A Stochastic Continuous Optimization Backend for MiniZinc with Applications to Geometrical Placement Problems

Thierry Martinez, François Fages, Abder Aggoun

A Stochastic Continuous Optimization Backend for MiniZinc with Applications to Geometrical Placement Problems

Thierry Martinez[1] and François Fages[1] and Abder Aggoun[2]

[1] Inria Paris-Rocquencourt, Team Lifeware, France
[2] KLS-Optim, France

Abstract. MiniZinc is a solver-independent constraint modeling language which is increasingly used in the constraint programming community. It can be used to compare different solvers which are currently based on either Constraint Programming, Boolean satisfiability, Mixed Integer Linear Programming, and recently Local Search. In this paper we present a stochastic continuous optimization backend for MiniZinc models over real numbers. More specifically, we describe the translation of FlatZinc models into objective functions over the reals, and their use as fitness functions for the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) solver. We illustrate this approach with the declarative modeling and solving of hard geometrical placement problems, motivated by packing applications in logistics involving mixed square-curved shapes and complex shapes defined by Bézier curves.

1 Introduction

MiniZinc [11] is a medium-level constraint modeling language which is becoming a standard in the Constraint Programming community. It is high-level enough to express most constraint problems easily, but low-level enough to be mapped onto existing solvers easily and consistently. This mapping is done through a flattening process which takes as input a MiniZinc instance and produces a FlatZinc instance. FlatZinc is a low-level solver input language designed to be easy to translate into the form required by a solver. It is chosen for that reason as target language for MiniZinc.

Currently, there exist FlatZinc backends for Mixed Integer Linear Programming (CPLEX, OR-tools[1], SCIP, ...), Finite Domain Constraint Programming solvers (Choco[2], Eclipse[3], Gecode[4], JaCoP, Opturion-CPX[5], Oscar, SICStus prolog, ...), SAT solvers (MinisatID,...) and recently Local Search (iZplus[6], Oscar-cbls [1]).

[1] `https://code.google.com/p/or-tools/`
[2] `https://github.com/chocoteam/choco-parsers`
[3] `http://eclipseclp.org/doc/bips/lib_public/flatzinc/`
[4] `http://www.gecode.org/flatzinc.html`
[5] `http://www.opturion.com/cpx`
[6] `http://www.minizinc.org/challenge2014/descriptionizplus.txt`

Most of FlatZinc implementations are thus dedicated to discrete domains. However, constraint optimization and decision problems over real numbers can be expressed in MiniZinc with high generality. Curently, such continuous constraint problems can be solved either using Linear Programming backends, with restrictions on the linearity of the constraints, or using interval arithmetic backends (e.g. G12ic, Eclipse fzn_ic).

In this paper, we study another kind of solver based on *stochastic continuous optimization* for solving FlatZinc instances over real numbers, using namely the *Covariance Matrix Adaptation Evolution Strategy* (CMA-ES) [6]. More specifically, we show how a FlatZinc instance over real numbers can be translated into a fitness function which can be directly used by CMA-ES to compute approximate solutions to the problem. The transformation we describe is quite general and applies virtually to any MiniZinc model over real numbers. The choice of CMA-ES among other evolutionary or particle swarm optimization algorithms is motivated by the absence of parameterization for this algorithm and by its performances on hard problems.

For discrete domains, there has been related work on the design of high-level constraint-based modeling languages for local search and genetic algorithms. The seminal work of Van Hentenryck and Michel on Comet [10,7] showed how a finite domain constraint model can be compiled into an objective function for local search metaheuristics, such as Tabu search, with default neighborhoods derived from the constraint model. In [1], Björdal et al. present a constraint-based local search backend for MiniZinc and show that it produces competitive results on the 2010 to 2014 MiniZinc challenges. In these systems, the local search solver is limited to finite domain constraints and use neighborhoods derived from the finite domains of the variables.

Here in the continuous domain, we illustrate our CMA-ES backend for FlatZinc with the solving of hard geometrical placement problems which, to the best of our knowledge, go beyond the state-of-the-art of declarative constraint modeling and solving. As a matter of fact, the only FlatZinc implementations listed on the MiniZinc web page that parse the FlatZinc instances presented in this paper are those based on exact methods using interval arithmetic (i.e. Eclipse fzn_ic and G12ic) but none of them can find solutions in reasonable computation time even for the examples presented here. In [9], we have already shown that the non-overlap constraint between squares, cubes, rectangles, boxes, triangles, polygons circles and spheres, can be associated with a *measure of overlap* between objects which can be used directly as a fitness function in CMA-ES for packing mixed shapes in a bin, with an interesting trade-off between generality and efficiency. The measure of overlap does not need to be the area of the intersection (and should not if one object can be included in another) but can be any measure equal to 0 in case of non-overlap, and capable of guiding the continuous optimization solver by measuring progress toward satisfaction [4]. On a benchmark of consecutive sizes circle packing problems, we showed that CMA-ES finds solutions at 2% of the best known costs obtained by running the three global optimization methods reported in Castillo et al. [3]. In [12], Salas

