2D object reconstruction with ASP

Alessandro Dal Palù[1], Agostino Dovier[2], and Andrea Formisano[3]

[1] Dipartimento di Scienze Matematiche, Fisiche e Informatiche, Università di Parma
[2] Dipartimento di Scienze Matematiche, Informatiche e Fisiche, Università di Udine
[3] Dipartimento di Matematica e Informatica, Università di Perugia

Abstract. Damages to cultural heritage due to human malicious actions or to natural disasters (e.g., earthquakes, tornadoes) are nowadays more and more frequent. Huge work is needed by professional restores to reproduce, as best as possible, the original artwork or architecture opera starting from the potsherds. The tool we are presenting in this paper is devised for being a digital support for this kind of work. As soon as the fragments of the opera are cataloged, a user (possibly young students, and even children, using a tablet or a smartphone as playing with a video game) can propose a partial reconstruction. The final part of the job is left to an ASP program that first computes a pre-processing task to find coherence between (sides of) fragments, and then tries to reconstruct the original object. Experiments are made here focusing on 2D reconstruction (frescoes, reliefs, etc).

Keywords: Answer set programming, 2D object reconstruction, geometric reasoning.

1 Introduction

This paper describes a declarative approach to the automated solution of object reconstruction problems. In general terms, the goal consists in recomposing an artifact by assembling the collection of its fragments. Automated tools supporting this task have wide application, because variants of such a general problem arise in many fields. Common, relatively simple, forms of the problem can be found in cardboard puzzles, such as edge-matching puzzles, packing puzzles, or jigsaw puzzles [15, 14]. Plainly, these puzzles abstract and expose in a simpler setting the core difficulties one encounters in restoring a fresco, a wall painting, or an ancient palace facade that have been damaged, for instance, as consequence of an earthquake [28, 22]. Similar tasks are the assembling of shredded documents and torn photos usually associated with forensic investigations or historical and philological studies [12, 16, 33, 25]. The hard problem of recomposing ancient broken artifacts, such as damaged potteries, terracottas, or sculptures, frequently emerges in archaeology [1, 22, 29].

The object reconstruction problem shares significant aspects with other geometrical optimization problems, such as cutting, bin-packing, and nesting problems, that are extremely relevant in industrial contexts (see [2, 18, 4], among many). Vast literature exists on modeling and solving any of these problems and the proposed techniques and methods are often applicable to the others thanks to the strict relationship existing among them.

Among the various formalizations, we mention here the so called *pixel/raster method*, the *nofit polygon method*, and the *Phi-function method* (see [2] and the references therein for a detailed description). On the one hand, many alternative approaches have been proposed to solve some variants of the general problem, by using different tools and techniques such as dynamic programming, genetic algorithms, linear programming, greedy algorithms, integer programming, particle swarm optimization, (convex) optimization methods, etc. cf. [19, 28, 32, 23, 18, 27, 26]. On the other hand, while focusing on the 2D object reconstruction problem, various approaches appeared in the literature put emphasis on different aspects/features. A first classification can be done by distinguishing between *apictorial* and *pictorial* techniques. In the former case, the solving methods are shape-based, namely they only consider the shape of the fragments to be assembled. Often some restriction on the admissible shapes is imposed, e.g., convexity and/or homogeneity of shapes, some degree of smoothness/regularity of fragments' edges, etc. (cf. [15, 11, 10]), Conversely, other techniques rely on the availability of chromatic and pictorial information (image color, texture, features, orientation, etc.) to achieve better results [25, 12, 26, 16, 33]. Other options are also sometimes considered in the literature, such as dealing with extraneous fragments, missing fragments, and eroded edges. Consequently, incompleteness and tolerances in edge-matching have to enter into play, increasing the complexity of the model/solution.

All mentioned approaches mainly exploit numerical algorithm and analytic techniques. None of them focuses on declaratively modeling and solving object reconstruction problems. To the best of our knowledge, few attempts have been pursued in exploiting logic programming or constraint programming to automate this kind of spatial/geometrical reasoning. Prolog has been used in [31] to deal with cartographic map overlay, whereas [3, 24] exploit Constraint Logic Programming (CLP) to model nesting problems of non-convex polygons. A framework based on Answer Set Programming (ASP) modulo theories supporting generic spatial reasoning is described in [30]. No proposal has been advanced to deal with object reconstruction problems in the specific.

In what follows we will make the initial steps towards an ASP-based framework for object reconstruction. We will restrict the treatment to the 2D case, but the very same ideas can be generalized to the 3D case.

In contrast to the proposals mentioned earlier, a purely declarative approach enables a higher-level abstraction, focusing on modeling (instead on specifically tailored algorithms), greater elaboration tolerance, and incremental modeling. The use of ASP-based non-monotonic reasoning admits incompleteness and uncertainty in modeling. This allows one to design a framework that can be incrementally completed so as to encompass an increasing number of features, such as dealing with missing fragments, damaged borders, erosion, partial pictorial information, errors, approximations, imprecise measures in fragments' extracted features, etc.

