415430	0.685601	1,760
5	2.701301	0.685211	2.949650	0.574686	2.831360	0.623705	2,528
6	3.000000	0.666667	3.076380	0.633975	3.000000	0.666667	4,240
7	3.000000	0.777778	3.265700	0.656367	3.108350	0.724498	5,472
8	3.304764	0.732504	3.543870	0.636993	3.430100	0.679950	11,024
9	3.613125	0.689406	3.872370	0.600190	3.777500	0.630714	14,720
10	3.813025	0.687796	4.089480	0.597948	3.941590	0.643660	14,816
11	3.923804	0.714462	4.218920	0.618003	4.132690	0.644062	14,940
12	4.029601	0.739022	4.434350	0.610269	4.285840	0.653294	14,976
13	4.236067	0.724464	4.614080	0.610622	4.433790	0.661291	15,068
14	4.328428	0.747252	4.837260	0.598314	4.678800	0.639528	15,276
15	4.521356	0.733758	4.992760	0.601740	4.900330	0.624656	15,400
16	4.615425	0.751096	5.185520	0.595024	5.037030	0.630624	15,408
17	4.792033	0.740304	5.316410	0.601468	5.193660	0.630235	15,520
18	4.863703	0.760920	5.491150	0.596961	5.293990	0.642254	15,696
19	4.863703	0.803193	5.586120	0.608883	5.432230	0.643869	15,972
20	5.122320	0.762249	5.738290	0.607387	5.591260	0.639749	16,596
25	5.752824	0.755408	6.385920	0.613046	6.312330	0.627423	17,516
30	6.197741	0.781016	6.998590	0.612491	6.959850	0.619328	18,468
35	6.697170	0.780343	7.621730	0.602505	7.483660	0.624942	19,128
40	7.123850	0.788190	8.126000	0.605768	8.000540	0.624916	20,728
45	7.572910	0.784669	8.559630	0.614189	8.478760	0.625962	23,992
50	7.947515	0.791604	9.011700	0.615682	8.948360	0.624430	29,264
55	8.211100	0.815755	9.440760	0.617090	9.333180	0.631399	55,460
60	8.646220	0.802599	9.870930	0.615794	9.865880	0.616425	78,848
65	9.017400	0.799376	10.342000	0.607721	10.179400	0.627294	98,532
70	9.345650	0.801454	10.665700	0.615343	10.492600	0.635815	117,804
75	9.672029	0.801728	10.943500	0.626257	10.939800	0.626674	122,044
80	9.968150	0.805120	11.352900	0.620693	11.274100	0.629403	128,716
85	10.163100	0.822935	11.683800	0.622654	11.543200	0.637923	139,332
90	10.546100	0.809210	11.992200	0.625815	11.953100	0.629918	142,992
95	10.840200	0.808442	12.244100	0.633680	12.195400	0.638755	157,080
100	11.082149	0.814241	12.612100	0.628675	12.506500	0.639337	190,244

benchmarks corresponding to each problem size (i.e., number of circles). Figure 4 illustrates the plot formed by the Density obtained by the mean values of the fitness function for the four metaheuristics and the benchmarks corresponding to each problem size (i.e., number of circles). From Figs. 3 and 4, we can see that the best densities were obtained by DE for most cases. Those figures show the distance of ECPP's solutions with respect to the optimal ones.

5.1.2 Results using DT-based repair

Table 2 has the same structure as Table 1. The results displayed in Table 2 represent a series of instances that take values for $n = 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100$. The results produced by our approach were obtained by 30 independent executions with random initial

