New encodings of the (Euclidean) travelling salesperson problem in constraint answer set programming on difference logic

ALESSANDRO BERTAGNON, *Department of Environmental and Prevention Sciences, University of Ferrara, C.so Ercole I D'Este, 32, Ferrara, Italy.*
E-mail: alessandro.bertagnon@unife.it

MARCO GAVANELLI, *Department of Engineering, University of Ferrara, Via Saragat 1, Ferrara, Italy.*
E-mail: marco.gavanelli@unife.it

Abstract

The Travelling Salesperson Problem (TSP) is a very well-known problem in computer science. Many real-world instances belong to the class of Euclidean TSP, in which the nodes to be visited lie on the Euclidean plane, and additional information is available with respect to the generic TSP, i.e. the coordinates of the nodes to be visited are known. In previous publications, we showed that the additional available information can be exploited to speed up the search, both in Constraint Logic Programming (CLP) and in Answer Set Programming (ASP). Constraint ASP (CASP) is a framework that joins CLP and ASP, and it aims at combining the features of both languages. In this article, we address the (Euclidean) TSP in CASP, and more specifically in the clingo[*DL*] language and solver. We propose new encodings for the TSP in clingo[*DL*]; the new encodings are applicable to the general TSP (also to instances that are not Euclidean) and show a speedup of several orders of magnitude with respect to previous encodings. A further speedup can be obtained in Euclidean instances by exploiting geometric reasoning.

1 Introduction

One of the best-known problems in computer science is the famous Travelling Salesperson Problem (TSP); its name derives from the classic description of the problem: a Travelling Salesperson has to visit a set of cities exactly once, minimizing the total travelled distance. Formally, a graph $G = (N, E)$ is given with weights on the edges $w : E \mapsto \mathbb{R}^+$; the objective is to find a closed path on the graph (a *circuit*) visiting all nodes exactly once, passing from one node to the following only taking existing edges, and minimizing the total weight of the edges in the circuit (the total length of the circuit).

The weight of an edge is also called the *distance* between the two connecting nodes.

One often recurrent case of TSP is the Euclidean TSP, in which each node corresponds to a point in the plane, there is an edge for each pair of nodes, and the distance between two nodes is the Euclidean distance, i.e. it corresponds to the length of the segment connecting the two points. As the general TSP, also the Euclidean version is NP-Hard, although it admits a Polynomial Time Approximation Scheme [3, 43]. The usual approach to solve a Euclidean instance is to convert it into a general TSP (by computing a distance matrix) and then solving it with a usual TSP solver. Such an approach makes sense, since there exist very fast TSP solvers, such as Concorde [2].

Concorde is a solver based on a variety of techniques, including Integer Linear Programming and Local Search, and can solve in a short time very large instances (e.g. instances in the order of tens of thousands of nodes have been solved). Moreover, it is a *complete* approach, meaning that it is able to find the true optimal solution and also to prove that better solutions cannot exist (as opposed to

Vol. 36, No. 2, © The Author(s) 2026. Published by Oxford University Press.
This is an Open Access article distributed under the terms of the Creative Commons Attribution License (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted reuse, distribution, and reproduction in any medium, provided the original work is properly cited.
https://doi.org/10.1093/logcom/exaf072

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

heuristic methods, which can only find near-optimal solutions, and even when they find the optimal, they are unable to prove that no better solutions can exist).

On the other hand, Concorde is able to solve only the TSP, and it cannot solve all variants of the TSP or other routing problems. Often, real-life problems are not as simple as academic problems, and, in the specific case we are addressing, they might include a routing component, in which the objective is to find a route, possibly minimizing the travelled distance, but also have to satisfy the so-called *side constraints*. For example, in the TSP with Time Windows the nodes cannot be reached at any possible time, but only within a time interval (e.g. they might represent shops or offices with opening hours); in the Generalized TSP one does not need to visit all nodes, but only a subset; in the Vehicle Routing Problem there are more than one vehicle available, and all start from a common node *depot* and there are capacity constraints on the vehicles. These are only some of the endless variants that may appear in real-life problems, and they cannot be addressed by algorithms tailored for the TSP. For these reasons, a variety of other more general techniques, which can accommodate side constraints or additional decision variables, are also used to solve routing problems; such techniques include Mixed-Integer Linear Programming (MILP), Constraint Logic Programming (CLP), Constraint Programming (CP) and Answer Set Programming (ASP), just to name a few. Notably, some of these languages are *declarative*, meaning that the user only has to state the problem in a declarative way (through logic sentences or constraints), while it is solely a task of the solver to find the solution of the problem, using clever techniques to reduce the search space (constraint propagation in the case of CLP or CP, linear relaxation in the case of MILP, or clause learning in the case of the most efficient modern ASP solvers [1, 26]).

With these techniques, it was recently found that exploiting the additional information naturally present in the Euclidean case can reduce the search space and help find the optimal solution in a shorter time [9, 11]. For instance, in a Euclidean instance, not only the distance matrix is available, but also the coordinates of the nodes on the Euclidean plane are known. Such information can be enriched with the vast library of mathematical concepts defined since the dawn of geometry in the Euclidean plane, such as straight lines, segments, angles, linear combinations and convex hulls, just to name a few. The exploitation of such techniques to speed up Euclidean TSP solving were first successfully employed in the CLP on Finite Domains (CLP(FD)) research area [9], and improved through the integration of machine learning techniques [7]. More recently, these ideas were extended to the ASP logic language [11], where the speed-up was even more noticeable, and where the problem definition was significantly easier.

To take advantage of both the ease of formulation offered by the ASP approach and the efficiency of CLP propagation techniques, various Constraint ASP (CASP) solvers have been proposed [4, 5, 18, 19, 28, 33, 37, 40, 41, 47, 48]. Among the constraint languages used in CASP, Difference Logic is particularly well-suited for modelling problems involving inequalities over integer variables of the form $x - y \leq c$.

In this context, clingo[*DL*] [34] is an ASP solver that integrates Difference Logic constraints into the ASP framework. Although it supports a limited fragment of constraints, this restriction enables the use of highly specific and efficient propagation algorithms.

In this work, we have a twofold contribution. First, we introduce new clingo[*DL*] encodings of the TSP; with the new encodings, we were able to find the optimal solution with a speed-up of several orders of magnitude with respect to the existing encoding in the literature, pushing the state of the art in TSP solving in the CASP realm with Difference Logic.

Second, we extend the works in [9] and [11] about Euclidean TSP to the CASP realm. Since the language of clingo[*DL*] was enough to express the Euclidean TSP, such an approach is highly promising. We apply the techniques developed in the ASP encodings to speed up Euclidean instances also to the clingo[*DL*] encodings; the results show that even in the new encodings a further speed

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

up is obtained when exploiting the additional information in Euclidean instances. An extensive experimentation compares several ASP and clingo[*DL*] encodings.

The rest of the article is organized as follows. The next section introduces the required preliminaries. Section 3 explains two TSP encodings in ASP. Section 4 recaps geometric reasoning techniques for the Euclidean TSP, and their implementation in ASP. Section 5 proposes new encodings of the TSP and of the Euclidean TSP for clingo[*DL*]. Section 6 describes the experimental evaluation and its results. Related work is discussed in Section 7, and, finally, we conclude.

2 Preliminaries

An ASP program Π is a finite set of implications $H \leftarrow B$ called *clauses*. We report here the subset of the syntax of interest for this article; the interested reader can find the full ASP syntax in [12]. The head H is called the *head* of the clause, and it is either an atom of the form `a(t1,...,tn)` (where `t1,...tn` are terms), or a choice atom `{a(t1,...,tn)}` or an aggregate atom. B is the *body* of the clause and consists of a set of literals of the form either `a` or `not a`, where `a` can be an atom or an aggregate atom. Aggregate atoms have the form $A\{\mathtt{t}_1,\ldots,\mathtt{t}_m:\mathtt{c}_1,\ldots,\mathtt{c}_m\} \circ \mathtt{n}$ where A is one of the keywords `#sum`, `#min`, `#max` or `#count`, while $\circ$ is a relational operator belonging to the set `{=, <,>, >=, <=, !=}`, where `!=` stands for $\neq$. If the body of a clause is empty, it is intended as the special value *true*, and the clause is called a *fact*. If the head of a clause is empty, it is intended as the special value *false*, and the clause is called an *Integrity Constraint (IC)*.

A choice rule of the form `{a}:-`*Body* can also be rewritten as a pair of clauses without choice rules:

```
a:-Body,not na.
na:-Body,not a.
```

where `na` is a new atom not occurring elsewhere in the program.

The declarative semantics of an ASP program Π is defined by the Stable Model semantics [30] of a corresponding ground program.

While a program Π can contain variables, a ground program $gr(\Pi)$ can be built as an equivalent program that does not contain variables. Notice that in general even if Π is finite, its grounding $gr(\Pi)$ could be infinite; for this reason, the various ASP solvers impose syntactic restrictions that ensure that there exists an equivalent ground program.

