Geometric reasoning on the Traveling Salesperson Problem: comparing Answer Set Programming and Constraint Logic Programming Approaches

Alessandro Bertagnon[1], Marco Gavanelli[2]

[1]*Department of Environmental and Prevention Sciences, University of Ferrara, C.so Ercole I D'Este, 32, Ferrara, Italy*
[2]*Department of Engineering, University of Ferrara, Via Saragat 1, Ferrara, Italy*

Abstract
The Traveling Salesperson Problem (TSP) is one of the best-known problems in computer science. Many instances and real world applications fall into the Euclidean TSP special case, in which each node is identified by its coordinates on the plane and the Euclidean distance is used as cost function. It is worth noting that in the Euclidean TSP more information is available than in the general case; in a previous publication, the use of geometric information has been exploited to speedup TSP solving for Constraint Logic Programming (CLP) solvers.

In this work, we study the applicability of geometric reasoning to the Euclidean TSP in the context of an ASP computation. We compare experimentally a classical ASP approach to the TSP and the effect of the reasoning based on geometric properties. We also compare the speedup of the additional filtering based on geometric information on an Answer Set Programming (ASP) solver and a CLP on Finite Domain (CLP(FD)) solver.

Keywords
Answer Set Programming, Euclidean Traveling Salesperson Problem, Experimental Comparison of ASP and CLP(FD)

1. Introduction

The Traveling Salesperson Problem (TSP) is a classic Constraint Optimization Problem (COP) that, given a graph with non-negative weights on the edges, involves finding the minimal cost cycle that visits each node exactly once. Many instances and real world applications fall into a special case called *Euclidean TSP*, as also evidenced by the many Euclidean instances present in the famous TSPLIB [1] benchmark set. In the Euclidean TSP each node is identified by its coordinates in the Euclidean plane, and the weight is the Euclidean distance.

The usual way to tackle Euclidean TSPs is to compute the distance matrix and address the problem as a general TSP. However, in the Euclidean TSP more information is available than in the general case: the coordinates of the nodes are available and geometric concepts, like angles and straight lines, can be defined in the Euclidean plane. In a previous publication [2], we showed that the use of geometric information can be exploited to speedup TSP solving

ASPOCP 2023: 16th Workshop on Answer Set Programming and Other Computing Paradigms, July 09–10, 2023, London, UK
alessandro.bertagnon@unife.it (A. Bertagnon); marco.gavanelli@unife.it (M. Gavanelli)
0000-0003-2390-0629 (A. Bertagnon); 0000-0001-7433-5899 (M. Gavanelli)

in Constraint Logic Programming (CLP) solvers. The work started from the observation that, due to the triangular inequality, in a Euclidean TSPs two edges cannot cross each other, since eliminating the crossing would result in another cycle of shorter length. This observation was then extended and exploited in the development of efficient algorithms that reduce the search space.

Beside CLP, Answer Set Programming (ASP) is another logic programming paradigm that due to the expressiveness of the language and the availability of efficient solving systems [3, 4, 5, 6] is used in advanced applications.

In this paper we study the applicability of geometric reasoning to the Euclidean TSP in the context of ASP. We propose an encoding of Euclidean TSP as ASP program with the reasoning based on geometric properties. We compare how solving time is affected when different types of geometric information are exploited in the encoding.

Finally, we compare the speedup of the additional filtering based on geometric information on an ASP solver and a CLP on Finite Domain (CLP(FD)) solver.

2. Preliminaries

An ASP program consists of a set of clauses interpreted in the Stable Model semantics [7]. Each clause in an implication $H \leftarrow B$. The full ASP syntax is reported in [8]. The head H of the clause can be an atom of the form $\mathrm{a}(\mathtt{t}_1, \dots, \mathtt{t}_n)$ (where $\mathtt{t}_1, \dots \mathtt{t}_n$ are terms), a choice atom $\{\mathrm{a}(\mathtt{t}_1, \dots, \mathtt{t}_n)\}$ or an aggregate atom. The body B is a set of literals of the form of the form either `a` or `not a` where again `a` can be an atom or an aggregate atom. An aggregate atom can be $A\{\mathtt{t}_1, \dots, \mathtt{t}_m : \mathtt{c}_1, \dots, \mathtt{c}_m\} \circ \mathtt{n}$ where A can be `#sum`, `#min`, `#max` or `#count`, and $\circ$ can be a relational operator.

Clauses with empty body are called *facts*, while clauses with empty head are called *Integrity Constraints (ICs)*, and their body must evaluate to false in all stable models.

Since each node in a graph of a Euclidean TSP is associated with a point in a plane, in the rest of the paper we will often use the terms “point” and “node” indistinctly.