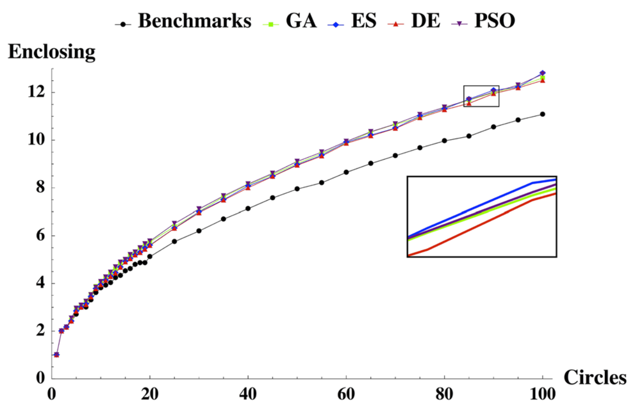

Fig. 3 Radius of the enclosing circle using repulsion-based repair

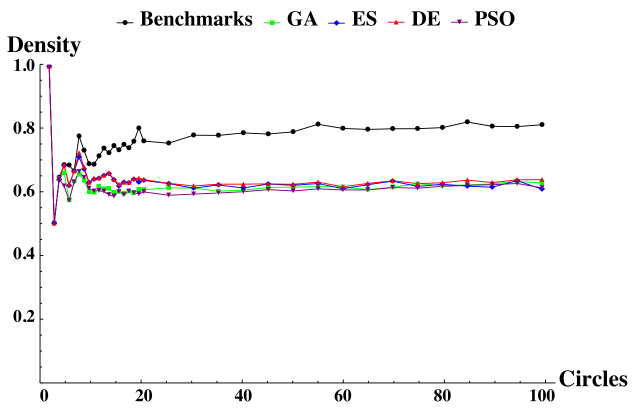

Fig. 4 Density of the enclosing circle using repulsion-based repair

solutions. The table shows the mean of the 30 executions. The values highlighted in each row represent the best result obtained.

From Table 2, we can see that GA outperforms the other metaheuristics. This result was not at all expected; it is vox populi in the evolutionary computation community that DE is one of the best metaheuristics for real-valued optimization problems. The best metaheuristic was GA. As of today, the reason remains unknown to the authors of this paper.

Figure 5 illustrates the plot formed by the mean values obtained for the solution for the four metaheuristics and the benchmarks corresponding to each problem size (i.e., number of circles). From Fig. 5, we can see that the best solutions were obtained by GA.

Figure 6 illustrates the plot formed by the Density obtained by the mean values of the fitness function for the four metaheuristics and the benchmarks corresponding to each problem size (i.e., number of circles). From Fig. 6, we can see that the best densities were obtained by GA.

5.2 Non-unit circle tests

Although the unit circle version of CPP has been solved analytically for as many as 1,014 circles (Specht 1999), there is an infinite number of instance problems for each number of circles in the non-unit circle case, and there is no analytical solution for any of these problems.

In the absence of analytical solutions, to assess the performance of ECPP with non-unit circle problem instances, we performed two sets of experiments. First, we compared ECPP's performance on the benchmark problems presented at Zimmermann (2006). The problem presented in those web pages is to solve CPP for a set of circles where their radii $r_i = i, i = 1, \ldots, N$. Figure 7 shows a plot of the radii of the enclosing circles of solutions obtained using Delaunay triangulation-based repair with genetic algorithms for problem sizes from 1 to 50. GA was executed with a population of 50 individuals and 500 generations.

The interpretation of these results has to take into account the following facts. The plot labeled as MathRec is the world records for these problems; no single algorithm provides the best solution to all problem instances. Even more, not all participants provided solutions to all problem instances. In general, algorithms are best at a single solution or at most at a range of them. None of the solutions presented in the MathRec contest was based on evolutionary computation. ECPP is the first attempt to solve CPP using metaheuristics of this kind.

In the second experiment to assess ECPP's performance on non-unit CPP, we control the variance of the circle sizes and observe ECPPs performance (based on packing density) as the relative sizes of the circles vary. These experiments were conducted using only the best metaheuristic search and repair mechanisms found in the previous tests, i.e., genetic algorithms, using the Delaunay triangulation repair. For the sake of brevity, we only present experiments conducted to a set of 20 circles, which represent a considerably large search space of 40 dimensions.