To start within a simplified setting, we assume that fragments are represented as polygons (not necessarily convex). Moreover, each edge of a fragment may be characterized by some features (pictorial info, texture, ...) W.l.o.g., we can represent all these features by associating a single *color* to the edge. For the time being we do not consider missing/extraneous fragments and eroded edges.

The paper is organized as follows. In Section 2 we briefly survey syntax and semantics of Answer Set Programming. In Section 3 we define the desired input for the instances of the problem as an ASP program for preprocessing of the input data. The preprocessed data is given as input to the main search core, described in Section 4. Some initial experimental results are presented in Section 5. Finally, a brief discussion on current and future work, and some conclusions are drawn in Section 6.

2 Preliminaries — ASP

Answer Set Programming (ASP) is a dialect of logic programming developed for Knowledge Representation and Reasoning, that allows a free use of negation as failure in clause bodies. Disjunctive heads are also allowed in ASP, but we will not use this feature in this paper, and therefore we will present the simplified syntax. The meaning of a ASP program is regulated by stable model semantics. We refer the reader to [17, 20] or to the recent book [13] for all details.

Given a set of propositional symbols (atoms) $\mathcal{P}$, an ASP rule is as follows:

$$p_0 \leftarrow p_1, \cdots, p_m, \mathit{not}\ p_{m+1}, \cdots, \mathit{not}\ p_n \tag{1}$$

where for $i \in \{0, \ldots, n\}$, $p_i \in \mathcal{P}$. A rule is a *fact* if $n = 0$, namely, if it has the form $p_0 \leftarrow$. Given a rule r, the atom p_0 is referred to as the *head* of the rule (and denoted by $\mathit{head}(r)$), while the set of literals $\{p_1, \cdots, p_m, \mathit{not}\ p_{m+1}, \cdots, \mathit{not}\ p_n\}$ is referred to as the *body* of the rule (and denoted by $\mathit{body}(r)$). Customarily, $\mathit{body}^+(r) = \{p_1, \cdots, p_m\}$ and $\mathit{body}^-(r) = \{p_{m+1}, \cdots, p_n\}$. An ASP program is a collection of ASP rules.

ASP syntax allows also first-order atoms of the form $p(t_1, \ldots, t_k)$, where p is a predicate symbol and $t_1, \ldots, t_k$ are variables and/or constants (nested terms are not allowed). Given a program P, the ASP program composed by all the ground rules obtained replacing all variables in a clause with constant symbols in all possible ways is called the *grounding* of P. Most ASP solvers start an execution by *grounding* the program, thus removing all variables.

As usual, models of programs are Herbrand models that can be described by a set of atoms M. An atom is true in a model M if it belongs to M and it is false otherwise. A body is satisfied by a model M if all its positive literals belong to M and no negated literal of the body belongs to M. A rule is satisfied by a model M if whenever its body is satisfied, its head is true in M. M is an answer set of a program P if M is the minimal model of the *reduct program* P^M, which is obtained from P and M as follows:

– Remove from P all rules r such that $M \cap \mathit{body}^{-}(r) \neq \emptyset$;
– Remove all negated atoms from the remaining rules.

P^M is a *definite program*, i.e., a set of rules that does not contain any occurrence of negation and as such it admits a unique minimal (Herbrand) model.

A *constraint* is a rule of the form

$$\leftarrow p_1, \cdots, p_m, \mathit{not}\ p_{m+1}, \cdots, \mathit{not}\ p_n \tag{2}$$

This is a simply a shorthand for an ASP rule of the form $p \leftarrow p, p_1, \cdots, p_m$, $\mathit{not}\ p_{m+1}, \cdots, \mathit{not}\ p_n$, where p does not occur elsewhere in the program. A constraint states explicitly that it is impossible that in a model M all $p_1, \ldots, p_m$ are true and all $p_{m+1}, \ldots, p_n$ are not. Other syntactical extensions are commonly employed, such as choice rules or aggregates. A choice rule is of the form

$$\{p\} \leftarrow \mathit{body} \tag{3}$$

If the *body* is satisfied by a model M then p is justified either to hold or not. An aggregate has the form $\mathit{op}\{t{:}q\}$, where q is an atom (possibly involving variables), t is a term possibly sharing variables with q, and op is an aggregate operator such as $\#\mathit{count}$, $\#\mathit{sum}$, $\#\mathit{max}$, $\#\mathit{min}$. Given an answer set M, $t{:}q$ stands for the collection of all ground instances of t that correspond to satisfied instances of q. Each aggregate is evaluated by applying the operator op to such a collection of terms. We refer the reader to [9] for a description of these extensions.

3 Input and pre-processing

An instance of our problem is a set of 2D pieces that should be assembled together. We focus on *almost-perfect* matches between pairs of sides belonging to different pieces. In particular, we consider only matches where the positions of sides' vertices are closer than a given tolerance.