3. Filtering techniques for the Euclidean TSP

As stated in previous sections, Euclidean TSP instances include the coordinates on the plane of the points; such coordinates in ASP can be represented by facts `point(I,X,Y)` where `I` is a numerical identifier of the point and `X,Y` are its coordinates in the Euclidean plane. Instances also include `cost(A,B,C)` facts where `C` is the Euclidean distance between points `A` and `B`. A cycle on the graph, so a solution, will be represented by a set of `cycle(A,B)` atoms.

The basic Euclidean TSP encoding in ASP is presented in Listing 1. This encoding includes the *degree constraint* that ensures that, in the solution, the degree of each node is equal to two. Line 1 ensures that each point has exactly one outgoing edge while line 2 ensures that each point has exactly one incoming edge. Lines 3–5 state that starting from the point with smallest `I` all other points can be reached through the cycle; this ensures that all points are visited. Line 6 is the objective function; we want to minimize the cost of the cycle.

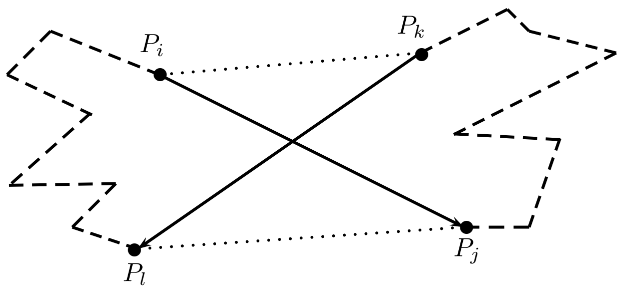


Figure 1: A self-crossing circuit.

```
1 { cycle(A,B) : cost(A,B,_) } 1 :- point(A,_,_).
1 { cycle(A,B) : cost(A,B,_) } 1 :- point(B,_,_).

reached(A) :- A = #min{ I : point(I,_,_) }.
reached(B) :- cycle(A,B), reached(B).
:- point(B,_,_), not reached(B).

#minimize{ C,A,B : cycle(A,B), cost(A,B,C) }.
```

Listing 1: Euclidean TSP encoding in ASP

3.1. Avoiding intersections

The following is a well-known result in the literature.

Theorem 1. *(Flood [9]). The optimal solution of a metric TSP in the plane cannot include two edges that cross each other.*

Intuitively, in Figure 1 instead of taking segments $\overline{P_iP_j}$ and $\overline{P_kP_l}$, a shorter tour chooses the dotted edges $\overline{P_iP_k}$ and $\overline{P_lP_j}$.

The Euclidean TSP is a special case of the metric TSP since the Euclidean distance satisfies the triangular inequality; from Theorem 1, one can avoid, during the search for an optimal Euclidean TSP, those paths that include crossing edges.

Avoiding crossings is particularly simple and declarative in ASP; in Listing 2, line 1 imposes that a pair of segments in the TSP should not cross each other (except, possibly, in one endpoint). The `cross(A,B,C,D)` predicate declaratively states that segment $\overline{AB}$ and segment $\overline{CD}$ cross each other. To verify the intersection of two segments given their endpoints, we implemented the *Faster Line Segment Intersection* algorithm proposed by F.Antonio [10]. The idea of this algorithm is to represent each point as a 2D vector ($\mathbf{A} = (x_a, y_a)$). Now a generic point $\mathbf{P}$ on $\overline{AB}$ can be represented parametrically by a linear combination of $\mathbf{A}$ and $\mathbf{B}$ as follows:

$$\mathbf{P} = \alpha\mathbf{A} + (1 - \alpha)\mathbf{B}$$

where α is in the interval $[0, 1]$ when representing a point on the segment $\overline{AB}$. Finding the crossing point $\mathbf{Q}$ of two segments $\overline{AB}$ and $\overline{CD}$ reduces to solving the following system of equations:

$$\begin{cases} \mathbf{Q} = \mathbf{A} + r(\mathbf{B} - \mathbf{A}) \\ \mathbf{Q} = \mathbf{C} + s(\mathbf{D} - \mathbf{C}) \end{cases}$$

However, we are not interested in explicitly calculating the intersection point, so we do not need r and s explicitly, it is sufficient to verify that r and s are in the range $[0, 1]$ (see lines 2–18).

```
1  :- cycle(A,B), cycle(C,D), cross(A,B,C,D).

2  cross(A,B,C,D):-
3    point(A,Ax,Ay), point(B,Bx,By), point(C,Cx,Cy), point(D,Dx,Dy),
4    Den = (Bx-Ax)*(Dy-Cy)-(By-Ay)*(Dx-Cx),
5    Den > 0,
6    NumR = (Ay-Cy)*(Dx-Cx)-(Ax-Cx)*(Dy-Cy),
7    NumR > 0, NumR < Den,
8    NumS = (Ay-Cy)*(Bx-Ax)-(Ax-Cx)*(By-Ay),
9    NumS > 0, NumS < Den.
10
11 cross(A,B,C,D):-
12   point(A,Ax,Ay), point(B,Bx,By), point(C,Cx,Cy), point(D,Dx,Dy),
13   Den = (Bx-Ax)*(Dy-Cy)-(By-Ay)*(Dx-Cx),
14   Den < 0,
15   NumR = (Ay-Cy)*(Dx-Cx)-(Ax-Cx)*(Dy-Cy),
16   NumR < 0, NumR > Den,
17   NumS = (Ay-Cy)*(Bx-Ax)-(Ax-Cx)*(By-Ay),
18   NumS < 0, NumS > Den.
```