An interpretation I of a ground program Π is a subset of the set of atoms that occur in Π; an atom $a \in I$ is considered *true* in I, while if $a \notin I$, then a is *false* in the interpretation I.

Given a ground program Π and an interpretation I, the *reduct* Π^I of Π with respect to I is a program defined as follows. All rules whose body is false in I are removed. An interpretation I is a *stable model* of program Π if there is no $J \subset I$ such that J is a model for Π^I [20].

The semantics of aggregate atoms follows the stable model semantics extended for aggregates [20]. Intuitively, an aggregate atom $A\{\mathtt{t}_1,\ldots,\mathtt{t}_m:\mathtt{c}_1,\ldots,\mathtt{c}_m\} \circ \mathtt{n}$ is true under an interpretation I if the result of applying the aggregation operator A to the values of the terms t_i for which the conditions c_i hold in I, satisfies the comparison with respect to n using the relational operator $\circ$.

In order to simplify writing ASP encodings of complex problems, the standard ASP syntax [12] also contains *conditional literals*. A conditional literal has syntax $A : B$, where A is a literal and B is a conjunction of literals. A conditional literal occurring in the body of a predicate is expanded at grounding time as a conjunction of literals; the meaning of $A : B$ is the conjunction of all instances of A such that B is true, or $\bigwedge_B A$. For example,

```
1 p:- a(X):b(X),c(X).
```

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

means that `p` is true if `a(X)` is true for all instances of `X` that make true the conjunction `b(X),c(X)`.

Intuitively, a CASP program Π is an ASP program in which the head of rules can also be a special atom C called a *constraint*; more precisely, a CASP program is built on a set of atoms $\mathcal{A} \cup \mathcal{L}$ where $\mathcal{A}$ is the set of *regular atoms* while $\mathcal{L}$ is the set of *constraint atoms* [34]. Regular atoms correspond to those used in standard ASP programs, and the term is used here to distinguish them from constraint atoms introduced in the CASP extension. While the meaning of regular atoms is defined in the logic program, the meaning of the constraint atoms is defined in an external theory $\mathcal{T}$. Given a ground set of constraint atoms $\mathcal{S}$, a *solution* is an assignment of numeric values to the numeric variables such that all the constraints in $\mathcal{S}$ are satisfied in $\mathcal{T}$. For example, the atom `3*x <= 7` can be interpreted in the theory of linear equations over integers, and one solution is that the numeric variable `x` takes value 1.

A set $X \subseteq \mathcal{A} \cup \mathcal{L}$ is a constraint answer set of a program Π if there is some solution $\mathcal{S}$ such that X is a stable model of the logic program[1]

$$\Pi \cup \{\{a\}.|a \in (\mathcal{L} \setminus H(\Pi)) \cup \mathcal{S}\} \cup \{\texttt{:-}a|a \in (\mathcal{L} \setminus H(\Pi)) \setminus \mathcal{S}\}.$$

where $H(\Pi)$ is the set of atoms occurring in the head of a rule in Π.

In particular, in clingo[*DL*] [34], a constraint has the form $a - b \leq c$ where c is an integer constant, while a and b are terms (which in the theory of integer linear equations are interpreted as numeric variables, named also *DL variables*). A set of *DL* constraints is satisfied if there exist integer values that can be assigned to all numeric variables occurring in the constraints, satisfying all the constraints.

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 TSP encodings in ASP

3.1 Input data

The input data will be provided by means of predicates.

Since each node of the graph corresponds to a point in the plane, we will exploit the information about coordinates of the points in order to improve the solution time of the TSP. The points to be visited are provided by the predicate `point/3`; it is usually defined as a set of facts of the form `point(I,X,Y)`, where `I` is the node identifier, while `X` and `Y` are the coordinates of the point in the plane. To simplify the presentation, we will assume that the node identifier is an integer, although this is not a strictly necessary assumption; the only strong requirement is that the identifier of each node is unique.

The edges and their weights are represented as facts `cost/3`; a fact `cost(A,B,C)` will represent an edge from node `A` to node `B` having weight (or cost) `C`.

3.2 A Simple ASP encoding of the TSP

One common encoding of the TSP in ASP is shown in Listing 1. The solution is provided by a predicate `cycle/2`; an atom `cycle(A,B)` in the answer set will represent the fact that the travelling salesperson will take the edge from node `A` to node `B`.

[1] To simplify the exposition, we refer to the *non-strict* atoms only [34], since by default clingo[*DL*] uses the non-strict semantics.

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

The first rule (line 1) ensures that each node A in the graph is visited, and that there is exactly one outgoing edge from A to any other node. Similarly, rule 2 states that for each node B there is exactly one incoming edge.

These two rules alone do not forbid the set of edges to form two or more separate sub-tours; to ensure the edges selected to be in the TSP actually form a single route, the IC in line 3 imposes that all nodes should be reachable from an arbitrary initial node following only edges in the TSP.

Predicate `reached/1` is true for all the nodes reachable from the node having minimum identifier.

Listing 1: Euclidean TSP encoding in ASP

```
1 { cycle(A,B): cost(A,B,_) }=1:- point(A,_,_).
2 { cycle(A,B): cost(A,B,_) }=1:- point(B,_,_).
3:- point(B,_,_), not reached(B).
4 firstpoint(X):- X = #min{ Y: point(Y,_,_) }.
5 reached(X):- firstpoint(X).
6 reached(B):- cycle(A,B), reached(A).
```

Finally, the objective function is reported in Listing 2. The function to be minimized is the sum of the costs C such that an edge $\overline{\mathtt{AB}}$ is in the tour.

Listing 2: Simple objective function for the TSP in ASP

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

In the following, we will refer to this encoding (Listings 1 and 2) as the ASP Simple Encoding (ASP-SE) to distinguish it from other encodings introduced later. Although well-known and intuitive, ASP-SE suffers from a poor propagation of the bound in the branch-and-bound computation. For example, consider the TSP instance with 6 nodes shown in Figure 1. At a certain point in the computation, a solution has been found with a cost 50. In a successive state of the computation, the search algorithm has already assigned 3 edges, totaling a partial cost of 30; the partial circuit 1-2-3-4 is already fixed. The figure shows only the remaining available edges, along with their associated costs. To complete the circuit, three additional edges are needed. We can immediately notice that even if we could choose three edges having cost 7 (the minimum that occurs in the instance), we would obtain a cost of 51, which is worse than the incumbent solution, and the search could be immediately stopped (possibly backtracking to a more promising node). On the other hand, the summation in Listing 2 cannot detect, in the current step of the computation, that the final cost will definitely be superior than that of the incumbent, and further search is necessary to try and assign the remaining edges to complete a circuit; clearly, as all possible combinations will provide a worse value than 50, this search is fruitless.

3.3 Advanced encoding

An objective function that performs better than the one in Listing 2, while remaining fully compatible with the encoding in Listing 1, was published in [27]. In current ASP syntax [12], it can be written as follows:

Listing 3: Advanced objective function for the TSP in ASP

```
1 outcost(X, C):- cost (X, _, C).
2 order(X, C1, C2):- outcost(X, C1), outcost(X, C2),
3   C1 < C2,
4   C <= C1: outcost(X,C), C < C2.
```

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

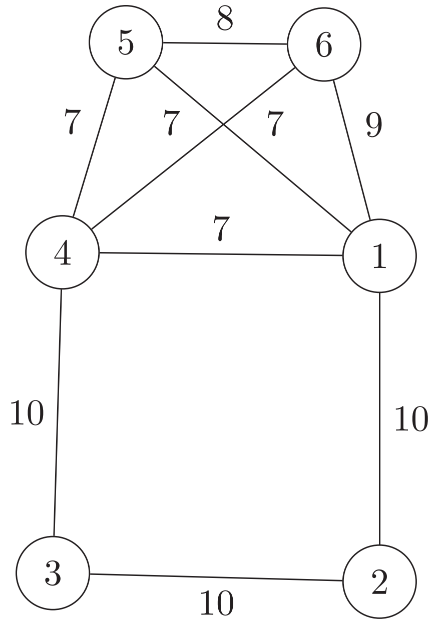

FIGURE 1. Example of propagation. Graph drawn with ASPECT [10].

```
5 penalty(X, C1, C2-C1):- order(X, C1, C2),
6   cycle(X, Y), cost(X, Y, C2).
7 penalty(X, C1, C2-C1):- order(X, C1, C2),
8   penalty(X, C2, _).
9 #minimize { C,X,P1: penalty (X, P1, C) }.
```

In the following, we will refer to the combination of the encoding in Listing 1 with the advanced objective function in Listing 3 as the ASP encoding with PEnalty (ASP-PE).

Predicate `outcost(X,C)` (line 1) is true if `C` is a cost of an edge outgoing from node `X` (independently of which edge or how many edges actually provide such cost).