Let us assume that the input is described by the predicate `poly/4` as in `poly(piece-id, vertex, x, y)`, where `piece-id` ranges among the pieces. Each piece is described by a sequence of vertices (a polygon), numbered by starting from index 0 and proceeding counter-clockwise along the perimeter. Each vertex is assigned a unique pair of coordinates. Moreover, each side of each polygon is numbered by the number assigned to its first vertex (in counter-clockwise order). Additional information can be added (e.g., describing the *color* of the side `color(piece-id, vertex, color)` or stating that the side is "external" in the composed object `external_side(piece-id, side)`). We do not explicitly deal with this piece of information in what follows. It suffices to say that the pre-processing stage (described below) is in charge to process such information in order to evaluate whether two sides are compatible and could be matched to form the assembled object.

The input file contains the definition of some parameters that are used either by the pre-processing stage or by the search stage: `tolerance/1`, `delta/1`, `offset/1`, `rangeX/2`, `rangeY/2`, and `borders/1`.

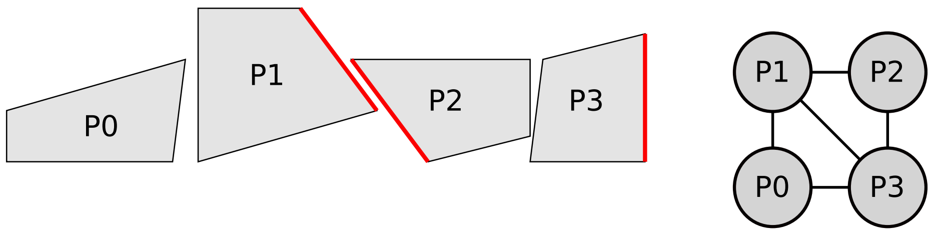

Fig. 1. Representation of pieces of Example 1 and the graph of their possible side-side matchings. Every segment length occurs exactly twice among the pieces, except for those colored in red that occur three times.

Example 1. Let us consider a four-pieces example. Pieces 0, 1, 2, 3 are represented by the following facts (see also Figure 1—left):

`poly(0,0,0,0).`
`poly(0,1,13,0).`
`poly(0,2,14,8).`
`poly(0,3,0,4).`
	`poly(2,0,33,0).`
	`poly(2,1,41,2).`
	`poly(2,2,41,8).`
	`poly(2,3,27,8).`

`poly(1,0,15,0).`
`poly(1,1,29,4).`
`poly(1,2,23,12).`
`poly(1,3,15,12).`
	`poly(3,0,41,0).`
	`poly(3,1,50,0).`
	`poly(3,2,50,10).`
	`poly(3,3,42,8).`

A first simple input pre-processing (a translation) is executed and results are represented through the predicate `poly_cooR` so as to set vertex 0 of each piece in position $(0, 0)$. For instance, piece number 3 has the normalized coordinates:

```
poly_cooR(3,0,0,0).           poly_cooR(3,1,9,0).
poly_cooR(3,2,9,10).          poly_cooR(3,3,1,8).
```

Rotations are allowed in order to find sides' matches, modulo an initial discretization choice (specified by the predicate `delta`, in our example the discretization step is set to 10). Admissible rotations are given by predicate `degrees`:

```
degrees(0).
degrees(X + Y) :- delta(Y), degrees(X), X + Y  < 360.
```

For each pair of pieces (p_1, p_2) and for each pair of sides $(i, i + 1), (j + 1, j)$, where $(i, i+1)$ is a side of p_1 and $(j+1, j)$ is one of p_2, we compute whether they are compatible and what is the relative rotation needed to match them. This is done by the predicate `good_match`, described below. The auxiliary predicate `numVertices/2` counts the number of vertices of a piece.[4]

[4] Notice that \ denotes the *modulo* operation in the ASP syntax of the grounder GRINGO 4. Here it is used to compute the successor of a vertex $(0 \mapsto 1, 1 \mapsto 2, \ldots, n \mapsto 0)$ in the order described earlier.

```
good_match(P1,I,P2,J,Angle,L1,L2):-
    poly_cooR(P1,I,I1x,I1y),  poly_cooR(P2,J,J1x,J1y),
    P1 != P2,   numVertices(P1,N1),    numVertices(P2,N2),
    poly_cooR(P1, (I+1) \ N1, I2x, I2y),
    poly_cooR(P2, (J+1) \ N2, J2x, J2y),
    ... % continued below
```

First, the relative positions of sides i and j when vertex 0 of the two pieces is put in $(0,0)$ are computed. Then, each pair of sides is analyzed to assess if they are compatible and, in this case, the required rotation is computed. Compatibility means that they have roughly the same size, namely their length is the same modulo a `tolerance`, specified by a predicate set to 16 in our example. Since ASP only deals with integers, in order to gain enough precision in computing ratios and trigonometric values, we scale all numbers by 2^{10} (effective tolerance is therefore $\frac{16}{1024} = \frac{1}{64}$ of unit). Two auxiliary predicates storing the (discrete) tables for sine and cosine are used.

```
    V1x = I2x-I1x,    V1y = I2y-I1y,
    V2x = -J2x+J1x,   V2y = -J2y+J1y,
    degrees(Angle),  cosTable(Angle,Cos),  sinTable(Angle,Sin),
    %% position of the points after rotation
    V2Rx = V2x * Cos - V2y * Sin,
    V2Ry = V2y * Cos + V2x * Sin,
    V1Rx = 1024 * V1x,
    V1Ry = 1024 * V1y,
    %% Check of the sizes
    tolerance(T1),
    V1Rx < V2Rx + T1,  V1Rx > V2Rx - T1,
    V1Ry < V2Ry + T1,  V1Ry > V2Ry - T1.
```