Listing 2: Constraint for avoiding crossing edges

It is interesting to notice that currently available intelligent grounders (such as `gringo` [11]) are able to compile the `cross/4` predicate in Listing 2 into a series of facts, and the integrity constraint in line 1 is grounded into a series of denials that forbid the pairs of edges that intersect. In their turn, these denials help the solver reducing the search space without adding new choices to the code.

3.2. Reasoning on the convex hull

A useful consequence of Theorem 1 is based on the concept of convex hull and is given in Corollary 1. The convex hull of a set of points P in a Euclidean space is the minimum convex set containing all the points. In the plane it corresponds to a convex polygon, and it is completely defined by its vertices. We denote with $H(P)$ the sequence of nodes on the boundary of the convex hull.

Corollary 1. *(Deineko, van Dal, and Rote [12]). Assuming that not all nodes of the graph lie on the same line, nodes on the boundary of the convex hull are always visited in their cyclic order in an optimal TSP.*

Corollary 1 can be exploited in several ways to reduce the search space; the main idea is to define an order of visit (either clockwise or counterclockwise) of the tour, and impose that in all points belonging to the border of the convex hull such order is preserved. In a sense this type of pruning can be thought as a symmetry breaking constraint, and in the experimental evaluation it will be compared with another symmetry breaking technique that imposes one order of visit of the tour.

The points on the boundary of the convex hull will be identified by means of the `hull(X)` predicate. The counterclockwise visiting order of the hull will be encoded through `convex_hull_next(A,B)` facts, which state that point `B` follows point `A` on the boundary of the convex hull.

In [2] we devised three ways to exploit the information about the hull for propagation. In the following we report them together with the corresponding ASP encoding. The first constraint, implemented in Listing 3, impose that the successor of a convex hull vertex cannot be another vertex member of $H(P)$ except for the one that immediately follows it.

```
:- not convex_hull_next(X,Y), cycle(X,Y), hull(X), hull(Y).
```

Listing 3: The successor of a convex hull vertex cannot be another vertex on the boundary of the convex hull except for the one that immediately follows it.

This integrity constraint imposes that edges connecting far away nodes in the border of the convex hull cannot be taken, and are immediately propagated by the unit propagation of the solver with no need of search.

The second way is reasoning on the angle formed by the incoming and the outgoing arcs in hull vertexes: in order to visit nodes in a counterclockwise order, the angle between the incoming edge and the outgoing edge of H_i cannot be negative (it must be between 0 and π) or, stated otherwise, it must correspond to a left turn.

```
:- hull(X), cycle(From,X), cycle(X,To), turn_right(From,X,To).

turn_right(From,P,To):-
  point(From,X,Y), point(P,X0,Y0), point(To,X1,Y1),
  (Y-Y0)*(X1-X0) - (X-X0)*(Y1-Y0) < 0.
```

Listing 4: In order to visit nodes in a counterclockwise order, the angle between the incoming edge and the outgoing edge of a convex hull vertex cannot be negative.

The third way results from stating that any path starting from a convex hull vertex cannot reach any other convex hull vertex except the one that directly follows it. Put it more precisely, each vertex in a path originating from a point H_i cannot reach any vertex in $H(P)$ except for $H_{(i+1 \mod |H(P)|)}$.

```
path_hull(A,B):- hull(A), cycle(A,B), not hull(B).
path_hull(A,C):- hull(A), path_hull(A,B), cycle(B,C), not hull(C).

:- path_hull(A,B), cycle(B,C), hull(C), not convex_hull_next(A,C).
```

Listing 5: Each path originating from a convex hull vertex cannot reach any convex hull vertex except for the one immediately following it.

Predicate `path_hull(A,B)` is true if from a vertex `A` on the border of the convex hull there is a path connecting it to an internal vertex `B` (i.e., a vertex that is not on the boundary of the convex hull) passing only through internal vertexes. The integrity constraint in line 8 imposes that such path cannot terminate in any vertex on the border of the convex hull, except the immediate successor of vertex `A`.