The `order/3` predicate (line 2) establishes an order between the costs of edges outgoing from each node. The conditional literal in line 4 is expanded, at grounding time, into a conjunction of inequalities; logically, it can be written as $\bigwedge_{outcost(X,C),C<C2} C \leq C1$, or $\bigwedge\{C \leq C1 \mid outcost(X,C), C < C2\}$, and read as *all outcosts that are smaller than C2 are also smaller than or equal to C1*. For example, in the graph in Figure 1, node 1 has four outgoing edges, with costs 7, 7, 9 and 10, so predicate `outcost` is true for `outcost(1,7)`, `outcost(1,9)`, `outcost(1,10)`. Consider now predicate `order`: if we take `C1=7` and `C2=10`, the conditional literal in line 4 is expanded to the set of atoms {`C <= 7 | outcost(1, C), C < 10`} i.e. there is an atom `C <= 10` for all values of `C` that satisfies the goal `outcost(1, C), C < 10`, namely `C=7` and `C=9`. The conditional literal is expanded to `7<=7, 9<=7`, which evaluates to `false`. This means that `order(1,7,10)` will not be in the answer set: this respects the intended meaning, since the cost immediately following 7 is not 10, but it is 9. The reader can verify that the answer set will contain `order(1,7,9)` and `order(1,9,10)`. Although the condition

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

may become very long, the predicate `order/3` is completely expanded to facts by the intelligent grounder `gringo`.

Let us name $c_1, c_2, c_3, \ldots,$ the costs for outgoing from node x, in increasing order. If we select an edge having cost c_k, the set of `penalty` atoms in the answer set will be $\{\texttt{penalty}(x, c_i, c_{i+1} - c_i) \mid 1 \leq i < k\}$, i.e. there will be the contributions $\{c_2 - c_1, c_3 - c_2, \ldots c_k - c_{k-1}\}$. The #minimize statement (line 9) adds all these contributions and minimizes them. The two versions of the objective functions differ for a constant: in the advanced version, for each node the minimum outgoing cost is not counted. Since they differ for a constant, the optimum obtained with one function is also optimum for the other function.

Let us now reconsider the example presented in Figure 1. For the sake of clarity and to simplify the exposition, we assume that both objective functions yield the same value. Specifically, we suppose that the advanced objective function also includes, for each node, an additional term corresponding to the minimum outgoing cost. This could be achieved by adding a further clause:

```
1 penalty(X, P, P):- P=#min{C:outcost(X, C)}.
```

Now, in the situation depicted in Figure 1, the `penalty` predicate contains a contribution of $3 \cdot 10$ (for the edges 1-2, 2-3 and 3-4) and would also contain a contribution of the minimum cost value of outgoing edges from nodes 4, 5 and 6, i.e. $3 \cdot 7$. Such contributions total $3 \cdot 10 + 3 \cdot 7 = 51$, which is higher than the cost of the incumbent solution, so the search can immediately stop, even if the solver has not assigned the remaining edges, saving the exploration of a significant part of the search space.

4 Geometric reasoning for the Euclidean TSP

In this section, we shortly recap the filtering based on geometric reasoning, and its implementation in ASP [11].

4.1 Avoiding intersections

A well-known result in the literature of the Euclidean TSP (see, e.g. [22]) is that an optimal TSP cannot include two edges that cross each other, unless the Euclidean TSP instance is trivial.

DEFINITION 1.
A Euclidean TSP instance is *trivial* if the points corresponding to the nodes of the graph all lie on the same straight line.

THEOREM 1.
(Flood [22]). If c^* is an optimal solution of a not trivial Euclidean TSP (see Definition1), then for each pair of edges $e_{i,j}, e_{k,l} \in c^*$, the edges $e_{i,j}$ and $e_{k,l}$ can intersect only in the endpoints.

The result is pretty intuitive, since, in case of crossing, there exists a 2-opt move (i.e. a modification of the solution swapping two edges) that provides a better solution. In fact, given a tour containing a crossing, a better tour can be obtained by removing the crossing edges and reconnecting the two parts of the tour avoiding the intersection; in the example in Figure 2, instead of taking the edges $e_{i,j}$, and $e_{k,l}$, a shorter tour takes $e_{i,k}$ and $e_{j,l}$. The new tour is feasible, because in a Euclidean TSP all pairs of nodes are connected by an edge, and because the triangular inequality holds (so the weight $w(e_{a,b}) \leq w(e_{a,c}) + w(e_{c,b})$ for all triplets of edges $e_{a,b}, e_{a,c}$ and $e_{c,b}$).

In order to detect if a segment $\overline{AB}$ crosses another segment $\overline{CD}$, one can search for the intersection point by writing a system of linear equations. Each interior point of a segment $\overline{AB}$ can be obtained

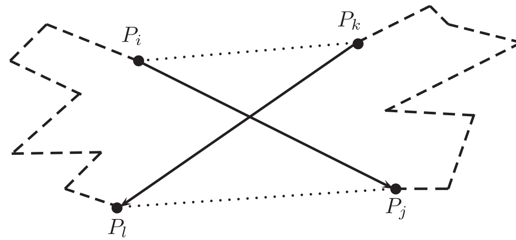

FIGURE 2. A self-crossing circuit.

by the convex combination of the coordinates of the two extremes: $\mathbf{P} = \lambda\mathbf{A} + (1-\lambda)\mathbf{B}$, where $\lambda \in]0,1[$. The open interval ensures that endpoints are excluded, as we are only concerned with interior intersections.

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

By solving the system of equations, it is possible to define a relation stating when two segments have an intersection. The encoding in ASP of such a relation can be found in [11], where a predicate `cross(A,B,C,D)` is provided, which is true if segments $\overline{\mathtt{AB}}$ and $\overline{\mathtt{CD}}$ intersect. Most importantly, the predicate is expanded by intelligent grounders, such as `gringo` [29], into a set of facts, allowing for very efficient processing. In order to remove crossings, adding the following IC suffices:

Listing 4: IC to avoid crossing edges
```
2 :- cycle(A,B), cycle(C,D), cross(A,B,C,D).
```

4.2 Convex hull reasoning

A well-known consequence of Theorem 1 is that the external points of the TSP must always be visited in order.

More precisely, the convex hull of a set of points in the Euclidean plane is the smallest convex set that contains all the points; if the set of points is finite, it corresponds to a convex polygon. The vertices of such a polygon will be denoted with $\mathcal{H}(P) = \langle H_0, H_1, \ldots, H_{|\mathcal{H}(P)|-1}\rangle$, and we will assume they are visited in counterclockwise order.

COROLLARY 1.
(Deineko, van Dal and Rote [16]). Unless 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.

In previous publications, we devised three ways to exploit Corollary 1 to speed up the search, and they were successful both in CLP [9] and in ASP [11].

In ASP, the points of the nodes on the boundary of the convex hull are provided through the `hull/1` predicate. A counterclockwise order is provided by means of the `convex_hull_next/2` predicate: `convex_hull_next(A,B)` means that vertex `B` follows immediately vertex `A` in the counterclockwise visit of the vertices in $\mathcal{H}(P)$.

The pruning obtained through this type of reasoning can be thought as a type of symmetry breaking; clearly in the experimental evaluation we will compare it with another symmetry breaking technique in order to have a fair comparison.

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

The first type of pruning imposes that the successor of a convex hull vertex cannot be any other vertex on the boundary of the convex hull, except for the immediate successor in the counterclockwise order:

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

The second method is based on the angle that the incoming edge forms with the outgoing edge of a vertex on the border of the hull: if the visiting order is counterclockwise, the angle on the vertex always corresponds to a left turn. The following IC excludes that the angle can corresponds to a right turn:

```
2:- hull(X), cycle(From,X), cycle(X,To),
    turn_right(From,X,To).
3 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.
```

The third pruning method states that all paths starting from a node on the border of the hull cannot terminate on another vertex on the border of the hull, except for the following one on the counterclockwise order:

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

We devised several methods to compute, in ASP, the convex hull and the counterclockwise visit order of its vertices; in this work we adopt the one named JARIVS2, which proved most efficient [11].

5 TSP encodings for CASP with difference logic

One of the issues in classic ASP solvers is that the grounding may become very large; one of the possible causes of large groundings is when some decision variable ranges on a large domain. The CASP set of languages exploit the compact representation of large domains in constraint solvers to alleviate the issue of large groundings in these situations. We will consider the language of clingo[*DL*], that features a very efficient implementation of ASP extended with difference logic constraints.

The concrete syntax of a CASP clause in clingo[*DL*] is the following:

```
7 &diff{ x1 - x2 <= k }:- Body.
```

where x_1 and x_2 are two *DL* variables, k is an integer constant, and *Body* is as usual the body of the clause.

In clingo[*DL*], instead of using the #minimize statement, the most efficient way to solve an optimization problem is to identify one of the *DL* variables as the variable to minimize and pass a parameter to the command line of the solver, as follows:

```
clingo-dl program.asp –minimize-variable=var
```

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

5.1 A first DL-based encoding

In order to exploit the *DL* theory solver, one possibility is to add the code in Listing 5 to the basic TSP encoding (Listing 1); Listing 5 takes the same idea used by Rajaratnam et al. (see Listing 9 in [45]) in a warehouse delivery problem.

