Improving the efficiency of Euclidean TSP solving in Constraint Programming by predicting effective nocrossing constraints*

Elena Bellodi[1] [0000−0002−3717−3779], Alessandro Bertagnon[1] [0000−0003−2390−0629], Marco Gavanelli[1] [0000−0001−7433−5899], and Riccardo Zese[1] [0000−0001−8352−6304]

DE, Ferrara University
Via G. Saragat 1
44122 Ferrara, Italy
{elena.bellodi,alessandro.bertagnon,
marco.gavanelli,riccardo.zese}@unife.it

Abstract. The Traveling Salesperson Problem (TSP) is a well-known problem addressed in the literature through various techniques, including Integer Linear Programming, Constraint Programming (CP) and Local Search. Many real life instances belong to the subclass of Euclidean TSPs, in which the nodes to be visited are associated with points in the Euclidean plane, and the distance between them is the Euclidean distance. A well-known property of the Euclidean TSP is that no crossings can exist in an optimal solution. In a previous publication, we exploited this property to speedup the solution of Euclidean instances in CP, by imposing a number of so-called no-overlapping constraints. The number of imposed constraints is quadratic in the number of nodes of the TSP. In this work, we observe that not all the no-overlapping constraints are equally useful: by experimental analysis, some of them provide a speedup, while others only introduce overhead.
We use a supervised machine learning approach on them to learn a binary classifier, with the objective to impose only those no-overlapping constraints that have been classified as effective. Preliminary experiments support the validity of the idea.

Keywords: Constraint Programming · Supervised Machine Learning · Euclidean TSP.

1 Introduction

The TSP is one of the best-known problems in computer science; given a graph in which the edges have non-negative weights (usually interpreted as traveling costs), the objective is to find a circuit visiting each node exactly once and with minimal total cost.

* Copyright ©2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
This work is partially supported by GNCS-INdAM.

The TSP is notoriously NP-hard, and it has been approached through various techniques. The most successful, to date, is Integer Linear Programming and its variants; the current state of the art is the TSP solver Concorde [2], that employs various techniques including local search, and branch-and-cut.

Concorde, however, cannot address problems in which additional constraints exist, such as the TSP with time windows (in which each of the nodes can be visited only within a given time window), or the Vehicle Routing Problem (in which more than one vehicle exists).

The TSP was also addressed in the literature of Constraint Programming, that offers more flexibility and lets one add the so-called side constraints [10, 25, 5, 14, 15, 12].

One interesting case of TSP is the Euclidean TSP, in which the nodes to be visited are associated with points in the Euclidean plane, and the distance between each pair of points is computed through the Euclidean distance. The Euclidean TSP is NP-Hard [20], but it admits Polynomial Time Approximation Schemes [3, 29]. Despite the impressive theoretical interest of these celebrated results, the usual way to address the Euclidean TSP is to convert it into a TSP by computing the distance matrix between each pair of nodes, and then use a TSP solver (e.g., Concorde) to find the optimal solution. This method completely disregards the additional information intrinsic in the Euclidean TSP formulation, such as the coordinates in the plane of the nodes.

In Constraint Programming, two methods exploit the information about the problem coordinates to speed up the solution process of the TSP. Deudon *et al.* [12] train a Deep Neural Network with the point coordinates in order to learn efficient heuristics to explore the search space.

In the other, instead, we proposed the first approach in which the information about the point coordinates was used to prune the search space of the constraint programming formulation [6]. The work started from the well-known observation that in an optimal Euclidean TSP two edges cannot cross each other, otherwise there exists another circuit with shorter length. We proposed a constraint `nocrossing` that imposes that the edges exiting from two given nodes do not cross each other. This constraint is imposed for each of the $\frac{n(n-1)}{2}$ pairs of nodes. Experimental results show the effectiveness of the approach.

As all constraints, each of them must be present in memory, can be awaken (activated) if suitable conditions occur (typically, the removal of an element from one domain), possibly performs some pruning and then becomes dormant again. When the solver wakes up a constraint, some computation time is spent in awaking, scheduling the constraint and performing the checks required by its logic. Nevertheless, the experimental results in [6] show that the additional pruning widely compensates the introduced overhead, globally.