and Chabert show that the overlap measures which were defined in an *ad hoc* manner in [9], can be computed by interval methods in IBEX[7] with a numerical algorithm that automatically measures the *penetration depth* of two objects of virtually any shape defined by conjunction and disjunction of non-linear inequalities. In this paper, we give general MiniZinc definitions for the penetration depths, or simpler overlap measures, between polygons, circles, and also *complex shapes defined by Bézier curves*, motivated by packing problems in the cosmetic and automotive industries. This illustrates the performance of MiniZinc-CMAES in terms of both declarative modeling and efficient (yet suboptimal) resolution of very hard geometrical packing problems with complex shapes and continuous rotations.

The rest of the paper is organized as follows. In the next section, we present the translation of a FlatZinc instance over real numbers in a fitness function over the reals, and the interface to the CMA-ES solver. In Section 4 we describe MiniZinc models of continuous packing problems involving continuous rotations, mixed square-curved shapes and complex shapes defined by Bézier curves. There we use some simple distance formulae for circles, Minkowski sums for the penetration depth between polygons [5] and De Casteljau's numerical algorithm for linearizing Bézier curves. In Section 5, we report on the performance results obtained through the compilation chain from MiniZinc, FlatZinc to CMA-ES, on complex shape packing problems. Finally, we conclude on the general perspective opened by this MiniZinc backend for continuous optimization and novel applications at the intersection of Optimization and Computer-Aided Design.

2 Compiling FlatZinc Instances over Real Numbers in Real-valued Fitness Functions

In this section we describe our transformation of a FlatZinc instance containing arithmetic and trigonometric constraints over float variables in a fitness function which aggregates the costs of each constraint violation. This transformation is at the heart of the continuous optimization backend.

2.1 Arithmetic expressions

The arithmetic expressions that constitute the constraint satisfaction problem need be rebuilt from the FlatZinc instance, since arithmetic sub-expressions and intermediary variables are introduced by the transformation from MiniZinc to FlatZinc. The constraints that result from this transformation are split in three groups:

1. inequality constraints, which are turned into *costs*,
2. arithmetic and trigonometric constraints, which always appear to be directed and are turned into *functional expressions*,

[7] `http://www.ibex-lib.org`

3. and equality constraints, which can either be solved statically if there exists a topological sort of the constraint graph that makes the constraint directed, or turned into a cost otherwise.

Every variable X of the model is associated to an expression $[X]$ defined as $[X] = X$ if X is one of the search variables, or as the arithmetic or trigonometric expression deduced from the constraints on X. The notation is extended to float constants, $[f] = f$, and vectors of variables and/or float constants, i.e. for a vector $\boldsymbol{U} = (X_1, \ldots, X_n)$, $[\boldsymbol{U}]$ denotes the vector $([X_1], \ldots, [X_n])$.

Concerning the first group, the inequality constraints in FlatZinc are either strict or non-strict inequalities between linear expression of the form:

```
constraint float_lin_lt(U0,U1,K);
constraint float_lin_le(U0,U1,K);
```

where K is a constant, U_0 is a vector of constant coefficient and U_1 is a vector of variables. The semantics is respectively $\boldsymbol{U}_0 \cdot \boldsymbol{U}_1 < K$ and $\boldsymbol{U}_0 \cdot \boldsymbol{U}_1 \leq K$ where $\cdot$ denotes the scalar product. The cost we associate to such an inequality constraint c is

$$\text{cost}(c) = \max(0, [\boldsymbol{U}_0] \cdot [\boldsymbol{U}_1] - K)$$

where $[\boldsymbol{U}_0]$ and $[\boldsymbol{U}_1]$ are the expressions constructed from the arguments. For strict inequality, one could refine the cost to

$$\text{cost}(c) = \begin{cases} 1 + [\boldsymbol{U}_0] \cdot [\boldsymbol{U}_1] - K & \text{if } [\boldsymbol{U}_0] \cdot [\boldsymbol{U}_1] \geq K \\ 0 & \text{otherwise} \end{cases}$$

in order to ensure that the cost is null if and only if the constraint is satisfied. However, this is not necessary to guide the search for solutions since the constraints $a < b$ and $a \leq b$ are equivalent almost everywhere in a continuous setting.

Concerning the second group, every float variable X of the model is considered as a search variable, except if it appears in the result position of one of the following directed constraint:

```
constraint float_min(A, B, X);
constraint float_max(A, B, X);
constraint float_times(A, B, X);
constraint float_sqrt(A, X);
constraint float_cos(A, X);
constraint float_sin(A, X);
```

If X is in the result position of one of these constraints, then $[X]$ is defined as the expression that computes the associated value.