An arbitrary node is selected as the starting point of the travel; it is the only node for which predicate `firstpoint` is true. To each node `X` is associated a variable `visit(X)`, that intuitively represents the time in which node `X` is visited, if we interpret the costs on edges as times necessary to travel on that specific edge (of course, the costs could also be thought as distances, and in such a case `visit(X)` would represent the total length travelled when node `X` is reached).

Listing 5: Simple DL encoding

```
1 &diff{0-visit(X)}<=0:- point(X,_,_).
2 &diff{visit(X)-visit(Y)} <= -C:- cycle(X,Y),
    cost(X,Y,C), not firstpoint(Y).
3 &diff{visit(X)-totalcost} <= -C:- cycle(X,F),
    cost(X,F,C), firstpoint(F).
```

Line 1 states that the visit time of each node cannot be less than zero; in principle this constraint is necessary only for the first node, but we found more efficient to constrain the visit time of all nodes. Line 2 states that if the edge $\overline{XY}$ is in the tour, then the visit time of `Y` cannot be lower than the visit time of `X` plus the travel time of the edge $\overline{XY}$.

Line 3 introduces the objective function `totalcost` (in the following we will always assume that clingo[*DL*] is invoked with the option `–minimize-variable=totalcost`): if the last edge goes from node `X` to the (first and final) node `F`, then the `totalcost` cannot be lower than the visit time of `X` plus the travel time of edge $\overline{XF}$.

In addition to the encoding by Rajaratnam et al. [45], further clauses can be added to provide an additional speed-up when the triangular inequality holds (such as in the Euclidean TSP); the following rules do not change the set of solutions, but can be added to further reduce the domain of the `visit` variables and obtain further pruning.

Listing 6: Additional clauses to speed up the search

```
4 &diff{0-visit(X)}<= -C:- point(X,_,_), cost(F,X,C),
     firstpoint(F).
5 &diff{visit(X)-totalcost}<= -C:- point(X,_,_),
     cost(X,F,C), firstpoint(F).
```

Clause 4 states that the travelled distance when reaching a node `X` cannot be lower than going directly from the first node to `X`. Clause 5 states that the `totalcost` cannot be lower than the total distance to reach a node `X` plus the distance from `X` to the first node.

We will call Simple *DL* encoding with Triangular inequality (SDLT) the encoding including the Listings 1, 5 and 6.

5.2 A second encoding

The summation to compute the total cost introduces some inefficiencies, that cannot be avoided in clingo[*DL*] since the most efficient way to minimize an objective function is through the `–minimize-variable` directive.

Clearly, the order in which the terms are summed is immaterial, so instead of performing the sums in the order the salesperson follows to travel each edge (as in the encoding in Section 5.1), another

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

way to compute the `totalcost` is to sum the single contributions of the edges, in the order of identifier. In other words, we assume that the nodes have identifiers in an integer interval (e.g. from 1 to some maximum number n), and compute the sum of the costs of the outgoing edges

$$totalcost = \sum_{i=1}^{n} w(i, next_i)$$

if $(i, next_i)$ is an edge in the current circuit. The code in Listing 7 can be added to the basic TSP encoding (Listing 1) obtaining the summation in the new order. In such a code, the total cost is computed with the *DL* variables `total(N)`, for each node `N`; the contribution of a single edge is computed in line 9 (similarly to line 2 of Listing 5), while the `totalcost` will be greater than or equal to the total obtained in the last node (Line 10 of Listing 7).

Listing 7: A second DL encoding
```
6 &diff{total(X-1)-0}<=0:- firstpoint(X).
7 &diff{0-total(X-1)}<=0:- firstpoint(X).
8 lastpoint(X):- X = #max{ Y: point(Y,_,_) }.
9 &diff{total(N-1)-total(N)}<= -C:- point(N,_,_),
    cycle(N,B), cost(N,B,C).
10 &diff{total(N)- totalcost}<= 0:- lastpoint(N).
```

Clearly, the `total` variables no longer have the interpretation of the time in which a node is visited. The main advantage of such an encoding is that now it is easier to impose bounds on the single `total` variables, which can speed up the search by restricting the set of available values for such variables. In the example in Figure 1, we noticed the importance of imposing that the cost for exiting from a node cannot be lower than the minimal weight among the edges exiting from such node. In order to impose such bound, we can add the following clauses:

Listing 8: A second DL encoding
```
11 &diff{total(N-1)-total(N)}<= -C:- point(N,_,_),
    mincost_exit_node(N,C).
12 mincost_exit_node(N,C):- point(N,_,_), C =
    #min{Cost:cost(N,B,Cost), point(B,_,_), B!=N}.
```

We will name the encoding consisting of Listings 1, 7 and 8 *DL encoding with Exit Bound (DLEB)*.

It is also interesting to notice that the two encodings SDLT and DLEB perform orthogonal types of reasoning, and can also be used together, in order to use both the bounds (this is a common encoding technique for example in CP). On the other hand, having a dual representation could result in an overhead, so it makes sense to find experimentally if having the dual representation provides an improvement or a slowdown.

5.2.1 Propagating Excluded Edges

Beside imposing a static bound on the minimum cost for exiting a node (as done in encoding DLEB, Section 5.2), it would be important to provide a dynamic bound that takes into consideration the current choices the solver has made in a partial solution, without waiting for a full solution to be computed.

Consider again the example in Figure 1: in encoding DLEB, until the solver has inserted in the (partial) answer set one atom of the form `cycle(6,X)` (for some node `X`), rule 9 (Listing 7) cannot be activated, so the current bound for exiting from node 6 is obtained by the first rule of

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

Listing 8, namely 7 (the weight of the edge 6–4). Suppose now that in some state of the search, the solver decided to remove the possibility to go from node 6 to node 4, i.e. the solver decided that `cycle(6,4)` will not be in the answer set. In such a state, the solver might not have decided yet whether to insert in the answer set `cycle(6,5)` or `cycle(6,1)`: this will be decided in further branches of the search tree. On the other hand, since `cycle(6,4)` has been ruled out, the minimum cost for exiting from node 6 raises from 7 to 8, but the encoding DLEB is unable to find such an improved bound. In order to compute such an improved bound, the first idea would be to add a rule

```
1 &diff{total(5)-total(6)}<= -8:- not cycle(6,4).
```

Experimentally, we found that such a rule does not have good performance, probably because of the use of negation. ASP solvers are usually able to handle more effectively Horn clauses, i.e. clauses that do not have negation in the body. For this reason we decided to rewrite the choice rule 1 in Listing 1, making explicit the even loop with negation (and adding as IC the degree constraints):

Listing 9: Explicit generation of `cycle` and `nocycle` predicates

```
1 cycle(X,Y):- cost(X,Y,_), not nocycle(X,Y).
2 nocycle(X,Y):- cost(X,Y,_), not cycle(X,Y).
3:- point(X,_,_), 1 != #count{ Y: cycle(X,Y)}.
4:- point(Y,_,_), 1 != #count{ X: cycle(X,Y)}.
```

Now, it is possible to add a Horn clause having in the body `nocycle` atoms, instead of negative literals using `cycle`. Such a clause can be used to strengthen the bound when, for a node `N`, the edge connecting `N` to the closest possible node `Good` is removed:

Listing 10: Strengthening of the lower bound when the best cost is unavailable

```
1 &diff{total(N-1)-total(N)}<= -C2:-
2 point(N,_,_), mincost_exit_node(N,C), order(N,C,C2),
  nocycle(N,Good):cost(N,Good,C).
```

The conditional literal `nocycle(N,Good):cost(N,Good,C)` is necessary in case there are at least two nodes that are at the same distance from node `N`: it is true if, for all `Good` nodes that are at the minimal distance from `N`, all the edges connecting `N` to `Good` are not selected (`nocycle`). If the edges having cost `C` equal to the minimum cost have been removed, then the cost for outgoing from node `N` will be at least `C2`, if `C2` is the second-lowest possible cost for exiting from node `N`.

For example, if the closest node is at distance 3, and the second-closest node is at distance 7, when the edge to the closest node is excluded, the lower bound becomes 7.

This encoding will be named *DL* encoding with Dynamic Bound on first cost (DLDB1).

5.2.2 Generalization of the excluded edges strengthening

Clearly, it would be interesting to generalize the strengthening also when further edges (beside the one with minimum cost) are removed. For this reason, we substitute the clause in Listing 10 with the following clause:

Listing 11: Strengthening of the lower bound when the best costs are removed

```
1 &diff{total(N-1)-total(N)}<= -C2:- point(N,_,_),
    outcost(N,C), order(N,C,C2),
2 nocycle(N,Good):cost(N,Good,Cgood), Cgood<=C.
```

A ground version of the clause in Listing 11 exists for each node `N` and for each cost `C` of an edge exiting from node `N`, and again it raises the lower bound of the cost for exiting node `N` to the next possible cost `C2`.

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

In this case, the conditional literal is true if all the edges connecting `N` to a `Good` node are excluded, where a node is `Good` if it has a cost lower than or equal to a cost `C`.

We will call the encoding including the rule in Listing 11 the *DL* encoding with Dynamic Bound (DLDB).