However, although globally the set of constraints is worth imposing, it still might be the case that some of the $\frac{n(n-1)}{2}$ constraints never perform any pruning, and only introduce overhead. Stated otherwise, one wonders whether all these $\frac{n(n-1)}{2}$ constraints equally contribute to the effectiveness of the method, or whether some of these constraints perform strong pruning, while others perform little or no pruning. If one were able to guess a priori which constraints will perform pruning and which, instead, will only provide overhead, she/he could avoid imposing the useless ones, reducing the overhead associated with the set of `nocrossing` constraints, while retaining all (or, almost all) their pruning.

In this paper, we experimentally study how much pruning is performed by each of the $\frac{n(n-1)}{2}$ constraints. We label each constraint as *useful* or *useless* considering the number of times the constraint is woken and the amount of pruning it performs. We then learn a random forest binary classifier to predict which of the constraints in a new instance will be useful and which will be useless.

Preliminary experimental results show that the approach is promising.

The rest of the paper is organized as follows: after some preliminaries, we recap from [6] the basic idea of removing crossings and the declarative semantics of the `nocrossing` constraint (Section 3). Section 4 explains the data we collected running the Euclidean TSP solver on various instances. Section 5 is devoted to the Machine Learning approach to learn the classifier. In Section 6 we show the experimental results both of the classifier and of the resulting Euclidean TSP solver exploiting its predictions. Finally, Section 8 concludes the paper and provides insights into future work.

2 Preliminaries

Let $G = (V, E, w)$ be a weighted graph, where V is a set of nodes, E is a set of edges, and $w : E \mapsto \mathbb{R}^+$. A *path* is a sequence $p_{v_{s_0}\text{-}v_{s_k}} = v_{s_0} e_{s_0,s_1} v_{s_1} \dots e_{s_{k-1},s_k} v_{s_k}$ such that

1. $v_{s_0}, v_{s_1}, \dots, v_{s_k} \in V$ and are all distinct, and
2. $e_{s_0,s_1}, \dots, e_{s_{k-1},s_k} \in E$.

Since a path is uniquely identified by the sequence of its nodes (or of its edges) in the proper order, to simplify the notation we will often write paths as sequences of nodes. The length of a path p is the sum of the weights of its edges: $L(p) = \sum_{i=0}^{k-1} w(e_{s_i,s_{i+1}})$. Given a path $p_{v_{s_0}\text{-}v_{s_k}}$, the sequence obtained by appending e_{s_k,s_0} to a path $p_{v_{s_0}\text{-}v_{s_k}}$ is also called a *circuit* c.

The Metric TSP is a TSP in which there is an edge connecting each pair of nodes, and the distance function w enjoys the triangular inequality: $w(e_{a,c}) \leq w(e_{a,b}) + w(e_{b,c})$.

A Euclidean TSP is a Metric TSP in which the distance function is the Euclidean distance. Let $\mathcal{P} = \{P_1, \dots, P_n\}$ be a set of points, where $P_i = (x_i, y_i)$. The graph associated with $\mathcal{P}$ is $G^{\mathcal{P}} = (\mathcal{P}, E^{\mathcal{P}}, w^{\mathcal{P}})$, where $E^{\mathcal{P}} = \{e_{i,j} \equiv (P_i, P_j) \mid P_i, P_j \in \mathcal{P}, i \neq j\}$ and $w(P_i, P_j) = d(P_i, P_j)$, where d is the Euclidean distance.

In the Constraint Programming literature, three main constraint models have been proposed to address the TSP: the *permutation* representation, the *successor* representation and the *set variable* representation [5].

In this paper we adopt the successor representation; it is defined with a set of n variables *Next*, each ranging on the set of available nodes. $\mathit{Next}_i = j$ means that the successor of node i is node j. The constraint model includes

- an `alldifferent`$(Next)$ constraint [32], that imposes that each node is the successor of exactly one other node,
- a `circuit`$(Next)$ constraint [10] that excludes the creation of sub-circuits, i.e., circuits that do not involve the whole set of nodes,
- and an objective function aimed at minimizing the total length of the TSP.