3.2.1. Computing the convex hull

Several algorithms exist to compute the border of the convex hull of a set of vertices, however we preferred to compute it with a declarative program directly in ASP. We can declaratively state that a vertex is on the border of the convex hull if it is not internal to any triangle having as vertexes any three nodes of the graph (line 1 in Listing 6). In order to check if a point `P` is internal to some triangle (predicate `inside` in Listing 6), it is enough to check if it stands always on the same side with respect to the line segments on the sides of the triangle; on its turn, this can be easily defined leveraging on the previously defined predicate `turn_right`.

This approach computes the set of vertices on the border of the hull in $O(n^4)$, if n is the number of vertexes; this is much higher than other available algorithms (e.g., Andrew's monotone chain has $O(n \log n)$ complexity), but it is worth noting that all the computation can be performed by the grounder, and indeed `gringo` provides a ground set of facts so that during the search for an answer set it is not necessary to compute $H(P)$. In future work we will explore other algorithms to compute the convex hull.

Once the points belonging to the boundary of the hull have been found, it is necessary to find what is their cyclic order. Given two points A and B on the boundary of the convex hull, point B is successor to point A, chosen the counterclockwise direction, if there are no other points to the "right" of segment $\overline{AB}$. (see line 5).

Note that all the computation in Listing 6, concerning the calculation of the convex hull, is performed by the grounder therefore it does not affect solving performance.

```
1  hull(P) :- point(P,_,_), not inside(P).

2  inside(P):-
3      point(P1,_,_), point(P2,_,_), point(P3,_,_),
4      turn_right(P,P1,P2), turn_right(P,P2,P3), turn_right(P,P3,P1).

5  convex_hull_next(A,B):- hull(A), hull(B), A != B, not turn_right(A,B,_).
```

Listing 6: Calculation of convex hull in ASP

3.3. Exploiting acyclicity checker

The `clingo` solver includes an efficient acyclicity checker [13]: edges identified with the `#edge` keyword must form an acyclic graph. Such acyclicity can be exploited in TSP solving as follows.

```
1 { cycle(A,B) : cost(A,B,_) } 1 :- point(A,_,_).
1 { cycle(A,B) : cost(A,B,_) } 1 :- point(B,_,_).
```

```
#edge(A,B) : cycle(A,B), A != 1, B != 1.
#minimize{ C,A,B : cycle(A,B), cost(A,B,C) }.
```

Listing 7: Euclidean TSP encoding in ASP with acyclicity constraints

4. Experimental evaluation

To assess the effectiveness of the proposed algorithms, we compared experimentally a classical ASP encoding for the Euclidean TSP with encodings using the reasoning based on geometric properties presented in the previous section. We also devised experiments to compare the speedup of the additional filtering based on geometric information on an ASP solver and a CLP(FD) solver. As CLP(FD) solver, we chose the `ic` library of ECL^{i}PS^e v.7.0 build #54 [14] and as ASP solver we chose clingo 5.6.2[1].

To generate realistic Euclidean TSP instances, we used the generator of the DIMACS challenge [15], that provides instances in two classes: *uniform* and *clustered.* We randomly generated instances from 10 to 35 nodes, in both classes. For each size and class, we generated 16 instances.

Uniform random generated instances consist of integer coordinate points uniformly distributed in a square of 10^3 side. Randomly generated instances consist of clusters of points, whose centers are uniformly distributed in a square of 10^3 side. Each point is then randomly associated with a cluster center and two normally distributed variables, each of which is then multiplied by $10^3/\#nodes$, rounded, and added to the corresponding integer coordinate of the chosen center to obtain the coordinates of the point.

For simplicity, we summarized in Table 1 the structure of the various ASP programs tested in the experimental evaluation. The ASP_BASE program is the reference encoding for TSP in ASP, ASP_HULL and ASP_NOCROSS implement only convex hull-based pruning and crossing removal-based pruning, respectively. Finally, ASP_GEOMETRIC implements both geometric reasoning. ASP_ACY_BASE and ASP_ACY_GEOMETRIC use the Euclidean TSP encoding with acyclicity constraint. Since convex hull-based constraints also behave as symmetry breaking by imposing the direction of cycle, we added the following constraint

```
:- cycle(A,1), cycle(1,B), A>B.
```

to the ASP_BASE program to remove symmetries and have a fair comparison.

Name	Configuration
ASP_BASE	Listing 1 + *symmetry breaking*
ASP_NOCROSS	Listing 1 + 2
ASP_HULL	Listing 1 + 3 + 4 + 5 + 6
ASP_GEOMETRIC	ASP_NOCROSS + ASP_HULL
ASP_ACY_BASE	Listing 7 + *symmetry breaking*
ASP_ACY_GEOMETRIC	Listing 7 + 2 + 3 + 4 + 5 + 6

Table 1
ASP programs tested in the experimental evaluation.