Going back to Example 1, with a tolerance set to 20 we obtain

```
good_match(0,1,3,3,0).      good_match(3,3,0,1,0).
good_match(1,0,0,2,0).      good_match(0,2,1,0,0).
good_match(2,3,1,1,0).      good_match(1,1,2,3,0).
good_match(2,3,3,1,37).     good_match(3,1,2,3,323).
good_match(3,2,2,0,0).      good_match(2,0,3,2,0).
```

Observe, for instance, that side 1 of piece 3 can match with side 3 of piece 2 with an angle of 323 degrees (see also Figure 1—right). Of course the predicate is symmetric (in the first argument) while angles are explementary to each other.

As last part of pre-processing, the predicate `check_ccw(P1,I,Sign)` computes the vector product of each three consecutive vertices and stores it in `Sign`. A constraint enforces that each computed value must be positive (i.e. the vertices are arranged along the counter-clockwise enumeration).

```
check_ccw(P1,I,Sign):-
    poly_cooR(P1, I, I1x, I1y),
    numVertices(P1,N1),
    I1 = (I+1) \ N1, I2 = (I+2) \ N1,
    poly_cooR(P1,I1,I2x,I2y),
    poly_cooR(P1,I2,I3x,I3y),
    %% Vectors are computed and used to compute the sign
    V1x=I2x-I1x,   V1y=I2y-I1y,
    V2x=I3x-I2x,   V2y=I3y-I2y,
    Sign = V1x*V2y - V1y*V2x.
%%% The sign cannot be negative
:-  check_ccw(_,_,S), S<0.
```

4 Main encoding

Given a problem instance, the outcome of the pre-processing consists of the extensional definitions of the predicates `poly_cooR`, `good_match`, the auxiliary predicates `numVertices/2`, `polyId/1`, the extensional definition of `degrees/1`, and the parameters defined by the user. In particular `offset/1` is used to define the predicate `range(-Of..Of) :- offset(Of)`. The trigonometric table is also included.

Given the set of possible matchings (predicate `good_match`) we would like to select those that lead to a coherent shape of the composed object. The overall idea is to place the first piece (i.e., first vertex in $(0, 0)$ keeping the same rotation as in the pre-processed input) and then placing the others by selecting matchings. Given our hypothesis, selecting a match imposes a unique roto-translation to be applied to the next piece and therefore admissible matches eventually lead to a complete placement of all polygons.

The process is driven by a tree construction, where piece 0 is at the root. Selected matchings (predicate `match`) are considered as "directional", thus leading from the root to leaves. Each match defines a matching relation between an already placed piece and the one added next, it also specifies which vertices are paired. Each match is selected from the `good_match` candidates. Each piece should appear in the tree, namely it should be reachable from the root following `match` edges. Additional constraints are then added to filter out results (e.g., to delimit a global bounding box and to impose some "non-overlapping" constraints).

Each possible match listed by predicate `good_match` can be non-deterministically elected to a match or not. Only one direction of the matching is retained and every side of a piece can be matched at most once:

```
{ match(P1,I,P2,J) } :-
       polyId(P1), polyId(P2), P1 != P2,
       good_match(P1,I,P2,J,A).
:- match(P2,A,P1,B), match(P1,B,P2,A).
:- match(P1,I,P2,J2), match(P1,I,P3,J3), P2!=P3.
:- match(P1,I1,P2,J), match(P3,I2,P2,J), P1!=P3.
```

In order to reduce the size of the produced grounding, because of the combinatorial combination of all admitted rotations and translations of pieces, we decouple the rotations from translations during the first part of the solving/reasoning. In particular, we compute the rotation for each piece, with respect to piece 0, as a simple combination of sequence of `match`es. Starting from the root, each selected match is associated to an angle obtained as the sum of those angles needed for the matches in the path leading to it. Furthermore, we impose that each piece should be reached and that it is reached in a unique way (precisely, with a unique rotation angle):

```
reached(0,0).
reached(P2, Ang1+Ang2):-
       P2>0,
       reached(P,Ang1), degrees(Ang1),
       match(P1,I,P2,J),
       good_match(P,I,P2,J,Ang2).
:- not reached(P,_), polyId(P).
:- polyId(P), reached(P,A1), reached(P,A2), A1!=A2.
```

This allow to discard possible sets of matches that are incoherent, from a rotation point of view, i.e. they build a directed acyclic graph, where a node should be rotated in multiple angles at the same time. The graph is not required to be a (spanning) tree. There can be two paths from the root to the same piece as long as the rotation angle obtained in the two paths is the same. For simplicity, we refer to a tree structure, being the minimally sufficient one to produce a complete placement.