3 Avoiding crossings

The following is a well-known result in the literature

Theorem 1. *[16]. Let c^* be an optimal tour of a Metric TSP. Then, for each $e_{i,j}, e_{k,l} \in c^*$ such that $\{i, j, k, l\}$ are all different, the segments $\overline{P_iP_j} \cap \overline{P_kP_l} = \emptyset$.*

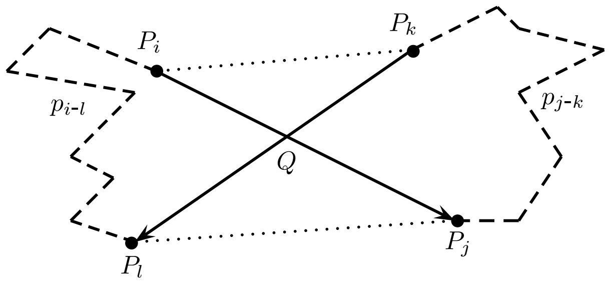

Fig. 1. A self-crossing circuit.

Proof. (sketch). In Fig. 1, instead of taking $\overline{P_iP_j}$ and $\overline{P_kP_l}$, a shorter tour chooses the dotted edges $\overline{P_iP_k}$ and $\overline{P_lP_j}$.

In order to speedup the search, in [6], we proposed a constraint that avoids, during search, the solutions that include crossings. The `nocrossing` constraint

$$\texttt{nocrossing}(i, Next_i, j, Next_j)$$

imposes that the segment $\overline{P_iP_{Next_i}}$ and the segment $\overline{P_jP_{Next_j}}$ do not intersect, or intersect at most in one of the extremes P_i or P_j.

Clearly, this constraint should be imposed for each pair of nodes, i.e. a quadratic number of constraints.

Notice that these constraints are aimed only at improving the efficiency of the solution, they are not necessary for its correctness; they are in fact redundant constraints, and it is well known in the CP literature that redundant constraints can improve the efficiency of the search.

One question could be whether all these constraints perform effective pruning, reducing the search space, or whether only some of them are actually useful, while others do not perform any significant pruning while introducing overhead. In next section, we try to reply to this question.

4 The collected data

To evaluate the performance of each constraint we collected data while solving Euclidean TSP instances. In order to have a statistically significant number of instances, we decided to use randomly-generated ones. We used the generator of the DIMACS challenge [24], which provides two-dimensional instances in TSPLIB format consisting of integer-coordinate points. The generated instances can belong to two different classes: *uniform* and *clustered*. Points of the instances belonging to the *uniform* class are uniformly distributed in a $10^6 \times 10^6$ square, while points belonging to the *clustered* class are located in clusters that are uniformly distributed in a $10^6 \times 10^6$ square. We randomly generated instances from 15 to 30 nodes, in both classes. For each size and class we generated 32 instances, for a total of 1024 instances.

For each `nocrossing` constraint, in each instance, we measured 3 indicators: the number of activations $N_{activations}$, the number of value deletions N_{pruned} from the domain of the variables involved, and the number of failures (and therefore backtracks) generated as a result of the deletion of values. The first two indicators were then combined to obtain a fourth one, denoted as `RTIO` and calculated as the ratio $\frac{N_{pruned}}{N_{activations}}$. A constraint with a low `RTIO` wakes up many times without being able to perform pruning so it produces an unwanted overhead, while a constraint with a high `RTIO` can perform a much stronger pruning compared to the number of activations and therefore it is worth imposing it. Figures 2, 3 and 4 graphically show the number of value deletions, the number of failures and the `RTIO` respectively, for a typical instance of Euclidean TSP. In each figure, the darker the color of the line, the higher the value of the corresponding indicator. Figure 5 furthermore illustrates the `nocrossing` constraints that have not performed any pruning ($N_{pruned} = 0$). We produced many of these figures in the hope to find some significant pattern, that could help us identify the useful or the useless instances of `nocrossing` constraints; unluckily we were not able to observe any interesting pattern. We decided then to introduce a machine learning step.