On the other hand, such a conditional literal can be expanded, during grounding, to a long sequence of `nocycle` atoms. Considering a TSP with n nodes, for each node we would have $n-1$ outgoing edges; in the worst case they have all different weights. So, for each node, the clause in line 1 of Listing 11 is grounded to $n-1$ clauses; to give an idea of the memory occupation, the number of `nocycle` atoms occurring in the first ground clause is 1, while the last ground clause has $n-1$ `nocycle` atoms. The total number of `nocycle` atoms for each node is $O(n^2)$; considering the ground clauses of all nodes of the TSP, the memory occupation of the ground version of the clause in Listing 11 can be estimated to $O(n^3)$. Such a large memory occupation can be detrimental also for the running time of the solver.

In order to reduce the memory occupation, one can write a recursive predicate `worse_eq_than(N,C)` that is true when the cost for the edge of the TSP outgoing from node `N` cannot be lower than `C`. Such a predicate can be defined as in Listing 12, and it can be used as in line 5.

We will call the version using Listing 12 Compact *DL* encoding with Dynamic Bound (CDLDB).

Listing 12: Strengthening of the lower bound when the best value is removed

```
1 worse_eq_than(N,C2):- point(N,_,_),
    mincost_exit_node(N,C), order(N,C,C2),
    nocycle(N,Good):cost(N,Good,C).
2 worse_eq_than(N,C2):- point(N,_,_), outcost(N,C),
   order(N,C,C2),
3 worse_eq_than(N,C),
4 nocycle(N,Good):cost(N,Good,C).
5 &diff{total(N-1)-total(N)}<= -C2:-
    worse_eq_than(N,C2).
```

The number of `nocycle` atoms occurring in this version, in case the costs are all different, is only $O(n^2)$: in fact, each ground clause of predicate `worse_eq_than(N,C)` contains only one `nocycle` atom (since the costs are all different), and there is one ground clause for each node and each cost, i.e. $O(n^2)$.

6 Experimental evaluation

This section systematically compares different ASP and CASP encodings of the TSP, examining their computational efficiency and scalability in solving TSP instances. The experiments are designed to highlight the impact of encoding variations, including the introduction of geometric constraints presented in Sections 4.1 and 4.2. This exploration provides valuable insights into how specific encodings and constraints influence the practical applicability of (C)ASP in solving (Euclidean) TSP.

As an ASP solver, we chose clingo 5.6.2 [35] and as a CASP solver, we chose clingo-dl 1.5.0 [34].

To create realistic Euclidean TSP instances, we utilized the generator from the DIMACS challenge [14], which produces instances in two categories: *uniform* and *clustered*. We randomly generated

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

instances from 10 to 50 nodes, in both classes. For each size and class, we generated 16 instances, for a total of 1312 instances.

The uniform instances consist of points with integer coordinates distributed uniformly within a square with side length 10^3. Clustered instances are generated by first determining cluster centers, which are uniformly placed within a square of side 10^3. 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.

All tests were run with a time limit of 1800 s on Intel® Xeon® Platinum 8358 running at 2.6 GHz, using only one core and with 8 GB of reserved memory.

6.1 *Comparing simple and advanced ASP encodings of the TSP*

We conducted a comparison between the ASP-SE encoding presented in Section 3.2 and the ASP-PE encoding presented in Section 3.3, together with the geometric constraints (indicated as GEO) specific to the Euclidean TSP, as presented in Sections 4.1 and 4.2. The objective was to evaluate whether the geometric reasoning could provide an improvement in both encodings, and if the improvement was more significant when using a simple or advanced objective function. Figure 3a illustrates the computational time required to solve a progressively larger number of Euclidean TSP instances, with the horizontal axis showing the cumulative number of instances solved and the vertical axis indicating the CPU time in seconds. The introduction of geometric constraints significantly enhances performance: the ASP-SE + GEO encoding solves 406 instances compared to 236 instances solved by the standard ASP-SE encoding. Similarly, the ASP-PE + GEO encoding demonstrates a remarkable improvement, solving 722 instances compared to the 567 instances achieved by the ASP-PE encoding alone.

Figure 3b further compares the encodings as the problem size varies. Here, the horizontal axis represents the size of the Euclidean TSP instances, while the vertical axis indicates geometric mean of CPU time in seconds. At a problem size of 20 nodes, the ASP-SE encoding without geometric constraints times out, whereas the ASP-SE + GEO encoding solves the same instances in just a few seconds. Furthermore, while the ASP-PE encoding alone is limited to problem sizes around 45 nodes, the ASP-PE + GEO encoding manages to solve instances of this size within a few minutes. Notably, the ASP-PE + GEO encoding scales much more effectively, as evidenced by its significantly slower growth in computational time, highlighting the combined advantages of ASP-PE encoding and geometric constraints.

6.2 *Comparing DL-based encodings of the TSP*

The first comparison conducted among the Difference Logic-based encodings focused on evaluating SDLT (Section 5.1), DLEB (Section 5.2) and their combined use. Figure 4 provides a comparative analysis of the performance of these three *DL*-based encodings applied to the Euclidean TSP. Figure 4a shows CPU time as a function of the number of instances solved, with and without geometric constraints (labeled as GEO). Figure 4b presents a scatter plot comparing the CPU times for each encoding with and without geometric constraints, with the diagonal line indicating equal performance.

In Figure 4a, SDLT represented by the leftmost curve, solves fewer instances within the same time limit compared to DLEB and the combined SDLT + DLEB approach. This suggests that, while SDLT is effective, it does not handle as many instances as efficiently as the other encodings. However, when geometric constraints are introduced, all configurations show improved performance,

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

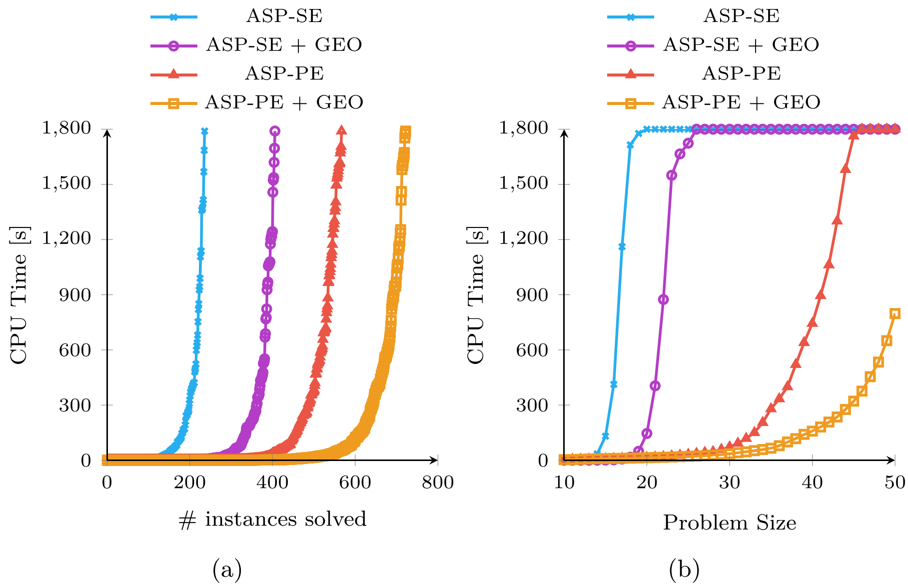

FIGURE 3. Comparison of *simple* and *advanced* ASP encodings of the TSP particularly when using geometric constraints specific to the Euclidean TSP.

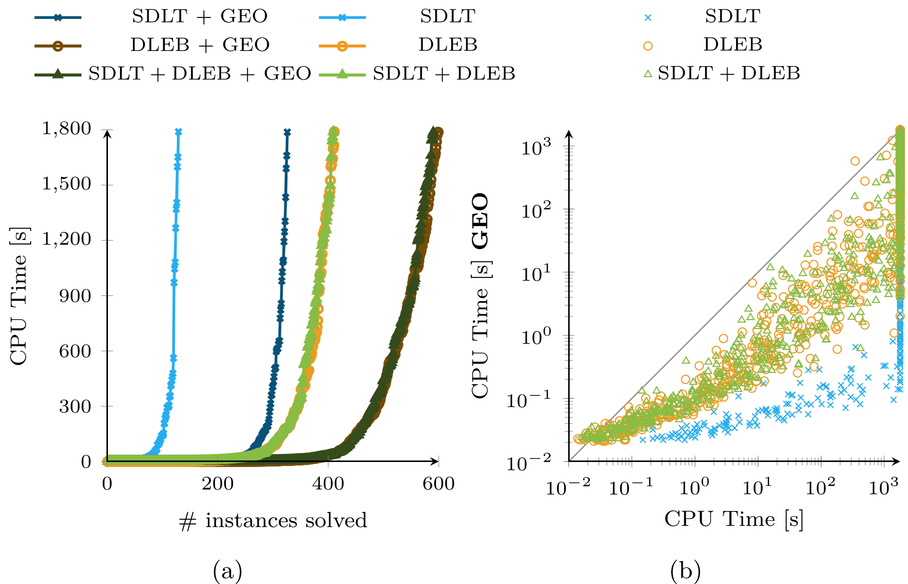

FIGURE 4. Comparison of *DL*-based encodings: SDLT (Section 5.1), DLEB (Section 5.2) and their combined use.