When the tree defined by `match` is generated, we need to check whether the proposed solution is admissible from the spatial point of view. The process requires to actually rotate pieces in place and translate them according to vertices being matched. We compute the `placement` of the pieces on the basis of the chosen match. Starting with the knowledge of the coordinates of vertices of the piece `P1`, the match of `P1` with `P2`, the already computed absolute `Angle`, and the relative shift that should occur between matched vertices, the placement is computed. (As before, we scale numbers by 2^{10} to add 10 binary digits of precision.) We also require that all pieces are placed exactly once and discard solutions that involve multiple placements for any piece.

```
placement(0,I,1024*X,1024*Y):-
       poly_cooR(0,I,X,Y).
placement(P2,J,X2,Y2):-
       P2 > 0, degrees(Angle),
       range(Deltax), range(Deltay),
       best_Shift(P1,P2,Deltax,Deltay),
       reached(P2,Angle),
       cosTable(Angle,Cos), sinTable(Angle,Sin),
       poly_cooR(P2,J,Ox,Oy),
```

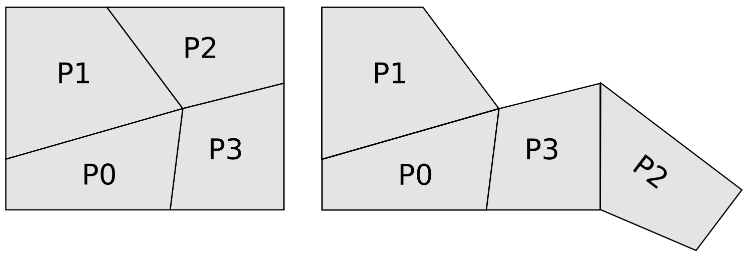

Fig. 2. Two possible solutions

```
       match(P1,_I,P2,J1),
       poly_cooR(P2,J1,Rx,Ry),
       X2 = ((Ox-Rx)*Cos - (Oy-Ry)*Sin)+1024*Deltax,
       Y2 = ((Oy-Ry)*Cos + (Ox-Rx)*Sin)+1024*Deltay.
:-not placement(P1,_,_,_), polyId(P1).
```

The predicate best_Shift computes the new absolute coordinates (Delta) of vertex J of the piece P2, i.e. matching the vertex $I+1$ of the piece P1. Therefore in the placement, P2 is translated so that J lies in the origin, and then it is rotated (around vertex J). Finally, it is translated by Delta, so that vertex J ends up in the correct location.

```
best_Shift(P1,P2,Deltax,Deltay):-
       P2 > 0, range(Deltax), range(Deltay),
       match(P1,I,P2,J),
       numVertices(P1,N1),
       placement(P1,(I+1)\N1,X2,Y2),
       X2=1024*Deltax,
       Y2=1024*Deltay.
```

These constraints exclude matchings that allow two different "best shift":

```
:-best_Shift(P1,P2,DX1,DY1),best_Shift(P1,P2,DX2,DY2),DX1<DX2.
:-best_Shift(P1,P2,DX1,DY1),best_Shift(P1,P2,DX2,DY2),DY1<DY2.
```

At the end the values provided by placement are divided by 2^{10} to rescale in the initial sizes, generating the output predicate poly_out/4 which defines the final positions of pieces. Clearly, more solutions might be possible. For example, Figure 2 shows two solutions to our sample problem. We list here the results for the vertices "0" of the four pieces in the leftmost solution of Figure 2:

```
poly_out(0,0,0,0)           poly_out(1,0,0,4)
poly_out(2,0,14,8)          poly_out(3,0,13,0)
```

Input parameters rangeX/2, rangeX/2, and borders are used to set a bounding box around the object to be reconstructed. If the predicate borders is set to yes, we further require that the sides of the bounding box include a vertex of a piece.

```
%%% Bounding Box:
:- poly_out(A,B,X,Y), rangeX(_,Max),   X > Max.
:- poly_out(A,B,X,Y), rangeX(Min,_),   X < Min.
:- poly_out(A,B,X,Y), rangeY(_,Max),   Y > Max.
:- poly_out(A,B,X,Y), rangeY(Min,Max), Y < Min.

%%% Borders filled
p1 :- poly_out(A,B,MinX,MinY), rangeX(MinX,MaxX), rangeY(MinY,MaxY).
p2 :- poly_out(A,B,MinX,MaxY), rangeX(MinX,MaxX), rangeY(MinY,MaxY).
p3 :- poly_out(A,B,MaxX,MinY), rangeX(MinX,MaxX), rangeY(MinY,MaxY).
p4 :- poly_out(A,B,MaxX,MaxY), rangeX(MinX,MaxX), rangeY(MinY,MaxY).
borders :- p1,p2,p3,p4.
borders :- not borders(yes).
:- not borders.
```

A further constraint is needed to avoid overlaps between pieces.