Each constraint was labelled by means of the `RTIO`, which can be seen as an indicator of the "goodness" of a constraint. A constraint belonging to a certain instance $\mathcal{I}$ is labeled as *useful* if its `RTIO` is greater than the arithmetic mean calculated on the `RTIO` of all the constraints belonging to instance $\mathcal{I}$, otherwise it is labeled as *useless*. The relation we wish to learn could be seen as a function mapping each pair of points (in a generic Euclidean TSP instance) to the set $\{useful, useless\}$. In principle, each point could be identified solely by its coordinates, but this could be a too specific information: the effectiveness of a constraint should be independent from rigid transformations of the whole set of points, such as rotations, axial symmetries, or even scaling.

For this reason, beside the (normalized) coordinates of each point, we computed a set of features trying to synthesize some further information that is invariant with respect to these transformations. As a guidance, we chose some features reflecting information exploited by effective TSP solving algorithms,

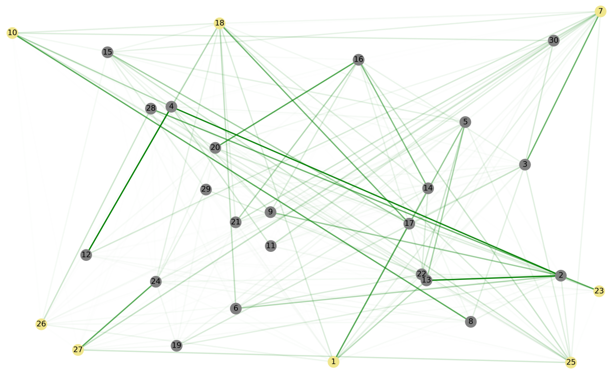

Fig. 2. Graphical representation of the number of value deletions performed by each `nocrossing` constraint in a Euclidean TSP instance. The darker the color of the segment, the higher the number of value deletions.

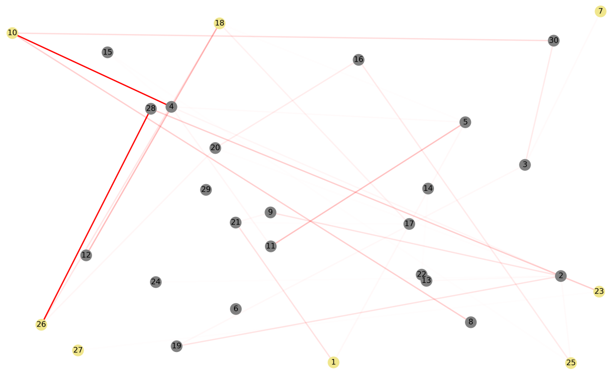

Fig. 3. Graphical representation of the number of failures generated by each `nocrossing` constraint in a Euclidean TSP instance. The darker the color of the segment, the higher the number of failures.

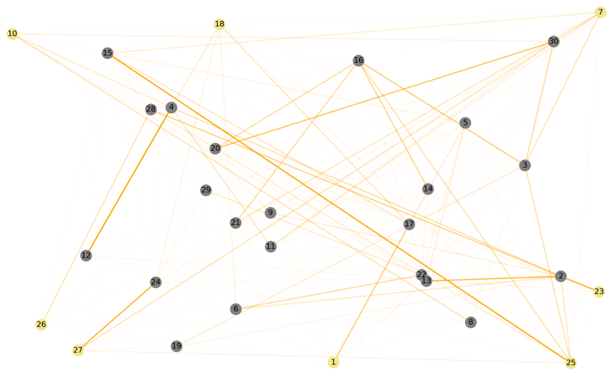

Fig. 4. Graphical representation of the `RTIO` of each `nocrossing` constraint in a Euclidean TSP instance. The darker the color of the line, the higher the `RTIO` value.