[1]https://potassco.org/clingo/

In CLP(FD), we adopted the *successor* representation. In the *successor* representation, n variables $Next_i$ are defined $Next = (Next_1, Next_2, \dots, Next_n)$; variable $Next_i$ represents the node that follows node i in the circuit, and its initial domain is $\{1, \dots, n\} \setminus \{i\}$. For example, if $n = 5$ and $Next = (3, 5, 4, 2, 1)$ the corresponding tour will be $(1, 3, 4, 2, 5, 1)$.

The basic Euclidean TSP encoding includes an `alldifferent`(*Next*) constraint on the *Next* array of variables, which ensures that each node has exactly one incoming edge (degree constraints), and a `circuit`(*Next*) [16, 17, 18] constraint (sometimes called `nocycle`) that avoids subtours, i.e., cycles of length less than n. This encoding also includes, as redundant representation, a set of *Prev* variables: $Prev_i$ represents the node that precedes node i in the circuit.

CLP_GEOMETRIC program includes the basic CLP(FD) encoding for Euclidean TSP together with constraints for crossing removal and convex hull-based pruning, which in this program also acts as a symmetry breaking constraint. CLP_BASE program, on the other hand, besides the basic encoding, includes only the $Next_1 < Prev_1$ symmetry breaking constraint.

4.1. Results

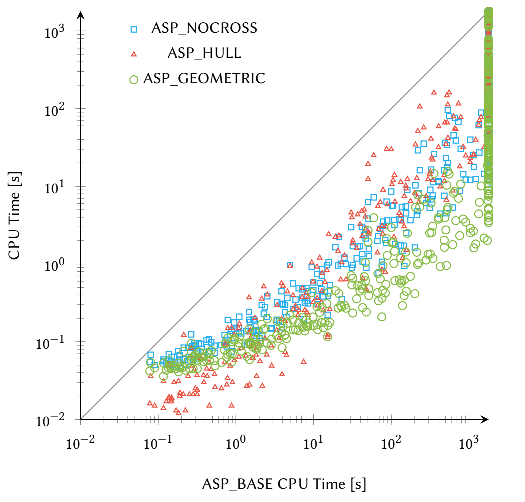
All tests were run with a time limit of 1800s on Intel® Xeon® Processor E5-2630 v3 running at 2.4GHz, using only one core and with 8GB of reserved memory.

Figure 2 compares ASP encodings without acyclicity constraints. In Figure 2a, each point represents one instance solved with ASP_BASE encoding (whose running time is in the x axis) and with one of the encodings exploiting geometric reasoning (whose running time is reported in the y axis). Since all points stand below the $y = x$ straight line, the geometric properties were able to speedup the running time in all instances. Note the log-log scale: speedups from 1 to 3 order of magnitudes were often obtained. Taken individually, both ASP_NOCROSS and ASP_HULL encoding showed a lower solving time when compared with ASP_BASE encoding. However, the best encoding turns out to be ASP_GEOMETRIC containing all the geometric reasoning presented in this paper. The graph in Figure 2b shows in the x-axis the number of instances that could be solved within the runtime plotted in the y-axis. In particular, note how the use of ASP_GEOMETRIC encoding allowed us to solve twice as many instances within the time limit.

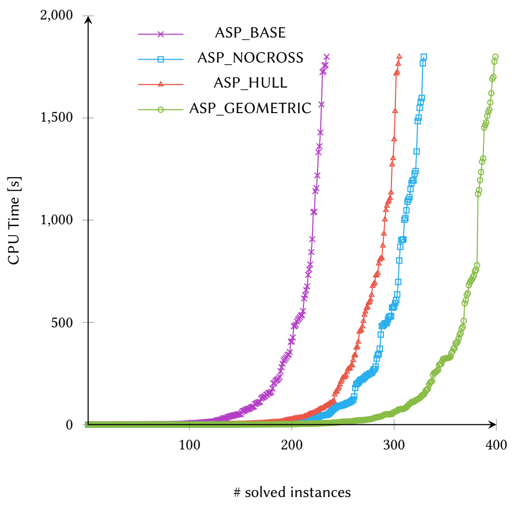
Figure 3 shows the effect of the acyclicity checker; the use of the acyclicity checker provided some improvement, and mostly in conjunction with the use of geometric constraints.

Grounding times of the various encodings are compared in Figure 4. The grounding time of the ASP_GEOMETRIC encoding grows quadratically as the number of `cross`/4 predicates grows quadratically. However, the time required for grounding never exceeds 4 seconds in the larger instances (more than 22 nodes) thus remaining negligible compared to the solving time, which for those same instances greatly exceeds 1000 seconds.

A comparison of how filtering based on geometric information performs differently on an ASP solver and a CLP(FD) solver is presented in Figure 5. We decided to compare the speedup obtained through the exploitation of geometric reasoning. In Figure 5a, where each point represents an instance of Euclidean TSP, the speedup obtained in the CLP(FD) solver (x-axis) is compared with the speedup obtained in the ASP solver (y-axis). On average, the use of geometric reasoning shows, on a single instance, a greater speedup in performance on the ASP solver than on the CLP(FD) solver.