We experimented with an approach resembling *pixel/raster method* mentioned in the introduction. In this case, the idea consists introducing a discretization of the plane into points and in marking each point depending on which piece covers it. This approach turned out to cause too much inefficiency during the grounding stage and demonstrated not suitable to process instances of reasonable size. We then opted in favor of a technique that directly checks that no pairs of sides intersect. The intuition about this approach can be grasped by considering the following constraint (where, for simplicity, we avoid explicitly listing the auxiliary predicates used in the body). Namely, given the positions of two sides (predicate `segment`) of two different polygons, the constraint rules out each solution where the two segment cross each-other. Crossing condition is detected by considering the position of each vertex of a segment with respect to the other segment (predicate `direction`).

```
:-     polyId(Pid1), polyId(Pid2),
       Pid1 < Pid2,
       placement(Pid1,S1,P1x,P1y),
       placement(Pid1,S1b,Q1x,Q1y),
       placement(Pid2,S2,P2x,P2y),
       placement(Pid2,S2b,Q2x,Q2y),
       segment(Pid1, S1, P1x, P1y, Q1x, Q1y),
       segment(Pid2, S2, P2x, P2y, Q2x, Q2y),
       direction(Pid2, S2, P1x, P1y, D1),
       direction(Pid2, S2, Q1x, Q1y, D2),
       D1*D2<0,
       direction(Pid1, S1, P2x, P2y, D3),
       direction(Pid1, S1, Q2x, Q2y, D4),
       D3*D4<0.
```

However, also this encoding causes a too much large grounding. To improve efficiency, some program transformation techniques (such as folding/unfolding of predicates' definitions and some precomputation of some predicates' extensions) have been applied to the above mentioned encoding, This permitted to achieve overall acceptable running times in the grounding and in the solving stages.

5 Results

In order to test the tool, we wrote a benchmark generator working in the following way. Let us consider a square of size `size` (e.g., made of glass). Suppose now it is broken in exactly p points, where p is 1,2,.... The generator randomly chooses p points and identifies a set of *triangular* pieces as shown in Figure 3.

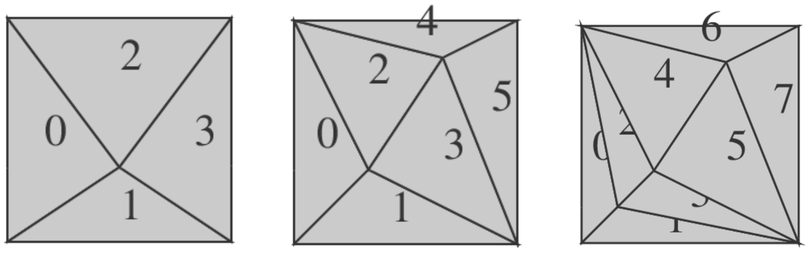

Fig. 3. Three simple benchmark instances, with $p = 1, 2, 3$. Figures comes from a Postscript file generated by running a script on the clingo output

Tolerance in preprocessing has been fixed to 10 (see Section 3). A smaller tolerance would reduce the number of possible `good_match`es (but we decided to leave a bit of redundancy, as needed in real cases where several types of errors can arise). The size of the box is stored in `rangeX` and `rangeY` and used also in `offset`. Thus, the predicate `range` holds for values from $-$`size` to $+$`size`. Borders are required to be matched.

Discretization for angles was set to 1°. This gives a decent degree of approximation but also makes the grounding stage rather heavy. However, experiments made with bigger values (e.g., 2, 3, 10) led to similar running times and to poorer results. If we accept less angles, we need to increase the tolerance to guarantee the same good matches.

For each size and for each number of internal points $p = 1, 2, 3, 4, 5$ (save for very small instances where we used only the small values of p) we generate 5 random instances. In Figure 4 we report the graph of the running times averaged on the 5 instances. Instances' `size` are in the x-axis, running times are in the y-axis, one line for each number of points p is drawn.

Experiments were conducted on Ubuntu Bash of Windows 10 Pro Desktop, Intel i7-2600, 3.4GHz, RAM 12GB, using clingo 4.5.4 (gringo 4.5.4, clasp 3.1.4). Although it does not emerge by looking at the average scenario, it is important to report that the grounding takes most of the time and that the running time strongly relies on how many "good matches" the instance admits. As a limit case, if every side has exactly one good match, matchings are deterministically assigned, and hence running times are close to zero even for large instances.

As one might expect running time grows either as the box size grows or as the number of broken points grows.

Redefining the `range` predicate (currently assigned by the input `offset` parameter) proved to be extremely effective. This sensibly reduces grounding and search time. For instance, in the case of the slowest computation of the set (instance number 5 of the benchmark set with `size` $= 20$ and $p = 5$), replacing the definition of `range(-20..20).` with `range(1..13).` reduced the running time from 326s to less than 2s. Therefore, a good strategy could be the one of starting with a low value for offset.