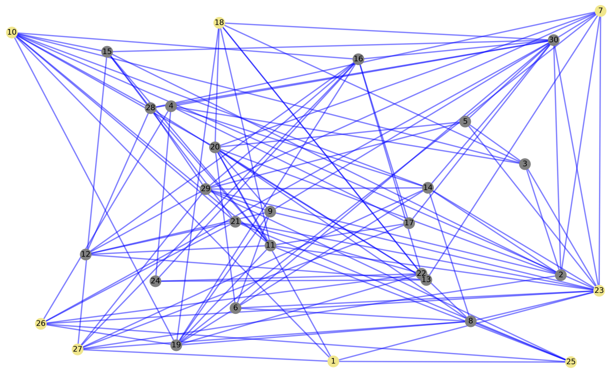

Fig. 5. Graphical representation of the `nocrossing` constraints that did not delete any value in a Euclidean TSP instance. A line between two points indicates that the `nocrossing` constraint applied on that pair of points was unable to perform any pruning. These constraints only introduce overhead.

in the hope that they could also serve as guidance for the effectiveness of the `nocrossing` constraints.

One effective algorithm for solving TSPs is the Held and Karp procedure [22], based on spanning trees. The minimum spanning tree of the set of nodes can be computed in polynomial time, and is a popular valid lower bound on the value of the optimal solution. One of the properties we chose for a pair of points is whether the segment connecting them belongs to a minimum spanning tree.

Another interesting property is the so-called necklace condition [13]. Suppose to find a set of discs, each centered on one of the points to be visited, such that the interiors of two discs do not intersect. Clearly, an optimal tour should enter and exit each of the discs, so a valid lower bound is twice the sum of the radii of the discs. From this observation, another interesting property could be the distance of each node to the closest other node.

Finally, in [6] we also introduced constraints that performed pruning based on the convex Hull of the set of points; that pruning was also extended to the case of *interior Hulls*, after (during search) some of the segments in the current path was already fixed.

Considering what has been introduced so far, we have identified the following 15 features for each `nocrossing`$(i, Next_i, j, Next_j)$ constraint in the dataset:

- `XPIN`, `YPIN`, `XPJN`, `YPJN`: normalized coordinates of the two points P_i and P_j involved in the constraint. These coordinates are obtained, starting from the original coordinates of the points, by repositioning each instance within a square space with a 100 unit long side;
- `DISN`: Euclidean distance calculated between points P_i and P_j using normalized coordinates;
- `LEVI`, `LEVJ`: level of points P_i and P_j. The idea is to distinguish the points on the perimeter of the convex hull from the internal ones, and have a numeric

value suggesting how deep in the interior of the figure is each point. The level of a point P is defined inductively with respect to the set $\mathcal{P}$ of all the points:

$$lev(P) = lev_{\mathcal{P}}(P).$$

The level of a point P with respect to a set $\mathcal{X}$ is 1 if P belongs to the *"exterior"* of $\mathcal{X}$ (precisely, the perimeter $Hull_{\mathcal{X}}$ of the convex hull of $\mathcal{X}$) and is defined inductively as 1 plus the level of P on the *"interior"* set $\mathcal{X} \setminus Hull_{\mathcal{X}}$ otherwise:

$$lev_{\mathcal{X}}(P) = \begin{cases} 1 & \text{if } P \in Hull_{\mathcal{X}} \\ 1 + lev_{\mathcal{X} \setminus Hull_{\mathcal{X}}}(P) & \text{otherwise} \end{cases}$$

– `CXNI`, `CYNI`, `CXNJ`, `CYNJ`: normalized coordinates of the nearest point to P_i and P_j respectively;
– `NDTI`, `NDTJ`: Euclidean distance of the closest point to P_i and P_j respectively, calculated using normalized coordinates;
– `NBHD`: number of points contained in the circle having as diameter the segment connecting points P_i and P_j;
– `NMST`: (boolean value) indicates whether the segment $\overline{P_iP_j}$ belongs to a minimum spanning tree.