We allow circle sizes to diverge from unity in such a way that we have control over the variance of their sizes. If the variance is zero, we have the unit-circle version of the problem, which was tested extensively and the results were reported in the previous section. As the variance increases, intuition leads us to suspect that smaller circles way fill in the gaps between larger circles, therefore, increasing the density of the solutions.

For each problem size (i.e., number of circles), we generate the circle sizes according to the following probability distribution function:

$$U[1 - \Delta, 1 + \Delta] \tag{24}$$

where Δ represents the allowed amount of variation. It is well known that the variance of this distribution is

$$\sigma^2 = \frac{1}{12}((1 + \Delta) - (1 - \Delta))^2 = \frac{\Delta^2}{3} \tag{25}$$

Table 2 Performance using Delaunay triangulation-based repair

N	Optimal		GA		DE		Time (s)
	r_0	Density	r_0	Density	r_0	Density	
2	2.000000	0.500000	2.000000	0.500000	2.000000	0.500000	34
3	2.154700	0.646170	2.154700	0.646171	2.154700	0.646171	204
4	2.414213	0.686291	2.732050	0.535898	2.732050	0.535898	408
5	2.701301	0.685210	3.000000	0.555556	3.000000	0.555556	680
6	3.000000	0.666666	3.000000	0.666667	3.000000	0.666667	1,054
7	3.000000	0.777777	3.000000	0.777778	3.000000	0.777778	1,190
8	3.304764	0.732502	3.645750	0.601888	3.645750	0.601888	2,380
9	3.613125	0.689407	3.800000	0.623269	3.800000	0.623269	2,958
10	3.813025	0.687797	4.000000	0.625000	4.000000	0.625000	3,162
11	3.923804	0.714460	4.055050	0.668960	4.055050	0.668960	3,808
12	4.029601	0.739021	4.055050	0.729775	4.055050	0.729775	4,216
13	4.236067	0.724465	4.550800	0.627724	4.573610	0.621477	4,556
14	4.328428	0.747252	4.605550	0.660032	4.605550	0.660032	4,896
15	4.521356	0.733759	4.752780	0.664043	4.752780	0.664043	5,508
16	4.615425	0.751097	4.815756	0.689908	4.815760	0.689908	5,984
17	4.792033	0.740302	5.000000	0.680000	5.000000	0.680000	6,290
18	4.863703	0.760918	5.000000	0.720000	5.000000	0.720000	6,766
19	4.863703	0.803192	5.187680	0.706005	5.187680	0.706005	7,480
20	5.122320	0.762248	5.467710	0.668990	5.455490	0.671990	7,786
25	5.752824	0.755401	6.023700	0.688991	6.003700	0.693588	9,860
30	6.197741	0.781006	6.465700	0.717612	6.434870	0.724505	10,302
35	6.697170	0.780343	7.206830	0.673875	7.056770	0.702840	10,472
40	7.123850	0.788190	7.599570	0.692600	7.600690	0.692396	12,920
45	7.572910	0.784669	8.000000	0.703125	8.062160	0.692325	14,212
50	7.947515	0.791602	8.550590	0.683877	8.453270	0.699715	16,796
55	8.211100	0.815755	8.825820	0.706078	8.899030	0.694509	17,850
60	8.646220	0.802599	9.120590	0.721282	9.264870	0.698992	21,590
65	9.017400	0.799376	9.568860	0.709893	9.728280	0.686817	21,760
70	9.345650	0.801454	9.888930	0.715813	10.051700	0.692814	23,834
75	9.672029	0.801726	10.379800	0.696120	10.613000	0.665866	26,690
80	9.968150	0.805120	10.718000	0.696408	10.861600	0.678115	28,798
85	10.163100	0.822935	11.057300	0.695221	11.113700	0.688183	30,396
90	10.546100	0.809210	11.574400	0.671812	11.582900	0.670823	32,708
95	10.840200	0.808442	11.748100	0.688311	11.927700	0.667744	33,626
100	11.082149	0.814239	12.067400	0.686704	12.433700	0.646843	36,176

so, by varying Δ we are controlling the variance of the circle sizes of the randomly generated problem instances.