For the third group, FlatZinc linear equality constraints are of the form:

```
constraint float_lin_eq(U0,U1,K) :: defines_var(X) :: weight(w);
```

The FlatZinc compiler generates the annotation `defines_var(X)` which directs the constraint from its arguments to its result, in the case where the constraint results from a linear arithmetic expression. In that case, $[X]$ is defined as the expression that computes the linear combination.

For the equality constraints that result from reification, of the form:

```
constraint float_eq_reif(X,Y,B);
constraint float_lin_eq_reif(X,U,B);
```

we eliminate them statically if enough information is available. Otherwise, the model is currently rejected.

It is worth noting that this minimal handling of reification is needed to cope with the code generated by the MiniZinc compiler for partial functions like `sqrt`, where the formal argument is unified with the actual argument only if the function is defined for this argument. On the other hand, general reified equality constraints impose integrity constraints on the boolean variables. Turning an integrity constraint into a cost function causes rugged landscapes which may be difficult to explore, although CMA-ES is also a pretty good solver in this case. Since our benchmarks do not use reified constraints nor discrete variables, they are currently out of the scope of our MiniZinc backend.

2.2 Cost aggregation

In our backend, the gathering of the cost functions can be tuned by using annotations. First, each constraint can be annotated with a weight which will affect the cost in the aggregation. By default, the weight of a constraint is 1.

```
annotation weight(float);
```

Second, every MiniZinc model contains one and only one *solve* instruction, which gives the objective. The *solve* item can be annotated with one of the following annotations which change the definition of the violation cost of the whole constraints. By default, `weighted_sum` is assumed.

```
annotation weighted_sum;
```

$$\text{violation\_cost} = \sum_c \text{weight}(c) \cdot \text{cost}(c)$$

```
annotation fuzzy;
```

$$\text{violation\_cost} = \max_c \text{weight}(c) \cdot \text{cost}(c)$$

```
annotation probabilistic;
```

$$\text{violation\_cost} = 1 - \prod_c \left(\frac{1}{1+\text{cost}(c)}\right)^{\text{weight}(c)}$$

Third, if the FlatZinc instance is a constraint satisfaction problem, the fitness function is defined to be equal to the violation_cost.

```
solve satisfy;
```

If the FlatZinc instance requires to minimize an expression e, the fitness function is defined to be equal to $\alpha \cdot (1 + \text{violation\_cost}) + e$, where α is a coefficient large enough to dominate e. Then α can be set with the following annotation applied to the solve item.

```
annotation alpha(float);
```

By default, $\alpha = 10^{10}$.

3 Stochastic Continuous Optimization with CMA-ES

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES[8]) [6] is one of the most powerful global optimization strategy for minimizing an objective function over the reals in a "black-box" scenario, i.e. without assuming any property about the objective function. This method is a multi-point method which uses a population of configurations (here valuations of the FlatZinc search variables, e.g. packings defined by the coordinates and orientations of the objects) to sample the search space, estimates the covariance matrix at each sampling, determines the next move in the most promising direction (e.g. translations and rotations of objects), and updates accordingly the *multi-variate normal distribution* for the next sampling (i.e. mean value and covariance of the variables).

CMA-ES behaves in effect like a second-order method where the landscape is estimated by sampling, according to some multi-variate normal distribution of the variables, which is itself updated during search in the most promising direction to adapt to the landscape, using an estimation of the second-order moment, the covariance matrix. When the objective function does not improve, CMA-ES can be restarted to find different local optima. We refer to [6] for more details on that stochastic optimization algorithm.

One advantage of CMA-ES is that it requires very little parameter tuning. All our benchmarks have been performed with the C implementation of CMA-ES using the same parameter set: a population size of 100, an initial standard deviation of 20 and a stopping criterion based on a difference less than 10^{-3} for the fitness function.

CMA-ES thus tries to minimize an arbitrary function $f : \mathbf{R}^n \to \mathbf{R}$, where n is the *dimension* of the search space (*i.e.* the number of FlatZinc search variables). The result is a vector $\boldsymbol{x} \in \mathbf{R}^n$ such that $f(\boldsymbol{x})$ is the smallest value encountered so far. The C implementation of CMA-ES expects that the function f has the following interface: `double f(double x[])`. The FlatZinc-to-CMA-ES back-end derives such a fitness function from the FlatZinc model according to the transformations described in Section 2.

4 MiniZinc Models of Geometrical Placement Problems

The problems addressed in this section are taken from the industry of cosmetics packaging. They consist of packing products with various shapes. In this application the objective is to pack a given quantity of a product in a minimum number of bins. The study of different forms in the industry of cosmetics packaging show that convex approximations of objects give poor results but that the objects can

[8] `https://www.lri.fr/~hansen/cmaesintro.html`

be modeled using a combination of Bézier curves. Existing Constraint Programming tools are limited and do not offer capabilities to solve such problems. We show here how to model and solve such problems in MiniZinc.