5 Machine learning the goodness of constraint propagators

The dataset of labelled constraints is suitable for the application of supervised machine learning algorithms, with the goal of learning a model able to predict which of the constraints will be useful and which will be useless in a new unseen instance of the Euclidean TSP. As each constraint is labelled as useful or useless, a *binary* classifier can be learnt that discriminates between the negative class (*useless*, represented by a label equal to 0) and the positive class (*useful*, represented by a label equal to 1).

Among the classification techniques, the Random Forest (RF) approach is known for its good computational performance and scalability. A random forest is a meta estimator that fits a number of decision tree classifiers on various subsamples of the dataset and uses averaging to improve the predictive accuracy and control overfitting [9]. The algorithm used in the experimental validation is the one available in the WEKA[1] workbench for machine learning. Given a training set $\mathbf{X}$ of instances of Euclidean TSP, learning a random forest in WEKA involves the following steps [1]:

1. Bootstrap samples $\mathbf{B}_i$ for every tree t_i are drawn by randomly selecting pairs of points (the examples) with replacement from $\mathbf{X}$ until the sizes of $\mathbf{B}_i$ and $\mathbf{X}$ are equal;

[1] https://www.cs.waikato.ac.nz/ml/weka/

2. A random subset of features (attributes) are selected for each $\mathbf{B}_i$ and used for the training of tree t_i in the forest;
3. An information gain metric is used to grow unpruned decision trees;
4. The final classification result is the most popular of the individual tree predictions.

The RF classifier can now be used to predict if on new unseen pairs of points the `nocrossing` constraint should be imposed, at the same time indicating to avoid imposing constraints that have been assigned the negative class (useless). The whole proposed procedure is the following:

1. A training dataset of various Euclidean TSP instances, where each constraint has been labelled according to its own `RTIO`, is used to learn a RF classifier;
2. Given any new instance of the problem, for whose constraints the three indicators (and so the class) are unknown, apply the classifier to find pairs of points that are classified as *useful* for imposing the `nocrossing` constraint;
3. Run the instance together with the selected `nocrossing` constraints to solve the Euclidean TSP.

With step 3, we try to eliminate the temporal overhead that might be introduced during the search for a solution by the constraints recognized as not effective by the random forest model.

An advantage of this procedure is that, once the classifier has been learnt, it can be reused on any new instance of the problem, without re-performing the machine learning step, which is executed only once.

6 Experiments

6.1 Results of the Machine Learning step

The machine learning task was carried out by training the RF classifier implemented in WEKA (version 3.8.4) [34]. The training phase was controlled by the following parameters:

– `-P 100`: size of each bag, as a percentage of the training set size; the default value of 100 was kept;
– `-I 100`: number of iterations (i.e., the number of trees in the random forest);
– `-num-slots 1`: number of execution slots (thread) for constructing the ensemble. The default 1 means no parallelism;
– `-K 0`: sets the number of randomly chosen attributes;
– `-M 1`: the minimum number of instances per leaf;
– `-V 0.001`: minimum numeric class variance proportion of train variance for split (it was kept the default value);
– `-S 1`: seed for random number generator (it was kept the default value).

Given a dataset of 517,120 `nocrossing` constraints, collected from 1024 instances of Euclidean TSP in order to investigate the performance of the classifier, data were split into 66% for training and the remainder for test, obtaining the performance summary shown in Table 1 and an accuracy of 0.908. Recalling that each pair of points can be referred to as an 'example', the TP Rate is the fraction of true positives (TP) over the total positive examples, the FP Rate is the fraction of false positives (FP) over the total negative examples, Precision is $TP/(TP + FP)$, Recall corresponds to the TP Rate, F-measure is the harmonic mean of Precision and Recall, ROC and PR Area represent the areas under the ROC and PR curves taking values in the range [0,1]. The ROC curve plots the TP Rate versus the FP Rate, the PR curve plots the Precision versus the Recall [11, 31]. Accuracy is the number of correctly classified examples over all examples. The highest possible value for all metrics is 1, and, except for the FP Rate, the highest the value, the better the performance.