We generated random problem instances for 20 circles varying Δ in the interval from 0 to 1, increasing delta by 0.1 for each experiment. For each setup, we performed 30 independent executions. This number of executions allows us to achieve statistical stability and draw conclusions about ECPP's performance on this type of problems. Figure 8 shows a plot of the density of the solutions as Δ varies for the described experiments.

6 Discussion

Comparing the four meta-heuristics using Delaunay triangulation-based repair, we can see that genetic algorithms perform best, followed closely by differential evolution, evolutionary strategies being further back, and in last place PSO.

As we can see in Fig. 5, the difference between genetic algorithms and differential evolution is minimal and perhaps one might think that these results do not show obvious differences. However, statistical tests were performed for 55 to 100

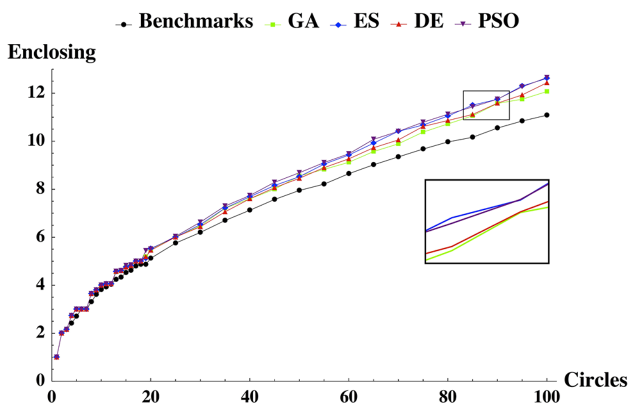

Fig. 5 Radius of the enclosing circle using Delaunay triangulation-based repair

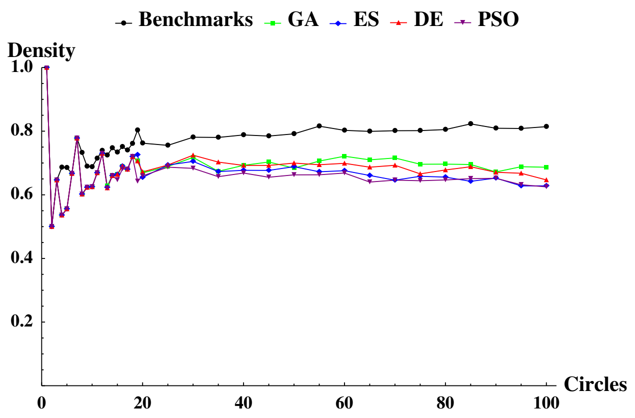

Fig. 6 Density of the enclosing circle using Delaunay triangulation-based repair

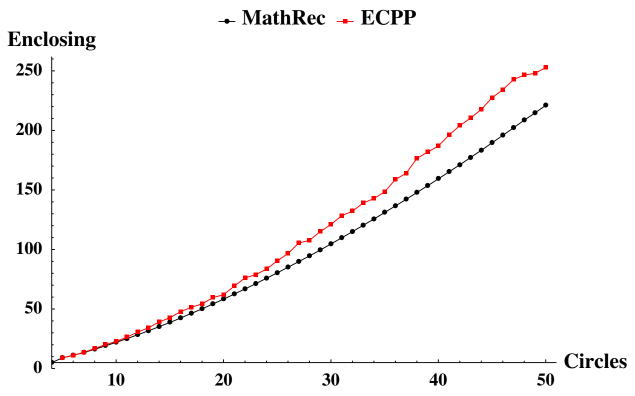

Fig. 7 Performance comparison: ECPP vs MathRec

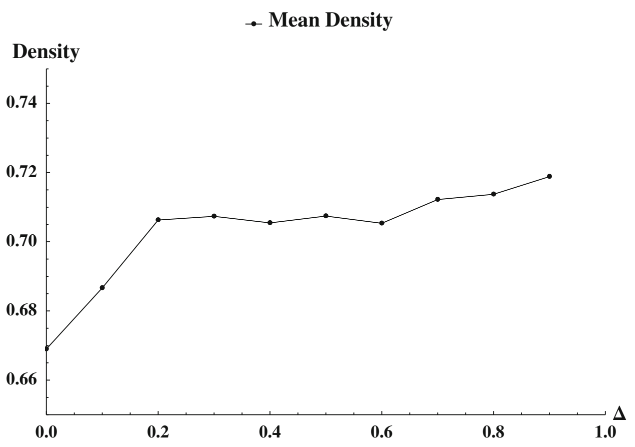

Fig. 8 Mean density of the enclosing circle with uneven circles

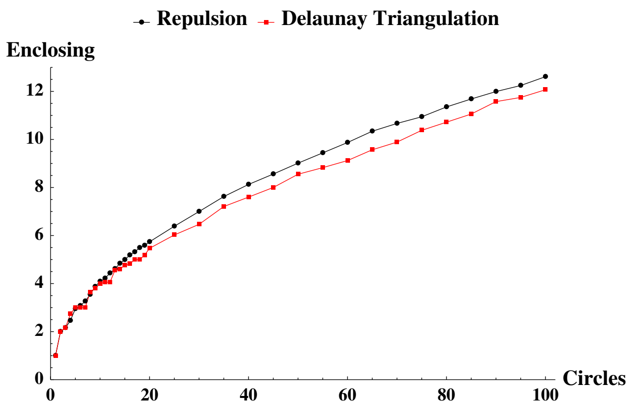

Fig. 9 Comparison between repulsion-based repair and Delaunay triangulation-based repair by genetic algorithms

circles, and there are indeed statistically significant differences between their respective means. For example for $n = 55$, we conducted a hypothesis test with a level of significance of 95 %, where the null hypothesis was $H_0{:}\mu_{\mathrm{GA}} = \mu_{\mathrm{DE}}$ and the alternative hypothesis is $H_1{:}\mu_{\mathrm{GA}} \neq \mu_{\mathrm{DE}}$. The calculated value was $p = 0.074016$, which indicates that there is indeed a difference. Therefore, if there is a statistically significant difference between genetic algorithms and differential evolution which were the closest ones, there is also a statistically significant difference between genetic algorithms and evolutionary strategies or PSO, whose means are more distant.