4.1 Overlap Measures Between Objects

The overlap measure should guide the optimization procedure towards a geometrical placement without overlap, i.e., ideally, the cost should decrease as long as the placement gets closer towards a placement without overlap. ϕ-functions [4] have been introduced for the same purpose of continuous optimization for geometrical placement problems, using decompositions in half-planes, triangles and circles. In this section, we describe three overlap measures for polygones and complex shapes delimited by Bézier curves using the intersection area, the penetration depth, or the sum of the pairwise distances between the intersection points of the borders.

Intersection area. Object intersection area can be used as an overlap measure, which could seem quite natural. However, this area can be costly to compute and does not guide the optimization well. For example, when an object fully contains another one, the overlap measure remains in a plateau for every possible position where the contained object stays inside the container, giving no direction for getting outside the overlap zone (figure 1). In pratice, we do not use this measure.

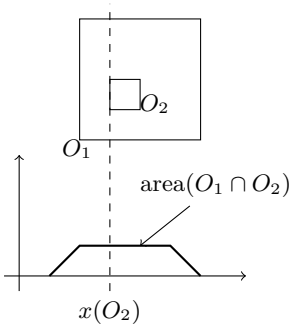

Fig. 1. Intersection area between two rectangles O_1 and O_2 in function of $x(O_2)$, with a plateau when O_2 is included in O_1.

Penetration depth. The penetration depth is a common measure in computer-aided design [2]: the penetration depth between two objects is the smallest norm such that there exists a translation vector to apply to one of the two objects such to lead to a placement where the two objects do not intersect.

The penetration depth between two circles (C_1, r_1) and (C_2, r_2) is trivial to compute:

$$\mathrm{pd}(C_1, r_1); (C_2, r_2)) = \max(0, r_1 + r_2 - C_1C_2)$$

that is to say the difference between the sum of their radius and the distance between their centers (figure 2). This distance was already used for circles in [9,4].

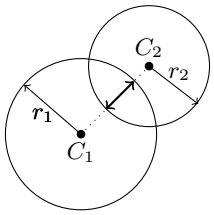

Fig. 2. Penetration depth between two circles (C_1, r_1) and (C_2, r_2)

More generally, the penetration depth between two objects O_1 and O_2 is equal to the distance between the origin $(0, 0)$ and the complementary of the Minkowski difference $O_1 \ominus O_2 = \{p_1 - p_2 \mid p_1 \in O_1, p_2 \in O_2\}$.

$$\mathrm{pd}(O_1; O_2) = \min\{\|\boldsymbol{u}\| \mid \boldsymbol{u} \notin O_1 \ominus O_2\}$$

Indeed, this distance is by definition the smallest norm such that there exists a vector $\boldsymbol{u}$ such that $\boldsymbol{u} \notin O_1 \ominus O_2$, that is to say a vector such that for all $p_1 \in O_1, p_2 \in O_2$, $\boldsymbol{u} \neq p_1 - p_2$, thus we have $O_1 \cap (O_2 + \boldsymbol{u}) = \emptyset$.

The Minkowski difference of two polygons is a polygon, computable in a time quadratic to the number of edges, and the Minkowski difference of two convex polygons is a convex polygon, computable in a time linear to the number of edges [5] (figure 3).

Note that the penetration depth only consider translations, whereas the search space we consider for optimization may include rotation angles as additional "dimensions". [12] extends the Minkowski difference to consider object rotations as well. This extension is difficult to interpret geometrically and makes overlap measures depend on the choice of the origin for each object. We will restrict ourselves to Minkowski difference in the Euclidean space.

Sum of the pairwise distances between the intersection points of the borders. For the overlap measure between two objects O_1 and O_2 that have non polygonal shapes like those delimited by Bézier curves, or for heterogeneous shapes (for example, when mixing Bézier curves and polygons), we prefer to use another measure simpler to compute: the sum of the pairwise distances between the intersection points of the borders ∂O_1 and ∂O_2. We suppose that $\partial(O_1) \cap \partial(O_2)$

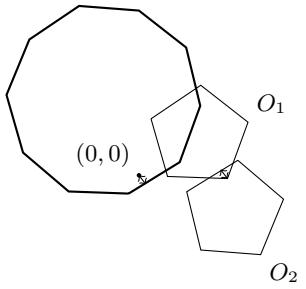

Fig. 3. Minkowski difference between two pentagons O_1 and O_2. The penetration depth between O_1 and O_2 is the distance between the origin and the border of the Minkowski difference.

is a finite set, applying infinitesimal offsets if necessary.

$$\mathrm{spd}(O_1; O_2) = \sum_{\substack{p_i, p_j \in \partial(O_1) \cap \partial(O_2) \\ i<j}} \|\overrightarrow{p_1 p_2}\|$$

There exist several methods to compute the intersections between two Bézier curves numerically [13] (figure 4). We use a dichotomic search by using de Casteljau's algorithm for splitting the curves. The dichotomic search can also be used to compute numerically the intersections between Bézier curves and circles.