The classifier shows a very good performance over the test `nocrossing` constraints especially for the negative class (useless). After this preliminary test, the entire dataset was used for training the RF classifier, in order to feed the learning algorithm with more data, taking 544.41s to complete.

Table 1. Performance of the RF classifier over the fraction of instances used for test.

Class	TP Rate (Recall)	FP Rate	Precision	F-Measure	ROC Area	PR Area
0	0.957	0.301	0.932	0.944	0.945	0.984
1	0.699	0.043	0.790	0.742	0.945	0.818

6.2 Results of the overall Euclidean TSP solver

In order to evaluate the improvements in solving time obtained thanks to the predictions made by in the machine learning step, we devised a series of experiments based on randomly-generated TSPs. As for the data collection phase described in Section 4 we used the generator of the DIMACS challenge [24]. We generated a total of 1024 test instances (different from the ones used for learning) varying the size from 15 to 30 nodes, in both *uniform* and *clustered* classes (32 instances for each size and class).

We compared three constraint models based on the successor representation as introduced in Section 2:

– the basic constraint model described in the preliminaries (denoted as `ECLP`), including the `circuit` and `alldifferent` constraints required by the successor representation plus the objective function;
– the constraint model imposing the `nocrossing` constraint for all pairs of nodes (denoted as `ALL`);
– the model imposing only the `nocrossing` constraints predicted as useful by the RF classifier (denoted as `PRED`).

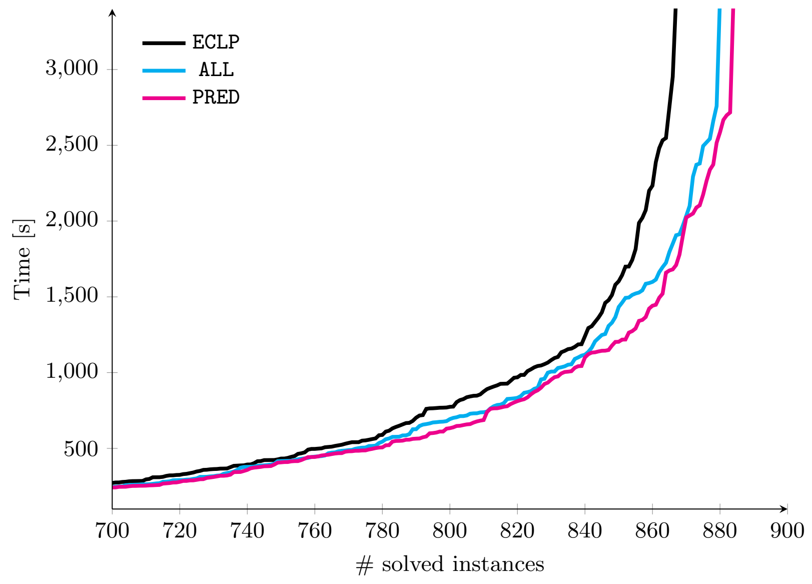

Fig. 6. Solving time as a function of the number of solved instances. Different curves correspond to the different models that were compared during the experiments.

All experiments use the *max regret* search strategy [10].

All algorithms are implemented in the ECL^{i}PS^e CLP language [33]. All tests were run on ECL^{i}PS^e v. 7.0, build #54, on Intel® Xeon® E5-2630 v3 CPUs running at 2.4GHz, with one core and with 1GB of reserved memory. The time limit for each run was set to 3480s.

Figure 6 summarizes the results, in particular it shows how many of the instances can be solved by each of the three constraint models when given a time limit reported in the y-axis. The constraint model that includes only the `nocrossing` constraints predicted to be useful is the one obtaining the best results overall, confirming the effectiveness of the approach proposed in this paper. The selection of constraints also increased slightly the number of instances that could be solved: among the 1024 tested instances, `ECLP` incurred in timeout on 158 instances, while `ALL` on 145, and `PRED` on 141. Randomly-generated TSPs instances, collected runtime data and datasets are available online[2].

7 Related work

There exists a wide literature on combinations of constraint programming with machine learning or data mining [8].