To date, the authors have no explanation why genetic algorithms outperform differential evolution when applying the Delaunay triangulation-based repair, even though most of the published literature mention that differential evolution performs better than genetic algorithms in real-coded problems.

To compare the results obtained by the repulsion and Delaunay triangulation-based repair processes, we plot the mean of size of the enclosing circle for the different problem sizes. Figure 9 shows that the Delaunay triangulation repair performs better than the repulsion-based repair. Also, the repairing process by Delaunay triangulation requires less time that means less evaluations are needed to obtain a result closer to the optimum than by the repulsion heuristic.

In very few cases, where the optimal configuration is a configuration that cannot be obtained by the Delaunay triangulation-based repair, this repair method has trouble approaching the optimum. This situation occurs when the circles in the optimal configuration form holes involving more than three circles, and sometimes with one or two circles in the hole, i.e. some circles are isolated. This repairing process forces the circles to be tangent, so it is impossible to approach the optimum. In those cases, the repulsion-based repair is more feasible to obtain a closer result to the optimum.

In the last test cases, dealing with uneven circles, the expected results were achieved. Starting with no variation, the results depart from the densities reached by the unit-circle problem. As the variance of the circle sizes increases, the density of the solutions increases. The density of the solutions seems to tend asymptotically to a limit which remains to be determined in future experiments.