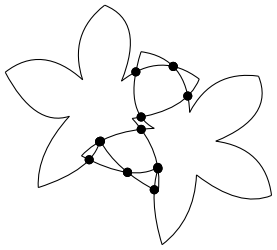

Fig. 4. Intersection points of the borders of two objects delimited by Bézier curves.

The intersections between a Bézier curve and a segment can be computed algebraically. Indeed, by changing the frame, we can suppose without loss of generality that the segment lays on the abscissa axis. The Bézier curve (p_0, p_1, p_2) intersects the axis for every parameter t, $0 \leq t \leq 1$, such that $(1-t)((1-t)y_{p_0} + typ_1) + t((1-t)y_{p_1} + typ_2) = 0$: this is a second-order polynomial in t. For each solution t_0, it suffices to check that the abscissa $(1-t_0)((1-t_0)x_{p_0} + t_0 x p_1) + t_0((1-t_0)x_{p_1} + t_0 x p_2)$ belongs to the segment.

This measure does not fulfill the requirements of an ideal overlap measure: the measure is null when one object is included in the other and is not monotonic with respect to the penetration depth. However, it is locally monotonic in a neighborhood around overlap-free placements: in the context of a local search,

by choosing an overlap-free initial placement (spreading the objects enough far ones from the others), this measure experimentally appears to be sufficient to preserve the overlap-freeness during the placement compaction process.

4.2 Continuous Packing Model

This section makes use of the overlap measures introduced above to express MiniZinc models for continuous packing of circles, arbitrary polygons with rotation and other complex shapes like rosettes delimited by Bézier curves. The penetration depth is used as measure of overlap between two circles and between two polygons, while the sum of distances between the intersection points is used for every other pair of shapes.

Circles. The following predicate expresses the constraint that the two circles $((x_1, y_1), r_1)$ and $((x_2, y_2), r_2)$ do not overlap.

```
predicate non_overlap_circles(
  var float: x1, var float: y1, var float: r1,
  var float: x2, var float: y2, var float: r2) =
  pow(x1 - x2, 2) + pow(y1 - y2, 2) > pow(r1 + r2, 2);
```

It is worth noticing that the inequality $(x_1 - x_2)^2 + (y_1 - y_2)^2 > (r_1 + r_2)^2$ is compiled into the cost function $(r_1+r_2)^2-(x_1-x_2)^2-(y_1-y_2)^2$, which is monotonic with respect to the penetration depth $(r_1+r_2)-\sqrt{(x_1 - x_2)^2 - (y_1 - y_2)^2}$, introduced in the previous section.

We consider a benchmark of circle placement problems [3] where there are n circles to place in a circular bin. The usual modelling [9] supposes that the circular bin is centered on the origin. The following function compute for each circle the minimum radius for the circular bin to contain the circle $(x, y), r)$.

```
function var float: bounding_circle_radius(
  var float: x, var float: y, var float: r) =
  sqrt(pow(x, 2) + pow(y, 2)) + r;
```

The search variables are the positions of the circle centers.

```
int: n;
array[1 .. n] of var float: x;
array[1 .. n] of var float: y;
```

The circle positions are constrained to be non-overlapping.

```
constraint forall(i in 1..n,j in i+1..n)(
  non_overlap_circles(x[i], y[i], radius(i), x[j], y[j], radius(j)));
```

The goal is to minimize the radius of the circular bin. Intermediary variables are introduced to store the minimal bounding radius for each circle to circumvent a limitation of the `max` function for arrays in MiniZinc that needs to know the bounds of the arguments.

```
array[1 .. n] of var 0.0 .. 1000.0: bounding_radii;

constraint forall(i in 1 .. n)(
  bounding_radii[i] = bounding_circle_radius(x[i], y[i], radius(i)));
```

```
constraint bounding_radius = max(bounding_radii);

solve minimize bounding_radius;
```

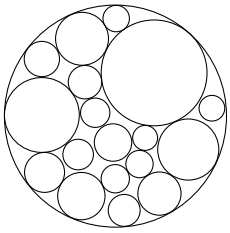

Fig. 5. Example of packing found by MiniZinc-CMAES for 18 circles of radii $i^{-1/2}$ for $1 \leq i \leq 18$ (circle packing benchmark of [3]).