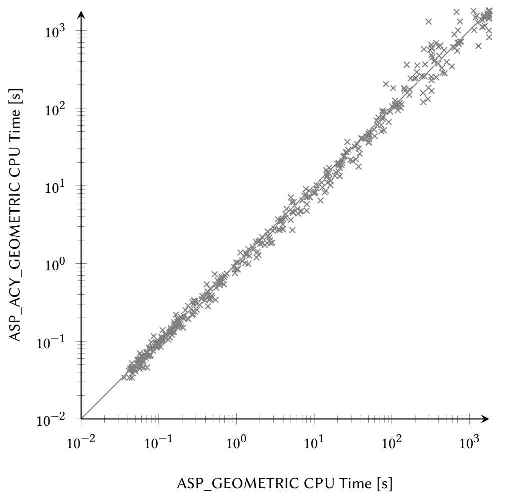


(a)

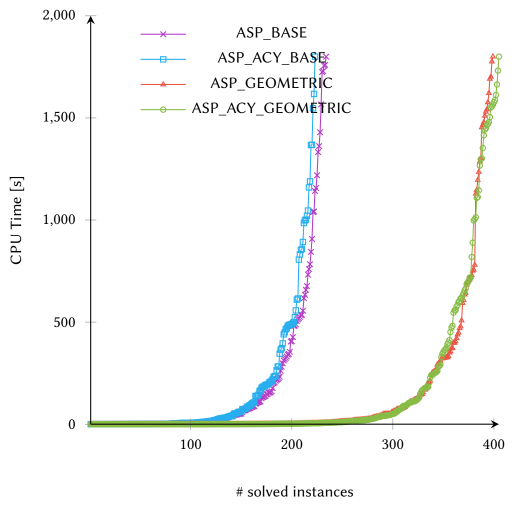


(b)

Figure 2: Comparison of Euclidean TSP ASP encodings using different geometric reasoning.



(a)



(b)

Figure 3: Comparison of ASP encoding using acyclicity constraint.

Considering solving times those recorded on the CLP(FD) solver still remain lower than those shown by the ASP solver. In fact, the cactus plot in Figure 5b shows that CLP_BASE without geometric constraints even succeeds in solving 191 more instances than the encoding ASP_GEOMETRIC that uses geometric constraints instead. If we further compare CLP_GEOMETRIC using geometric reasoning with ASP_GEOMETRIC the gap becomes even more marked as the higher number of instances solved within the time limit increases from 191 to 321.

Considering the comparison between ASP and CLP(FD) approaches, two aspects must be