7 Conclusions

In this paper, we propose an evolutionary computation-based solution to the circle packing problem, called ECPP. These solutions are based on two repair mechanisms: repulsion based and Delaunay triangulation based. We combine these repair heuristics with GA, ES, DE, and PSO (although only GA and DE are reported in this paper). The solution space of the circle packing problem is enormous and increases rapidly with the problem size, i.e., the number of circles to be allocated. Since these packing problems are NP-complete, heuristic search procedures are used.

The strength of evolutionary algorithms lies in the ability to search large and complex solution spaces in a systematic and efficient way. Evolutionary search strategies are not dependent on a particular problem structure and allow the user to use different methods for the encoding of the genotype. The performance of the search process is strongly related to the representation of the circle packing problem. The particular feature of our evolutionary algorithm developed for the circle packing problem is their two-stage approach. ECPP is used to explore and manipulate the solution space, and a second procedure is used to evaluate the solutions. The genotype needs to be repaired to check the quality and feasibility of the packing solution: the phenotype.

The operation on the layout rather than an encoded data structure raises a number of other issues, such as overlap. Overlapping configurations are invalid solutions and need to be resolved by rejecting, correcting, or temporarily accepting them. Rejection wastes precious computation time and may result in less dense layouts for highly uneven radii, since the slightest change in position or rotation could lead to invalid configurations, which will no longer contribute to the search process; after the repair process is applied to a prospect solution, it results in a valid solution. Correcting invalid configurations seems a better option, since often only minor repositioning is necessary to obtain a valid solution.

The acceptance of an invalid layout requires a penalty term in the evaluation function, as mentioned in the solutions presented in the literature. The penalty expression needs to be carefully designed balancing between layout compaction and overlap generation. Penalty functions are a less efficient guide to the search than a repair algorithm that avoids producing constraint violating configurations.

When the search process operates on an encoding, the packing rules applied by the decoding algorithm guarantee that all solutions considered in the search process are valid. There has been much speculation on whether this is beneficial with respect to the transmission of specific layout to the next generation and the next state in the neighborhood, respectively. We have not been able to find a satisfactory answer to this problem in the literature. The different solution approaches have not been compared with each other. Since much of their performance strongly depends on radii and the number of circles involved with respect to the formulation of the objective function, it is not sufficient to judge their performance purely on the basis of benchmarks achieved. This emphasizes the need for including density in the relative evaluation of the solutions presented in this paper.

The numerical results show that all methods can find acceptable solutions and their performances will be very close if the Delaunay triangulation repair is used. We can also observe that differential evolution performs better than the other three metaheuristics when a repulsion-based repair is applied and genetic algorithms perform better when a Delaunay triangulation-based repair is applied.

Our findings clearly show that ES and PSO provide the worst results of the four metaheuristics with the two repair mechanisms on most of the problem instances. GA and DE got better results, using either of the repair mechanisms. In some cases, the repulsion-based repair gives us better results than the DT-based repair. This occurs particularly because the configuration of the best-known solution is not formed by triplets of circles tangent to each other. Nonetheless, in general terms, the performance of DT-based repair is better, specifically given that this repair process forces the circles to be tangent with at least two other circles. Probably, a hybrid algorithm composed with the two repair mechanism could give us better results than those obtained with each heuristic separately, specifically if we apply the repulsion-based repair followed by the DT-based repair.

Finally, the last set of experiments shows that ECPP is able to handle problem instances with circles of different sizes, as effectively as for the unit circle instances. This result was expected, since the solution was designed without any size-related constraint. ECPP was compared with the results published at MathRec. Those results represent the state of the art in solving that specific problem instance, and no single algorithm provides the best solution to all problem sizes. Even more, not all participants provided solutions to all problem instances. In general, algorithms are best at a single solution or at most at a range of them. Furthermore, none of the solutions presented in the MathRec contest was based on evolutionary computation; to the authors' knowledge, ECPP is the first attempt to solve CPP using metaheuristics of these kind.

Future work will focus on the packing of polygons on rectangles and strips. So far, we have not found any evolutionary computation-based solution to the CPP that had been previously published; therefore, we cannot objectively compare with other results of metaheuristic search methods applied to solve CPP.