solving a larger number of instances within the same time limit. Figure 4b further reinforces this observation, with encodings that incorporate geometric constraints consistently outperforming their

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

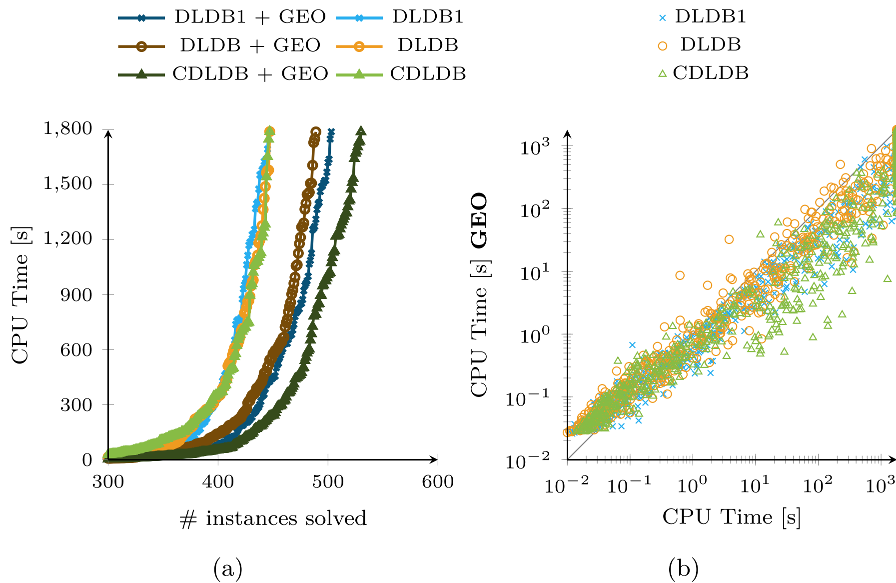

FIGURE 5. Comparison of *DL*-based encodings: DLDB1 (Section 5.2.1), DLDB and CDLDB (Section 5.2.2).

counterparts, with most points lying below the diagonal line. This improvement is particularly noticeable for SDLT which benefits most from geometric constraints but remains the least efficient encoding overall. The combined use of SDLT and DLEB does not show significant advantages, as DLEB alone appears to offer the best performance among the tested encodings.

The second comparison among the *DL*-based encodings focused on evaluating DLDB1, DLDB and CDLDB. Figure 5, as in the previous case, presents both a cactus plot and a scatter plot. The cactus plot illustrates the CPU time required to solve an increasing number of Euclidean TSP instances, allowing for a comparison of the efficiency of each encoding in terms of instances solved within a given time limit. The scatter plot, on the other hand, compares the CPU times for each encoding with and without geometric constraints, providing a direct performance comparison for individual instances. Figure 5a shows that the three encodings perform similarly overall, solving a comparable number of instances within the same time limit. However, the introduction of geometric constraints creates a noticeable difference: with these constraints, CDLDB becomes the best-performing encoding, followed by DLDB1, while DLDB proves to be the least efficient. The scatter plot in Figure 5b further highlights these differences, showing that although all three encodings benefit from geometric constraints, CDLDB gains the most, as indicated by its points lying below and farther from the diagonal line, reflecting a greater performance improvement.

The experiments reveal another notable finding, as shown in Figure 6: regardless of the use of geometric constraints, all our *DL*-based encodings outperform SDLT, which is the current state of the art for *DL*-based encodings in the literature. The plot employs a logarithmic scale, highlighting two critical observations. First, there is a substantial number of instances where SDLT times out. Second, for those instances where SDLT does not time out, all our proposed encodings solve the instances in under 2 s! To get an idea of the obtained speed up, we excluded those instances where all encodings exceed the 1800-s time limit and compared each encoding to SDLT: DLEB is 398 times

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

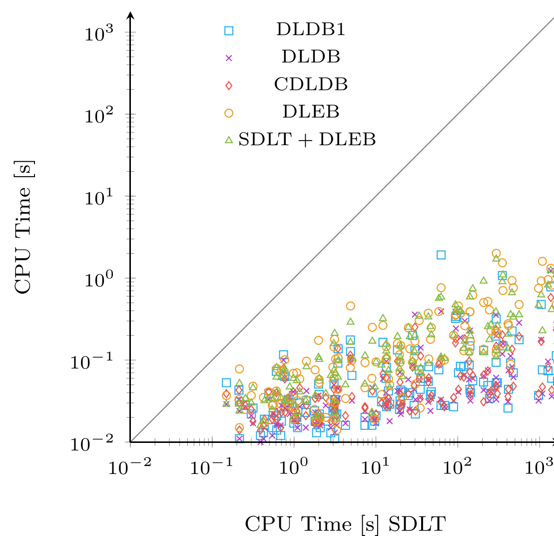

FIGURE 6. Comparison of *DL*-based encodings of the TSP without geometric constraints.

faster than SDLT, while the combined SDLT + DLEB approach is 409 times faster. The encodings DLDB1, DLDB and CDLDB demonstrate speed-ups of 2281, 2804 and 2718 times, respectively.

The final comparison focused on the best-performing encodings identified in each previous analysis. As shown in Figure 7, while CDLDB initially emerged as the most efficient *DL*-based encoding, it is notably surpassed by DLEB once geometric constraints are introduced. Thus, geometric constraints not only enhance computational efficiency across encodings, but also shift the performance hierarchy, positioning DLEB as the optimal choice among *DL*-based encodings for solving Euclidean TSP instances when such constraints are applied. The graph also highlights the strong performance of the ASP-PE encoding, particularly when combined with geometric constraints (ASP-PE + GEO).

6.3 Comparison to other CASP solvers and CLP

In this section, we present a preliminary comparison between the ASP and CASP solvers, specifically *ezcsp* [4] and *clingcon* [28]. Additionally, we provide a brief analysis of how these new results compare with CLP experimental results presented in our previous publication [11], focusing on the implications of the new findings. The experimental data used for this comparison are based on the same 1312 instances described at the beginning of this section.

To maintain consistency and avoid introducing biases through new or modified encodings, we utilized the existing encodings for the TSP presented by Yuliya Lierler in [38]. However, in our experiments we made two significant modifications compared to Lierler's experimental setup. First, in [38] Lierler compared several CASP solvers on decision problems, as some of the solvers did not support optimization problems, so she necessarily had to consider the decision version of the TSP. We recall that the decision version of the TSP aims to find any solution having a cost lower than a given threshold, and that (for a given instance), it is significantly simpler than the optimization version (the decision version is in NP, while the optimization version is not). In this work, instead, we focus on the optimization version of the TSP. The *clingcon* solver supports optimization, by

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

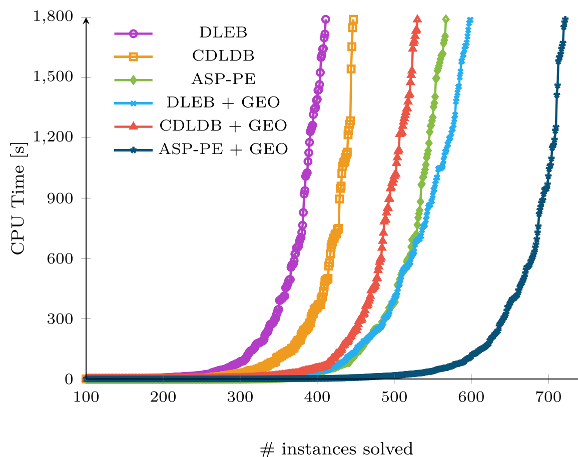

FIGURE 7. Comparison of best-performing encodings with and without geometric constraints.

adding a `&minimize` directive for the objective function. Second, since *ezcsp* does not support minimization of objective functions, we adopted the usual linear optimization approach, where the solver is called iteratively; in each iteration the threshold is set to the cost of the previous iteration. To provide a consistent reference with the previously illustrated results, we included the *clingo* solver in this comparison, even though it is not a CASP solver. For the *clingo* solver, we used the same encoding as for *clingcon*, but with the CASP-specific directives removed and the objective function implemented through the `#minimize` statement, making it equivalent to the ASP-SE encoding.

The results are plotted in Figure 8, which shows the progression of the objective function value over time for each solver. The metric used is the ratio between the best solution found and the known optimal solution for each instance, with the curves in the plot representing the average performance across all 1312 instances. Among the tested systems, *clingo* converges faster towards the optimal solution, achieving on average a solution that is 72% more expensive than the optimum, followed by *clingcon* which also performs reasonably well but maintains a higher gap from the optimal solution compared to *clingo*, with an average solution 131% more expensive than the optimal. The *ezcsp* solver shows the slowest convergence, which is likely partly due to the iterative minimization technique we had to employ due to its lack of native support for optimization directives. All the results presented so far in this paper outperform the ASP-SE encoding, which in turn already showed superior performance compared to the *clingcon* and *ezcsp* solvers. The significant performance gap observed could potentially be mitigated by developing dedicated encodings that better leverage the specific features of the *clingcon* and *ezcsp* solvers or by incorporating our geometric constraints into these systems. This will be explored in future work.