It is worth noticing that applying min and max functions to overlap measures allows Boolean combinations of geometrical shapes to be expressed. For instance, the figure 6 shows a placement for 20 geometrical rosettes, where each rosette $\mathcal{R}((x, y), r)$ is defined as the union of six intersections between pairs of circles:

$$\mathcal{R}((x,y),r) = \bigcup_{i=1}^{6} \mathcal{C}((x + \cos(2 \cdot i \cdot \frac{\pi}{6}) \cdot r, y + \sin(2 \cdot i \cdot \frac{\pi}{6}) \cdot r), r)$$
$$\cap\, \mathcal{C}((x + \cos(2 \cdot (i+2) \cdot \frac{\pi}{6}) \cdot r, y + \sin(2 \cdot (i+2) \cdot \frac{\pi}{6}) \cdot r), r)$$

Objects with rotations. The placement of each object in the subsequent examples is described by three search variables: the position on the x axis, the position on the y axis, and the rotation angle r.

```
set of int: position = 1 .. 3;
int: x = 1;
int: y = 2;
int: r = 3;
```

An object is described by a variable of type `array[position] of var float`, that is to say an array of three variables. The constants `x`, `y` and `r` are used as projectors: given an array `object`, components can be accessed as `object[x]`, `object[y]` and `object[r]`. (Note that MiniZinc has not yet support for records.)

Points are stored in an array of two coordinates. The points that describe the shapes of an object are expressed in a frame relative to the given object position and orientation. The function `image_of_point` defined below transforms the coordinates of a point to the global frame.

```
set of int: coordinates = 1 .. 2;
function array[coordinates] of var float: image_of_point(
```

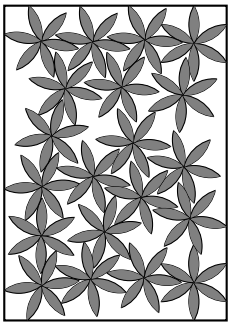

Fig. 6. Placement found by MiniZinc-CMAES for 20 geometrical rosettes, defined as unions of circle intersections.

```
  array[position] of var float: object,
  array[coordinates] of float: point
) = [
  cos(object[r]) * point[x] - sin(object[r]) * point[y] + object[x],
  sin(object[r]) * point[x] + cos(object[r]) * point[y] + object[y]
];
```

Object positions are stored in a matrix.

```
set of int: objects = 1 .. n;
array [objects, position] of var float: object_positions;
```

The following function returns the position of an object given its index.

```
function array[position] of var float: object_position(int: object) =
  [object_positions[object, d] | d in position];
```

Polygons. We consider pentagons with the following vertex coordinates (relative to the object frame).

```
array[1..5, coordinates] of float: pentagon =
  [| 2.2024586, 58.90577
   | 18.54966,8.594238
   | 71.45033,8.594238
   | 87.79755,58.90576
   | 45.0,90.0 |];
```

These pentagons approximate the Bézier rosettes that we consider below: the vertices join the ends of the petals (figure 7). It is not exactly the convex hull since one petal goes outside the pentagon but it is close to (and the convex hull of the rosette is not polyhedric). However, this approximation is sufficient to observe the gain obtained in the placements by considering the precise Bézier rosettes instead of such approximations.

We suppose that the following function computes the penetration depth between two (convex) polygons.

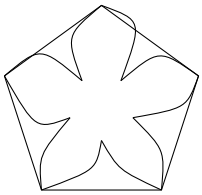

Fig. 7. The pentagon obtained by considering each petal's end as a vertex is not a correct approximation of a Bézier rosette.

```
function var float: penetration_depth_between_polygons(
  array[int, coordinates] of var float: vertices0,
  array[int, coordinates] of var float: vertices1
);
```

The Minkowski difference between two polygons can be expressed with arithmetic constraints through reification: the constraint is cumbersome to write directly, but can be automatically generated, for example by a ClpZinc model [8]. Alternatively, the function can be implemented in the back-end as an auxiliary C function, which is the case of our current implementation.

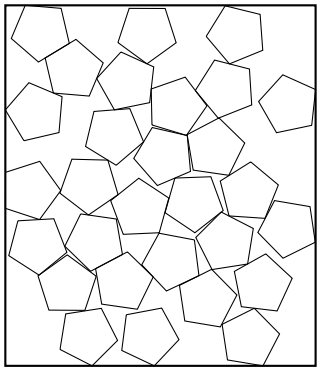

Fig. 8. Placement found by MiniZinc-CMAES for 29 pentagons

Bézier curves. The rosettes that we consider are delimited by the 10 following quadratic Bézier curves (one curve by line, each curve is described by three control points).

```
set of int: curves = 1 .. 10;
set of int: quadratic_bezier_control_points = 1 .. 3;
array [
  1 .. card(curves) * card(quadratic_bezier_control_points),
```

```
  coordinates] of float: curve_points =
  [| 2.2024586, 58.90577 | 16.01051, 34.989525 | 31.150425, 40.5
   | 31.150425, 40.5 | 17.211597, 23.888353 | 18.54966, 8.594238
   | 18.54966, 8.594238 | 42.69821, 17.38359 | 45.0, 30.437695
   | 45.0, 30.437695 | 52.512943, 17.424889 | 71.45033, 8.594238
   | 71.45033, 8.594238 | 73.003426, 26.34613 | 58.84958, 40.5
   | 58.84958, 40.5 | 82.6242, 44.69211 | 87.79755, 58.90576
   | 87.79755, 58.90576 | 70.514114, 71.00775 | 53.55951, 56.78115
   | 53.55951, 56.78115 | 63.23457, 83.36317 | 45.0, 90.0
   | 45.0, 90.0 | 29.181046, 76.72632 | 36.44049, 56.781155
   | 36.44049, 56.781155 | 18.055468, 72.20802 | 2.2024586, 58.90577 |];
```