A compendium of the experiments with detailed results can be found at http://dep.fie.umich.mx/CPP.

References

Addis B, Locatelli M, Schoen F (2008) Efficiently packing unequal disks in a circle. Oper Res Lett 36(1):37–42

Al-Modahka I, Hifi M, M'Hallah R (2011) Packing circles in the smallest circle: an adaptive hybrid algorithm. J Oper Res Soc 62(11):1917–1930

Castillo I, Kampas FJ, Pintér JD (2008) Solving circle packing problems by global optimization: numerical results and industrial applications. Eur J Oper Res 191(3):786–802

Deb K (2004) A population-based algorithm-generator for real-parameter optimization. Soft Comput 9:236–253

Demaine ED, Fekete SP, Lang RJ (2010) Circle packing for origami design is hard. arXiv:1008.1224

Dowsland KA, Gilbert M, Kendall G (2007) A local search approach to a circle cutting problem arising in the motor cycle industry. J Oper Res Soc 58(4):429–438

Francesco C, Cerrone C, Cerulli R (2014) A tabu search approach for the circle packing problem. In: 17th international conference on network-based information systems

Harary F, Randolph W, Mezey PG (1996) A study of maximum unit-circle caterpillars-tools for the study of the shape of adsorption patterns. Discret Appl Math 67(1–3):127–135

Hifi M, M'Hallah R (2009) Beam search and non-linear programming tools for the circular packing problem. Int J Math Oper Res 1(4):476–503

Lee D-T, Schachter BJ (1980) Two algorithms for constructing a delaunay triangulation. Int J Comput Inf Sci 9(3):219–242

Michalewicz Z (1996) Genetic algorithms+datastructures=evolutionary programs, 3rd edn. Springer, Berlin

Mladenovic N, Plastria F, Urosevic D (2005) Reformulation descent applied to circle packing problems. Comput Oper Res 32(9):2419–2434

Nordbakke MW, Ryum N, Hunderi O (2004) Curvilinear polygons, finite circle packings, and normal grain growth. Mater Sci Eng A 385(1–2):229–234

Pintér JD, Kampas FJ (2006) Nonlinear optimization in mathematica with math-optimizer professional. Math Educ Res 10:1–18

Shi Y-J, Liu Z-C, Ma S (2010) An improved evolution strategy for constrained circle packing problem. In: Advanced intelligent computing theories and applications. Springer, Berlin, pp 86–93

Specht E (1999) Packomania web site. http://www.packomania.com/

Sugihara K, Sawai M, Sano H, Kim DS, Kim D (2004) Disk packing for the estimation of the size of a wire bundle. Jpn J Ind Appl Math 21(3):259–278

Szabó PG, Markót MC, Csendes T, Specht E, Casado LG, García I (2006) New approaches to circle packing in a square. Springer, Berlin

Wang H, Huang W, Zhang Q, Xu D (2002) An improved algorithm for the packing of unequal circles within a larger containing circle. Eur J Oper Res 141(2):440–453

Wenqi H, Yan K (2004) A short note on a simple search heuristic for the diskspacking problem. Ann Oper Res 1–4:101–108

Xu Y-C, Xiao R-B, Amos M (2007) A novel genetic algorithm for the layout optimization problem. In: Evolutionary computation 2007 CEC 2007, pp 3938–3943

Yan-Jun S, Yi-Shou W, Long W, Hong-Fei T (2012) A layout pattern based particle swarm optimization for constrained packing problems. Inf Technol J 11:1722–1729

Zhang DF, Deng AS (2005) An effective hybrid algorithm for the problem of packing circles into a larger containing circle. Comput Oper Res 32(8):1941–1951

Zhi-Qin Q, Hong-Fei T, Zhi-Guo S (2001) Human–computer interactive genetic algorithm and its application to constrained layout optimization. Chin J Comput 5:553–560

Zimmermann A (2006) Al zimmermann's programming contests. http://www.recmath.org/contest/CirclePacking/index.php