Regarding CLP experimental results, the instances and machines used in this article are the same as those in Bertagnon and Gavanelli [11], allowing a direct comparison with them. The previous study showed that CLP outperformed ASP by a significant margin. However, the new results presented in the current article indicate a substantial change in the performance hierarchy. While the CLP approach is significantly faster than the ASP approach using the basic encoding, the gap reduces when using the advanced encoding ASP-PE. Introducing geometric constraints shifts the balance, with ASP-PE + GEO outperforming the CLP encoding of the Euclidean TSP with geometric constraints. Specifically, the ASP-PE + GEO solver achieved a maximum speed-

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

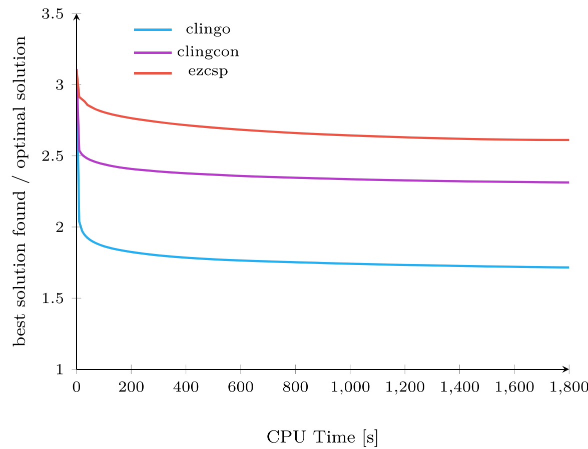

FIGURE 8. Performance comparison of ASP and CASP solvers. The plot shows the ratio between the best solution found and the known optimal solution as a function of time (in seconds). Lower values indicate better performance.

up of 3.66 and an average speed up of 1.55 over CLP with geometric constraints. These findings emphasize that the use of geometric information enhances significantly the performance of ASP solving, making even faster than CLP solutions.

7 Related work

The simple encoding of the TSP in ASP (Section 3.2) is present in various publications; e.g. [21] provides it without providing experimental results.

The TSP was also addressed in extensions of ASP, such as CASP. Lierler [38] compares experimentally several CASP systems on various benchmarks; in particular she provides an encoding of the travelling salesman problem for the CASP language EZCSP. Such an encoding leverages on the fact that in EZCSP one can impose a constraint stating that the sum of several variables must be lower (or equal to) a constant. This is not possible in the language of clingo[*DL*], in which all constraints can involve at most two variables, so the summation must be implemented with pairs of difference constraints. It would be interesting to compare the performances of the different systems.

A clingo[*DL*] encoding of a Delivery Problem is presented in [45]; the addressed problem is a multi-agent path finding problem, in which a fleet of robots transport items in a warehouse avoiding collisions. The problem includes a scheduling component, as well as a routing component. The routes are identified in a planar graph; due to the large size of the graphs, the authors opted to reduce the search space in various ways, potentially excluding the optimal solution but enabling them to find good solutions quickly.

CP was also often used to address the TSP or some of its variants. One of the most successful formulation is the so-called *successor* representation, where each node in the graph is associated with a variable, and its value indicates the following node in the TSP. Such formulation uses the `alldifferent` constraint [46] and also a `circuit` constraint, which serves to prevent subtours to occur in solutions, and for which several propagation algorithms have been proposed [6, 13, 25, 36]. Other research addressed extensions of the TSP, such as the Travelling Salesperson Problem

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

with Time Windows (TSPTW). Pesant et al. [44] addressed the TSPTW using the minimum spanning tree relaxation. Focacci et al. [23, 24] introduced reduced costs filtering to optimization constraints. Benchimol et al. [8] propose a Lagrangian relaxation scheme to perform improved reduced cost filtering in the TSP. Isoart et al. [32] introduce further pruning based on k-cutsets. Deudon et al. [17] trained a neural network to predict the most promising variable to branch on during search, based on the coordinates of the points in a Euclidean TSP.

8 Conclusion

In this article, we addressed the TSP, with emphasis on its Euclidean version in CASP.

The main contributions are two. First, we developed new encodings of the TSP in clingo[DL] [34], an efficient CASP system capable of handling difference constraints. The main advantage of clingo[DL] with respect to other CASP systems lies in its efficiency, leveraging on the fact that difference constraints can be solved in polynomial time. The main drawback stands in the expressivity, since it can handle only difference constraints, a problem that however does not affect the TSP formulations. The new encodings obtained impressive speed-ups of three orders of magnitude, enabling several instances that previously required over half an hour to be solved within seconds.

The second contribution concerns the resolution of Euclidean TSP instances. We integrated geometric reasoning, previously successful in CLP(FD) [9] and ASP solvers [11] to the CASP framework. By exploiting geometric reasoning, a further speed up was obtained in all encodings.

An extensive experimentation compared the developed encodings with improved ASP encodings found in the literature [27] and with other CASP systems. The geometric reasoning pruning synergizes with the advanced encoding [27] for the TSP: the combination of the improved encoding and the geometric reasoning makes ASP competitive with CLP solutions addressing the Euclidean TSP, while without the geometric reasoning ASP lags behind CLP solutions.

We evaluated state-of-the-art CASP systems, including EZCSP and clingcon, using the same encodings found in the literature. Our experiments showed that both systems required more solving time than the ASP solver clingo.

In future work, we plan to implement our advanced encodings for clingo[DL] and the geometric reasoning also in these systems, to assess the effectiveness of our approaches also in general CASP systems.

Moreover, we intend to evaluate our approach against other well-known TSP solvers, including Concorde [2], LKH (Lin–Kernighan–Helsgaun) [31, 39], and general-purpose MILP solvers such as Gurobi and IBM ILOG CPLEX. Solution approaches for the TSP can be divided into two categories: one is devising a constraint model and pass it to a solver, while the second is devising an imperative algorithm (which itself can also invoke a number of times a solver). Our approach is definitely declarative and falls clearly in the first category: we only define encodings in a declarative way, and the (C)ASP solver finds a solution.

The second category is definitely not declarative: an imperative algorithm has to be defined, possibly invoking a (typically exponential) number of times a solver. Concorde and LKH fall in this category. They are specifically tailored to the classical TSP and cannot accommodate side constraints, which are often essential in real-world scenarios. Additionally, LKH is a heuristic method and, although capable of producing near-optimal solutions, it cannot prove optimality.

Concerning MILP, it has to be noted that the most efficient techniques (such as Branch-and-Cut or Lagrangian Relaxation) fall again in the second category: they are involved imperative algorithms

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

that also invoke a MILP solver. Indeed, there exist also constraint models that can be passed to a MILP solver, but these are much slower than the best techniques developed in the MILP research area. The Miller-Tucker-Zemlin formulation [42] suffers from a not very effective linear relaxation, which reduces significantly the performance of the MILP solver. The Dantzig-Fulkerson-Johnson formulation [15], instead, requires an exponential number of constraints, and for this reason it is almost never used as a self-contained constraint model, but it is used within an imperative solving algorithm adding only the constraints that are strictly necessary. In contrast, declarative approaches such as CASP allow the user to focus on the high-level structure of the problem, relying on the solver to handle search and optimization.

These additional experiments will help position our approach within the broader landscape of TSP-solving methods, clarifying its advantages in terms of flexibility, generality and performance.

Acknowledgments

Alessandro Bertagnon and Marco Gavanelli are members of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

[1] M. Alviano, G. Amendola, C. Dodaro, N. Leone, M. Maratea and F. Ricca. Evaluation of disjunctive programs in wasp. In *Logic Programming and Nonmonotonic Reasoning*, M. Balduccini, Y. Lierler and S. Woltran, eds, pp. 241–255. Springer International Publishing, Cham, 2019.

[2] D. Applegate, R. E. Bixby, V. Chvátal and W. J. Cook. TSP cuts which do not conform to the template paradigm. In *Computational Combinatorial Optimization, Optimal or Provably Near-Optimal Solutions [Based on a Spring School, SchloßDagstuhl, Germany, 15–19 May 2000]*. Vol. 2241 of *Lecture Notes in Computer Science*, M. Jünger and D. Naddef, eds., pp. 261–304. Springer, 2001.

[3] S. Arora. Polynomial time approximation schemes for Euclidean TSP and other geometric problems. In *Proceedings of the 37th Conference on Foundations of Computer Science*, pp. 2–11, Association for Computing Machinery, USA, 1996.

[4] M. Balduccini and Y. Lierler. Constraint answer set solver ref4 and why integration schemas matter. *Theory and Practice of Logic Programming*, **17**, 462–515, 2017.