We suppose that the following function computes the sum of the distances between the intersections of two sets of curves. This function is implemented in the back-end as an auxiliary C function.

```
function var float: sum_of_distances_between_bezier_intersection_points(
  array[int, coordinates] of var float: curves0,
  array[int, coordinates] of var float: curves1
);
```

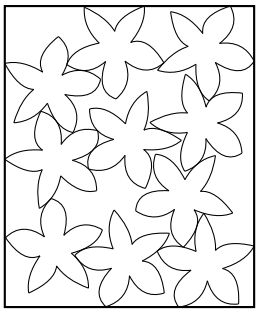

Fig. 9. Placement found by MiniZinc-CMAES for 10 Bézier rosettes

Mixing Bézier curves and rectangles. For computing the overlaps between Bézier curves and rectangles, we suppose that the following function computes the sum of the distances between the intersections of a Bézier curve and a rectangle. This function is implemented in the back-end (as an auxiliary C function), but the arithmetic could be expressed in MiniZinc as well.

```
function var float:
    sum_of_distances_between_bezier_and_polygon_intersection_points(
  array[int, coordinates] of var float: curves,
  array[int, coordinates] of var float: vertices
);
```

The figure 10 shows a placement found for 16 Bézier rosettes and 16 rectangles. It is worth noticing that even if the optimization procedure has found a non-trivial placement, for instance for the rectangles and the rosettes in the bottom right of the figure, some visually obvious improvements of the placement of the left rosettes are not found in this run of CMA-ES.

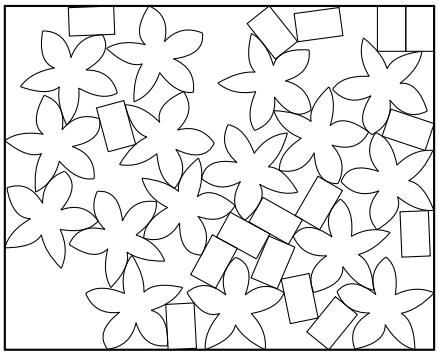

Fig. 10. Placement found by MiniZinc-CMAES for 16 Bézier rosettes and 16 rectangles, in this run of CMA-ES which stays stick in a local minimum.

5 Evaluation Results of MiniZinc-CMAES

The following tables 1 and 2 summarize the performance obtained with MiniZinc-CMAES, in terms of computation time, smallest area found, mean area and variance of the area among 50 runs of CMA-ES. For every example, results are averaged over 50 runs. All these results have been obtained with the default parameters of CMA-ES described in Section 3. It is worth noticing that smaller initial standard deviations tend to generate solutions with overlaps that the optimization fails to remove, and bigger standard deviations augment convergence times. Total time is the sum of the computation times for all the 50 restarts: for each problem, all the restarts have been computed in parallel on a cluster, one problem per core.

Roses	Total time (50 restarts)	Mean time	Smallest area found	Mean area	Variance (area)
10	17 min 32 s	21 s	25.266	27.303	1.893
11	31 min 13 s	37 s	27.369	29.692	1.585
12	41 min 18 s	49 s	28.761	32.511	3.675
13	1 h 1 min 39 s	1 min 13 s	32.936	34.987	2.648
14	1 h 17 min 37 s	1 min 33 s	33.816	37.714	2.926
15	1 h 43 min 19 s	2 min 3 s	37.233	41.085	3.57
16	2 h 6 min 57 s	2 min 32 s	39.729	43.86	5.891
17	2 h 26 min 23 s	2 min 55 s	41.883	46.113	5.947
18	3 h 11 min 20 s	3 min 49 s	43.582	49.123	13.828
19	3 h 47 min 24 s	4 min 32 s	46.74	52.594	10.769
20	5 h 8 min 42 s	6 min 10 s	49.006	54.89	8.579

Table 1. Computation time for placement of geometrical rosettes.