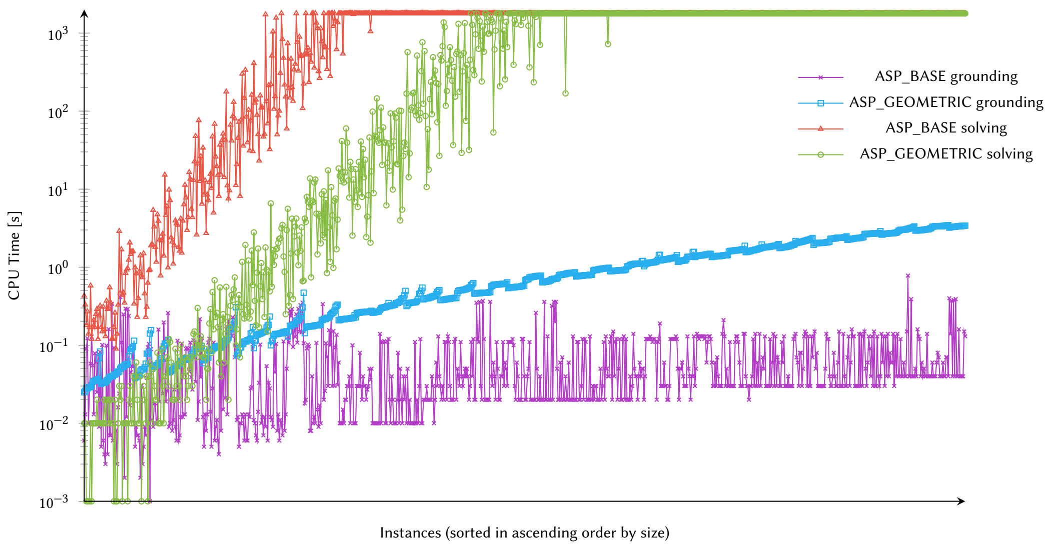


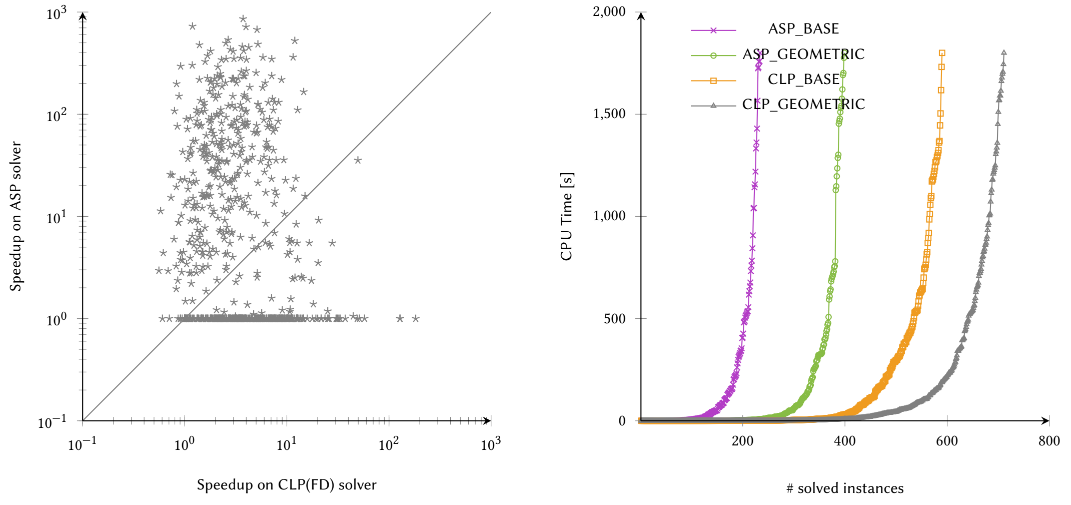
Figure 4: Comparison of grounding time of different Euclidean TSP ASP encodings.

noticed. Indeed, the CLP(FD) approach is faster, and able to solve more instances. On the other hand, the implementation of the propagators took several development time, was based on some non-trivial theorems that helped in reducing the computational complexity of the propagators [2], and takes about 1500 lines of code. The ASP approach, instead, is completely listed in this paper, and is based on declarative considerations, without the need to develop complex propagation algorithms.

In conclusions, we believe that the geometric reasoning on ASP provided a significant speedup and can help extending the applicability of ASP.

5. Related work

ASP was used to solve a Delivery Problem, in which robots have to move items inside a warehouse [19]. The extensive work describes in detail the problem, that can be considered as a problem of multi-agent path finding, and it includes a routing component within the warehouse, in which possible routes are identified within a graph, and it also includes a scheduling component, as the exact timing in which each robot is in each location influences the synchronization of activities. In particular, the robots should not collide, and this is achieved by declaring conflict zones, that can be occupied by at most one robot at the same time. The graphs are very large, so the authors decide to reduce the search space in various ways, possibly excluding the optimal solution but allowing them to obtain good solutions quickly. The graphs they consider are planar, meaning that there are no crossings between edges. They successfully employ clingo[dl] [20].



(a) (b)

Figure 5: Comparison of geometric filtering on an ASP solver (`clingo`) and a CLP(FD) solver (ECL^{i}PS^e).

6. Conclusion

In this paper we applied reasoning based on geometric constraints to the Euclidean TSP (that was developed in a previous publication [2] in the context of Constraint Logic Programming), a widely used case of the famous Travelling Salesperson Problem, within an ASP-based solving approach. The classical encoding of the TSP in ASP was significantly improved thanks to the geometric reasoning: without sacrificing the simplicity, declarativeness and succinctness of the the approach, speedups between 1 and 3 orders of magnitude were easily obtained.

In future work, we plan to investigate the effectiveness of the approach on real-life instances, while so far only randomly-generated instances were used. Another interesting research direction is the use of machine learning to select only those nocrossing constraints that are most effective; this approach was already studied in the CLP(FD) context [21] and provided a further speedup. Moreover, the use of Constraint ASP approaches could provide further improvements and also the geometric reasoning outlined in this paper could be adapted and applied to other routing problems on the Euclidean plane, such as the Euclidean Vehicle Routing Problem or the Euclidean Generalized TSP.

Finally, we plan to study how geometric constraints presented in this paper can be extended into non-Euclidean spaces such as TSP geographic instances where coordinates are expressed in terms of latitude and logitude.

Acknowledgments

This work was partially supported by GNCS-INdAM.

References

[1] G. Reinelt, TSPLIB - A traveling salesman problem library, INFORMS Journal on Computing 3 (1991) 376–384.

[2] A. Bertagnon, M. Gavanelli, Improved filtering for the Euclidean traveling salesperson problem in CLP(FD), in: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020, AAAI Press, 2020, pp. 1412–1419. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5498.