[5] M. Bartholomew and J. Lee. System ASPMT2SMT: Computing ASPMT theories by SMT solvers. In *Proceedings of the 14th European Conference on Logics in Artificial Intelligence, JELIA 2014, Funchal, Madeira, Portugal, September 24–26, 2014*. Vol. 8761 of *Lecture Notes in Computer Science*, E. Fermé and J. Leite, eds, pp. 529–542. Springer, Germany, 2014.

[6] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, **20**, 97–123, 1994.

[7] E. Bellodi, A. Bertagnon, M. Gavanelli and R. Zese. Improving the efficiency of Euclidean TSP solving in constraint programming by predicting effective nocrossing constraints. In *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*. Vol. 12414 of *Lecture Notes in Computer Science*, M. Baldoni and S. Bandini, eds, pp. 318–334. Springer, Germany, 2020.

[8] P. Benchimol, W. Jan, J.-C. van Hoeve, L.-M. R. Régin and M. Rueher. Improved filtering for weighted circuit constraints. *Constraints*, **17**, 205–233, 2012.

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

[9] A. Bertagnon and 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*, pp. 1412–1419. AAAI Press, 2020.

[10] A. Bertagnon and M. Gavanelli. ASPECT: answer set rePresentation as vEctor graphiCs in laTex. *Journal of Logic and Computation*, **34**, 1580–1607, 2024.

[11] A. Bertagnon and M. Gavanelli. Geometric reasoning on the euclidean traveling salesperson problem in answer set programming. *Intelligenza Artificiale*, **18**, 139–152, 2024.

[12] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca and T. Schaub. ASP-Core-2 input language format. *Theory Pract. Log. Program.*, **20**, 294–309, 2020.

[13] Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In *Logic Programming, Proceedings of the Fourteenth International Conference on Logic Programming, Leuven, Belgium, July 8–11, 1997*, L. Naish, ed., pp. 316–330. MIT Press, 1997.

[14] J. Cirasella, D. S. Johnson, L. A. McGeoch and W. Zhang. The asymmetric traveling salesman problem: algorithms, instance generators, and tests. In *Algorithm Engineering and Experimentation, Third International Workshop, ALENEX 2001, Washington, DC, USA, January 5–6, 2001, Revised Papers*. Vol. 2153 of *Lecture Notes in Computer Science*, A. L. Buchsbaum and J. Snoeyink, eds, pp. 32–59. Springer, 2001.

[15] G. Dantzig, R. Fulkerson and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, **2**, 393–410, 1954.

[16] V. G. Deineko, R. van Dal and G. Rote. The convex-hull-and-line traveling salesman problem: A solvable case. *Information Processing Letters*, **51**, 141–148, 1994.

[17] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak and L.-M. Rousseau. Learning heuristics for the TSP by policy gradient. In *Proceedings of the 15th International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR 2018, Delft, The Netherlands, June 26–29, 2018*. Vol. 10848 of *Lecture Notes in Computer Science*, W. J. Van Hoeve, ed., pp. 170–181. Springer, 2018.

[18] C. Drescher and T. Walsh. A translational approach to constraint answer set solving. *Theory and Practice of Logic Programming*, **10**, 465–480, 2010.

[19] I. Elkabani, E. Pontelli and T. C. Son. Smodels with CLP and its applications: A simple and effective approach to aggregates in ASP. In *ICLP*. Vol. 3132 of *Lecture Notes in Computer Science*, B. Demoen and V. Lifschitz, eds, pp. 73–89. Springer-Verlag, 2004.

[20] W. Faber, G. Pfeifer and N. Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, **175**, 278–298, 2011.

[21] W. Faber, F. Ricca and N. Leone. Answer set programming. In *Wiley Encyclopedia of Computer Science and Engineering*, B. Wah, ed. Wiley, Hoboken, 2008.

[22] M. M. Flood. The traveling-salesman problem. *Operations Research*, **4**, 61–75, 1956.

[23] F. Focacci, A. Lodi and M. Milano. Embedding relaxations in global constraints for solving TSP and TSPTW. *Annals of Mathematics and Artificial Intelligence*, **34**, 291–311, 2002.

[24] F. Focacci, A. Lodi and M. Milano. A hybrid exact algorithm for the TSPTW. *INFORMS Journal on Computing*, **14**, 403–417, 2002.

[25] K. G. Francis and P. J. Stuckey. Explaining circuit propagation. *Constraints*, **19**, 1–29, 2014.

[26] M. Gebser, R. Kaminski, B. Kaufmann, J. Romero and T. Schaub. Progress in ClaSP series 3. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2015, Lexington, KY, USA, September 27–30, 2015*. Vol. 9345 of *Lecture*

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

Notes in Computer Science, F. Calimeri, G. Ianni and M. Truszczynski, eds, pp. 368–383. Springer, 2015.

[27] M. Gebser, R. Kaminski, B. Kaufmann and T. Schaub. *Answer set solving in Practice*. In *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Springer International Publishing, Cham, 2012.

[28] M. Gebser, M. Ostrowski and T. Schaub. Constraint answer set solving. In *Proceedings of the 25th International Conference on Logic Programming, ICLP 2009, Pasadena, CA, USA, July 14–17, 2009*. Vol. 5649 of *Lecture Notes in Computer Science*, Patricia M. Hill and David Scott Warren, eds, pp. 235–249. Springer, 2009.

[29] M. Gebser, T. Schaub and S. Thiele. Gringo: a new grounder for answer set programming. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2007, Tempe, AZ, USA, May 15–17, 2007*. Vol 4483 of *Lecture Notes in Computer Science*, C. Baral, G. Brewka and J. S. Schlipf, eds, pp. 266–271. Springer, 2007.

[30] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP*, R. A. Kowalski and K. A. Bowen, eds, pp. 1070–1080. MIT Press, 1988.

[31] K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, **126**, 106–130, 2000.

[32] N. Isoart and J.-C. Régin. Integration of structural constraints into TSP models. In *International Conference on Principles and Practice of Constraint Programming*, T. Schiex and S. de Givry, eds, pp. 284–299. Springer, Cham, 2019.

[33] T. Janhunen, G. Liu and I. Niemelä. Tight integration of non-ground answer set programming and satisfiability modulo theories. In *Working notes of the 1st Workshop on Grounding and Transformations for Theories with Variables*, David B. Leake, ed., John Wiley & Sons, Inc., USA, 2011.

[34] T. Janhunen, R. Kaminski, M. Ostrowski, S. Schellhorn, P. Wanko and T. Schaub. Clingo goes linear constraints over reals and integers. *Theory and Practice of Logic Programming*, **17**, 872–888, 2017.

[35] R. Kaminski, J. Romero, T. Schaub and P. Wanko. How to build your own ASP-based system?! *Theory and Practice of Logic Programming*, **23**, 299–361, 2023.

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

[37] Y. Lierler. Relating constraint answer set programming languages and algorithms. *Artificial Intelligence*, **207**, 1–22, 2014.

[38] Y. Lierler. Constraint answer set programming: integrational and translational (or SMT-based) approaches. *Theory and Practice of Logic Programming*, **23**, 195–225, 2023.

[39] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, **21**, 498–516, 1973.

[40] G. Liu, T. Janhunen and I. Niemelä. Answer set programming via mixed integer programming. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning*, G. Brewka, T. Eiter and S. A. McIlraith, eds, pp. 32–42. AAAI Press, 2012.

[41] V. S. Mellarkod, M. Gelfond and Y. Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, **53**, 251–287, 2008.

[42] C. E. Miller, A. W. Tucker and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *Journal of the ACM*, **7**, 326–329, 1960.

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026

[43] J. S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. *SIAM Journal on Computing*, **28**, 1298–1309, 1999.

[44] G. Pesant, M. Gendreau, J.-Y. Potvin and J.-M. Rousseau. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, **32**, 12–29, 1998.

[45] D. Rajaratnam, T. Schaub, P. Wanko, K. Chen, S. Liu and T. C. Son. Solving an industrial-scale warehouse delivery problem with answer set programming modulo difference constraints. *Algorithms*, **16**, 216, 2023.

[46] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994*, Vol. 1, B. Hayes-Roth and R. E. Korf, eds, pp. 362–367. AAAI Press / The MIT Press, 1994.

[47] D. Shen and Y. Lierler. SMT-based constraint answer set solver EZSMT+ for non-tight programs. In *Proceedings of the Sixteenth International Conference on Principles of Knowledge Representation and Reasoning, KR 2018, Tempe, Arizona, 30 October - 2 November 2018*, M. Thielscher, F. Toni and F. Wolter, eds, pp. 67–71. AAAI Press, **2018**.

[48] J. Wittocx, M. Mariën and M. Denecker. The IDP system: a model expansion system for an extension of classical logic. In *Proceedings of Workshop on Logic and Search, Computation of Structures from Declarative Descriptions (LaSh)*, pp. 153–165. KU Leuven, Belgium, 2008.

Received 5 November 2024

Downloaded from https://academic.oup.com/logcom/article/36/2/exaf072/8442320 by dipartimento di scienze giuridiche universita degli studi di ferrara user on 27 April 2026