Shapes	Total time (50 restarts)	Mean time	Best area	Mean area	Area variance
10 + 10	7 j 17 h 35 min 10 s	3 h 42 min 42 s	66 715.294	70 080.254	$3.784 \cdot 10^5$
11 + 11	12 j 5 h 34 min 30 s	5 h 52 min 18 s	72 846.432	78 495.343	$4.434 \cdot 10^5$
12 + 12	41 min 18 s	49 s	87 257.346	90 238.345	$5.499 \cdot 10^5$
13 + 13	13 j 23 h 13 min 39 s	6 h 42 min 16 s	85 492.98	$1.024 \cdot 10^5$	$6.985 \cdot 10^5$
14 + 14	17 j 9 h 27 min 23 s	8 h 20 min 56 s	$1.11 \cdot 10^5$	$1.398 \cdot 10^5$	$8.219 \cdot 10^5$
15 + 15	23 j 9 h 18 min 63 s	11 h 13 min 35 s	$1.078 \cdot 10^5$	$1.584 \cdot 10^5$	$1.281 \cdot 10^6$

Table 2. Computation time for placement of mixed shapes: Bézier's rosettes and rectangles.

It is worth noticing that Eclipse and G12 with their interval constraint solvers can parse the MiniZinc models of the previous section that do not use predicates defined as auxiliary C functions in the back-end. However the performances are very poor with results obtained only for 3 circles.

6 Conclusion

We have presented here a stochastic continuous optimization backend for MiniZinc models over real numbers. We have shown the benefits of this approach using the CMA-ES solver for continuous optimization on a series of geometrical placement problems motivated by industrial applications in logistics, involving mixed square-curve shapes, and also complex shapes defined by Bézier curves. Probably because of the novelty of these problems for complex shapes, we have not identified benchmarks for comparing the techniques presented here, but in [9] we showed that the solutions found with CMA-ES on circle packing were at just 2% of the best solutions found with dedicated solvers.

The declarative modeling in MiniZinc combined to the solving using the transformation to CMA-ES described in this paper, does not come with any significant overhead and provides fully declarative solutions to very hard geometrical placement problems. The non-overlap constraint has a cost function based on the penetration depths between objects, using Minkowski sums for polygons, and a simpler measure of overlap for Bézier curves. A classical difficulty in the definition of the error function of a conjunction of constraints is the normalization of the error function for each constraint. This has been solved here by letting the modeller specify in MiniZinc the cost function if different from the default cost aggregation function (i.e. the sum of the costs).

The recourse to such a black-box optimization procedure for FlatZinc makes sense especially in presence of non-linear constraints, and in absence of integer variables, but the transformation we have given of a FlatZinc model in a non-negative real-valued cost function is quite general. We have focused on continuous placement problems, but our MiniZinc/CMA-ES can be applied in principle to any constraint model over real numbers. The examples taken here from industrial problems in logistics, including objects defined by Bézier curves,

should contribute to open a new domain of application of constraint methods in computational geometry, at the intersection of optimization and computer-aided design.

Acknowledgements. This work has been funded by the ANR Blanc Simi2 Net-WMS-2 grant ANR-11-BS02-0005. We would like to thank all the partners of this project for fruitful discussions.

References

1. G. Björdal, J.-N. Monette, P. Flener, and J. Pearson. A constraint-based local search backend for minizinc. *Constraints*, 20(3):325–345, 2015.
2. S. Cameron and R. Culley. Determining the minimum translational distance between two convex polyhedra. In *Robotics and Automation. Proceedings. 1986 IEEE International Conference on*, volume 3, pages 591–596, Apr 1986.
3. I. Castillo, F. J. Kampas, and J. D. Pintér. Solving circle packing problems by global optimization: Numerical results and industrial applications. *European Journal of Operational Research*, 191(3):786–802, 2008.
4. N. Chernov, Y. Stoyan, and T. Romanova. Mathematical model and efficient algorithms for object packing problems. *Computational Geometry*, 43:535–553, 2010.
5. D. Dobkin, J. Hershberger, D. Kirkpatrick, and S. Suri. Computing the intersection-depth of polyhedra. *Algorithmica*, 9:518–533, 1993.
6. N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.
7. P. V. Hentenryck and L. Michel. Synthesis of constraint-based local search algorithms from high-level models. In *Proc. AAAI*, pages 273–278, 2007.
8. T. Martinez and F. Fages. On translating minizinc constraint models into fitness function for evolutionary algorithms: Application to continuous placement problems. In *Proceedings of the sixth Workshop on Bin Packing and Placement Constraints BPPC'15, associated to CP'15*, Sept. 2015.
9. T. Martinez, L. Vitorino, F. Fages, and A. Aggoun. On solving mixed shapes packing problems by continuous optimization with the cma evolution strategy. In *Proceedings of the first Computational Intelligence BRICS Congress BRICS-CCI'13*, pages 515–521. IEEE Press, Sept. 2013.
10. L. Michel and P. V. Hentenryck. The comet programming language and system. In *Proc. Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 881–881, 2005.
11. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, pages 529–543, 2007.
12. I. Salas and G. Chabert. Packing curved objects. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI'15*, Buenos Aires, Argentina, 2015.
13. T. W. Sederberg. Chapter 7, planar curve intersection. Technical report, Computer Aided Geometric Design Course Notes, 2011.