[3] T. Syrjänen, I. Niemelä, The smodels system, in: T. Eiter, W. Faber, M. Truszczynski (Eds.), Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings, volume 2173 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 434–438. URL: https://doi.org/10.1007/3-540-45402-0_38. doi:10.1007/3-540-45402-0_38.

[4] E. Giunchiglia, Y. Lierler, M. Maratea, Answer set programming based on propositional satisfiability, J. Autom. Reason. 36 (2006) 345–377. URL: https://doi.org/10.1007/s10817-006-9033-2. doi:10.1007/s10817-006-9033-2.

[5] F. Calimeri, M. Manna, E. Mastria, M. C. Morelli, S. Perri, J. Zangari, I-DLV-sr: A stream reasoning system based on I-DLV, Theory Pract. Log. Program. 21 (2021) 610–628. URL: https://doi.org/10.1017/S147106842100034X. doi:10.1017/S147106842100034X.

[6] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory Pract. Log. Program. 19 (2019) 27–82. URL: https://doi.org/10.1017/S1471068418000054. doi:10.1017/S1471068418000054.

[7] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: R. A. Kowalski, K. A. Bowen (Eds.), ICLP, MIT Press, 1988, pp. 1070–1080.

[8] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, ASP-Core-2 input language format, Theory Pract. Log. Program. 20 (2020). URL: https://doi.org/10.1017/S1471068419000450. doi:10.1017/S1471068419000450.

[9] M. M. Flood, The traveling-salesman problem, Operations Research 4 (1956).

[10] F. Antonio, Faster line segment intersection, in: D. Kirk (Ed.), Graphics Gems III, Academic Press Professional, Inc., San Diego, CA, USA, 1992, pp. 199–202.

[11] M. Gebser, A. Harrison, R. Kaminski, V. Lifschitz, T. Schaub, Abstract gringo, Theory Pract. Log. Program. 15 (2015) 449–463. URL: https://doi.org/10.1017/S1471068415000150. doi:10.1017/S1471068415000150.

[12] V. G. Deineko, R. van Dal, G. Rote, The convex-hull-and-line traveling salesman problem: A solvable case, Inf. Process. Lett. 51 (1994) 141–148.

[13] J. Bomanson, M. Gebser, T. Janhunen, B. Kaufmann, T. Schaub, Answer set programming modulo acyclicity, Fundamenta Informaticae 147 (2016) 63–91.

[14] J. Schimpf, K. Shen, Ecl^{i}ps^{e} - from LP to CLP, Theory and Practice of Logic Programming 12 (2012) 127–156.

[15] J. Cirasella, D. S. Johnson, L. A. McGeoch, W. Zhang, The asymmetric traveling salesman problem: Algorithms, instance generators, and tests, in: A. L. Buchsbaum,

J. Snoeyink (Eds.), Algorithm Engineering and Experimentation, Third International Workshop, ALENEX 2001, Washington, DC, USA, January 5-6, 2001, Revised Papers, volume 2153 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 32–59. URL: https://doi.org/10.1007/3-540-44808-X_3. doi:10.1007/3-540-44808-X_3.

[16] N. Beldiceanu, E. Contejean, Introducing global constraints in CHIP, Math. Comput. Model. 20 (1994) 97–123.

[17] Y. Caseau, F. Laburthe, Solving small TSPs with constraints, in: L. Naish (Ed.), Logic Programming, Proceedings of the Fourteenth International Conference on Logic Programming, Leuven, Belgium, July 8-11, 1997, MIT Press, 1997, pp. 316–330.

[18] L. G. Kaya, J. N. Hooker, A filter for the circuit constraint, in: F. Benhamou (Ed.), Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings, volume 4204 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 706–710.

[19] D. Rajaratnam, T. Schaub, P. Wanko, K. Chen, S. Liu, T. C. Son, Solving an industrial-scale warehouse delivery problem with answer set programming modulo difference constraints, Algorithms 16 (2023). URL: https://www.mdpi.com/1999-4893/16/4/216. doi:10.3390/a16040216.

[20] T. Janhunen, R. Kaminski, M. Ostrowski, S. Schellhorn, P. Wanko, T. Schaub, Clingo goes linear constraints over reals and integers, Theory Pract. Log. Program. 17 (2017) 872–888. URL: https://doi.org/10.1017/S1471068417000242. doi:10.1017/S1471068417000242.

[21] E. Bellodi, A. Bertagnon, M. Gavanelli, R. Zese, Improving the efficiency of Euclidean TSP solving in constraint programming by predicting effective nocrossing constraints, in: M. Baldoni, S. Bandini (Eds.), AIxIA 2020 - Advances in Artificial Intelligence - XIXth International Conference of the Italian Association for Artificial Intelligence, Virtual Event, November 25-27, 2020, Revised Selected Papers, volume 12414 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 318–334. URL: https://doi.org/10.1007/978-3-030-77091-4_20. doi:10.1007/978-3-030-77091-4_20.