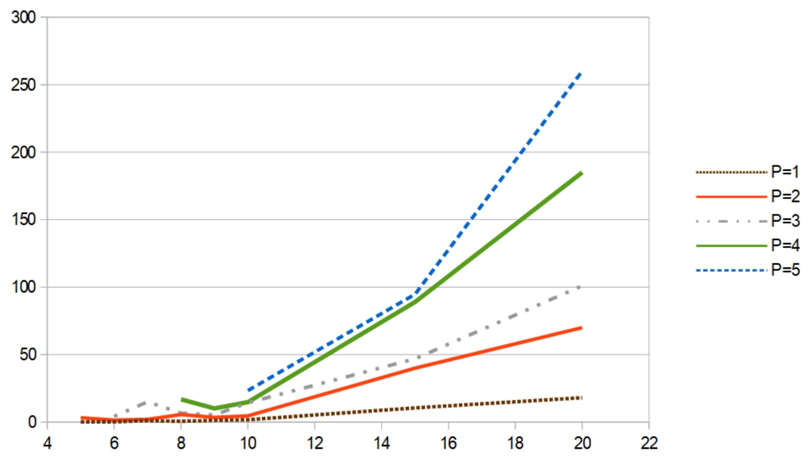

Fig. 4. Results of the running time averaged by random instances of the same size and number of "broken points"

6 Future work and Conclusions

The future work will focus on scaling to larger and real instances. In particular, in order to test ASP capabilities in solving such puzzles, we plan to acquire real jigsaw puzzles (with hundreds of pieces). Some pre-processing with OpenCV will be deployed to identify the polylines that define each piece. Moreover, image color, texture and features around borders can be added to the graph of compatibility by edge attributes, as modelled above. In particular, we plan to build a pipeline that acquires batch of pieces, segments them, extracts the outlines and

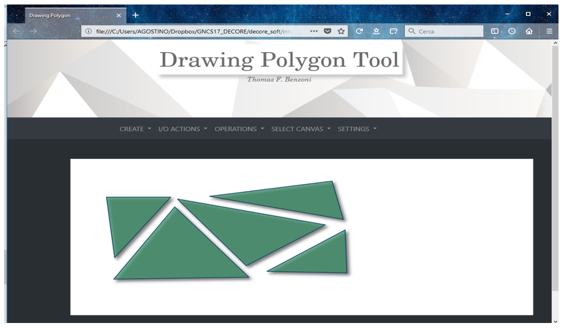

Fig. 5. The visualization/interaction tool

finally pairwise compares the shapes/features. The shape compatibility is vector based and therefore it should efficiently identify the graph. We expect to handle polylines made of hundreds of segments, in order to capture the piece peculiar features. The pre-processed output will describe the `good_match` compatibility graph, to be used as input by the ASP solver. The pre-processing procedure we have in mind is similar to the one described in [21]. However, the reasoning part will take advantage of ASP reasoning engine, which differentiate substantially our work, in comparison to that work and other approaches, e.g. [25, 12, 10]

Current results show that ASP reasoning takes a small fraction of time compared to grounding time. When dealing with larger instances, we expect to face two main issues: the first is the ground program size, which requires more refined models and the handling of integers. The second issue is the search itself, which may show an increase in the reasoning time, due to the NP-complete nature of the problem. In this case, especially for dense graphs with multiple choices for each piece, we plan to introduce more tailored graph-based heuristics.

Another line of research is devoted to support semi-automatic solving process, where partial solutions are iteratively refined, by interleaving domain expert feedback and ASP inferences. This is rather common setup for tasks where the expert can not precisely formalise his/her knowledge into a complete set of rules that drive the identification of the solution. The core of the tool is under development. It is designed to help the assembly process (see Figure 5): it allows to draw and manipulate pieces, to generate the underlying ASP code and to visualize solutions. In the future, it may interface with 2D/3D scanners and guide the assisted matching process with interactive suggestions and auto-completions. Ad-hoc propagators can be exploited during the search following the ideas of [5].

Finally, we believe this problem can well suit a GPU implementation of the ASP solver [7, 8, 6]. In particular, the presence of 2D/3D geometrical manipulations and the viable parallel exploration of (almost) independent regions of the graph represent good candidates for an efficient GPU deployment.

To sum up, this paper represents an initial step for the development of tools for object reconstruction with the advantages of declarative programming and of the speed of modern ASP solvers.

Acknowledgments. The drawing interface was implemented by Thomas F. Benzoni, as part of his bachelors's thesis at University of Perugia. A. Dal Palù, A. Dovier, and A. Formisano are INdAM GNCS members. The research presented in the paper is partially supported by grants: INdAM GNCS 2017 "DECORE", B.I.M. 2018.0419.021, RdB-UniPG2016/17 "YASMIN", and PRID UniUD "ENCASE".

References

1. C. S. Belenguer and E. V. Vidal. An efficient technique to recompose archaeological artifacts from fragments. In *International Conference on Virtual Systems & Multimedia*, pages 337–344. IEEE, 2014.
2. J. A. Bennell and J. F. Oliveira. The geometry of nesting problems: A tutorial. *European Journal of Operational Research*, 184(2):397–415, 2008.
3. M. A. Carravilla, C. Ribeiro, and J. F. Oliveira. Solving nesting problems with nonconvex polygons by constraint logic programming. *International Transactions in Operational Research*, 10(6):651–663, 2003.
4. N. I. Chernov, Y. G. Stoyan, and T. Romanova. Mathematical model and efficient algorithms for object packing problem. *Computational Geometry*, 43(5):535–553, 2010.
5. B. Cuteri, C. Dodaro, F. Ricca, and P. Schüller. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *Theory and Practice of Logic Programming*, 17(5-6):780–799, 2017.
6. A. Dal Palù, A. Dovier, A. Formisano, and E. Pontelli. CUD@SAT: SAT solving on GPUs. *Journal of Experimental & Theoretical Artificial Intelligence*, 27(3):293–316, 2015.
7. A. Dovier, A. Formisano, and E. Pontelli. Parallel answer set programming. In Y. Hamadi and L. Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 237–282. Springer, 2018.
8. A. Dovier, A. Formisano, E. Pontelli, and F. Vella. A GPU implementation of the ASP computation. In M. Gavanelli and J. H. Reppy, editors, *International Symposium on Practical Aspects of Declarative Languages*, volume 9585 of *LNCS*, pages 30–47. Springer, 2016.
9. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
10. D. Goldberg, C. Malon, and M. W. Bern. A global approach to automatic solution of jigsaw puzzles. *Computational Geometry*, 28(2-3):165–174, 2004.
11. D. J. Hoff and P. J. Olver. Automatic solution of jigsaw puzzles. *Journal of Mathematical Imaging and Vision*, 49(1):234–250, 2014.
12. E. Justino, L. S. Oliveira, and C. Freitas. Reconstructing shredded documents through feature matching. *Forensic science international*, 160(2-3):140–147, 2006.
13. M. Kifer and A. Liu. *Declarative Logic Programming: Theory, Systems, and Applications*. ACM, 2018.
14. F. Kleber and R. Sablatnig. Scientific puzzle solving: Current techniques and applications. In *International Conference on Computer Applications and Quantitative Methods in Archaeology: Making History Interactive*, 2009.
15. S. Z. Kovalsky, D. Glasner, and R. Basri. A global approach for solving edge-matching puzzles. *SIAM Journal on Imaging Sciences*, 8(2):916–938, 2015.

16. H. Liu, S. Cao, and S. Yan. Automated assembly of shredded pieces from multiple photos. *IEEE Transaction on Multimedia*, 13(5):1154–1162, 2011.
17. V. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm*, pages 375–398. Springer Verlag, 1999.
18. A. Martinez-Sykora, R. Alvarez-Valdés, J. A. Bennell, R. Ruiz, and J. M. Tamarit. Matheuristics for the irregular bin packing problem with free rotations. *European Journal of Operational Research*, 258(2):440–455, 2017.
19. A. Naif. Solving square jigsaw puzzles using dynamic programming and the Hungarian procedure. *American Journal of Applied Sciences*, 6:1941–1947, 11 2009.
20. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.
21. P. Ondrúşka. Automatic assembly of jigsaw puzzles from digital images. *Bachelor Thesis, Charles University in Prague, Department of Software Engineering*, 2011.
22. R. Pintus, K. Pal, Y. Yang, T. Weyrich, E. Gobbetti, and H. E. Rushmeier. A survey of geometric analysis in cultural heritage. *Computer Graphics Forum*, 35(1):4–31, 2016.
23. D. Pomeranz, M. Shemesh, and O. Ben-Shahar. A fully automated greedy square jigsaw puzzle solver. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 9–16. IEEE Computer Society, 2011.
24. C. Ribeiro and M. A. Carravilla. A global constraint for nesting problems. *Artificial Intelligence Reviews*, 30(1-4):99–118, 2008.
25. M. Ş. Sağiroğlu and A. Erçil. A texture based matching approach for automated assembly of puzzles. In *International Conference on Pattern Recognition*, pages 1036–1041. IEEE Computer Society, 2006.
26. M. Ş. Sağiroğlu and A. Erçil. Optimization for automated assembly of puzzles. *TOP: An Official Journal of the Spanish Society of Statistics and Operations Research*, 18(2):321–338, 2010.
27. M. Shalaby and M. Kashkoush. A particle swarm optimization algorithm for a 2D irregular strip packing problem. *American Journal of Operations Research*, 3:268–278, 2013.
28. E. Sizikova and T. Funkhouser. Wall painting reconstruction using a genetic algorithm. *Journal on computing and cultural heritage*, 11(1):3:1–3:17, Dec. 2017.
29. M. I. Stamatopoulos and C.-N. Anagnostopoulos. A totally new digital 3D approach for reassembling fractured archaeological potteries using thickness measurements. *The e-Journal of the International Measurement Confederation (ACTA IMEKO)*, 6(3):18–28, 2017.
30. P. A. Walega, C. P. L. Schultz, and M. Bhatt. Non-monotonic spatial reasoning with answer set programming modulo theories. *Theory and Practice of Logic Programming*, 17(2):205–225, 2017.
31. P. Y. F. Wu and W. R. Franklin. A logic programming approach to cartographic map overlay. *Computational Intelligence*, 6:61–70, 1990.
32. R. Yu, C. Russell, and L. Agapito. Solving jigsaw puzzles with linear programming. In R. C. Wilson, E. R. Hancock, and W. A. P. Smith, editors, *British Machine Vision Conference*. BMVA Press, 2016.
33. K. Zhang and X. Li. A graph-based optimization algorithm for fragmented image reassembly. *Graphical Models*, 76(5):484–495, 2014.