One of the main ideas is portfolio selection (see, e.g., the survey [26] and references therein): given a set of algorithms (or solvers) that solve a same problem,

[2] https://github.com/abertagnon/rcra-2020

select the best one for solving a given instance. The approach is based on obtaining data about the running time of the algorithms by collecting a high number of instances and running each of the available algorithm on each instance. Also, for each instance a number of features are computed, hopefully synthesizing the most important characteristics of the instance that make it easy or hard to solve. These can be simple statistical measures, including the size of the instance, up to measures from the literature, such as the treewidth of the instance. After that, a classifier is learned trying to predict, given the set of features of a new unseen instance, which of the available solvers will be the fastest for that specific instance. Once a new instance is provided, the classifier is used to choose the best solver for solving it.

Our work could, in principle, be seen as an algorithm portfolio in which the classifier chooses among an exponential number of solving algorithms, each imposing a subset of the set of possible `nocrossing` constraints. However, we do not learn such a complex classifier (although in principle it could provide better results) because the number of possible solvers would be too large, but we try to predict which of the single `nocrossing` constraints will be effective.

Although less strictly related to this paper, we cite other approaches to combine Machine Learning and CP, including Empirical Model Learning [28, 27], trying to learn some features of a physical system and including its input/output relation as a new constraint, or approaches that try to learn single constraints or a whole constraint model given examples from the user [7, 4].

Various works in the CP literature address the TSP or some of its variants. Some of them propose implementations of the `circuit` constraint [10, 25, 19, 14]: considering that the Hamiltonian Circuit Problem is NP-complete, obtaining Arc-Consistency for the `circuit` constraint would be NP-complete, so the works in the literature forego the idea of achieving Arc-Consistency, and propose instead different tradeoffs between computation time and pruning power.

Considering the objective function, various works exploit relaxations (as usually done in Integer Linear Programming) to quickly rule out non-promising solutions; the usual relaxations are the minimum spanning tree [30, 17, 18] and the assignment problem relaxations [17, 18].

Starting with Benchimol *et al.* [5], various works propose to integrate the objective function into a constraint that ensures Hamiltonianicity of the graph [15, 23, 12] in order to achieve stronger pruning based on the current upper bound.

8 Conclusions and Future work

In this paper, we proposed to predict and select the set of `nocrossing` constraints in Euclidean TSPs formulated in CP.

The first experimental results are encouraging, but more extensive experiments will also be carried out. The performance of the solver with constraints pruned by the RF classifier will be compared with the same number of constraints but pruned at random, to validate the relevance of the proposed approach. We also believe that much more efficiency could be obtained in future research.

One first direction is refining the RF model to check if better performance may be obtained, but also testing other supervised learning techniques. For instance it would be interesting to predict, instead of two classes, the actual ratio of pruning power versus activations $\frac{N_{pruned}}{N_{activations}}$, run experiments imposing only those `nocrossing` constraints whose ratio is predicted to be higher than a given threshold, and experimentally find the best possible value of the threshold. This changes the machine learning step from a classification to a regression task, which, on the other hand, may be more challenging. One could then move from supervised learning to reinforcement learning techniques and thus avoid collecting a training set.

Some improvement could be obtained by expanding the dataset of experiments, i.e. by running experiments in more instances to widen the available data about number of activations and pruning of the `nocrossing` constraints. Also, in the current dataset only one search heuristic was employed, namely *max regret* [10]. Since max regret is a dynamic search heuristics, it might be the case that changing the set of `nocrossing` constraint, the search strategies change radically the exploration of the search tree, possibly shuffling the order in which `nocrossing` constraints are activated, and making effective some constraints that were not and vice-versa. So, a more precise classifier could be obtained by generating datasets with different search strategies.

Another method could be to use *data augmentation* techniques; for example by rotating all points in an instance of a same angle, one obtains the same TSP instance (and the optimal solution does not change), so in principle the number of activations should not change, while the input data would be different from the viewpoint of the machine learning algorithm, since the point coordinates would change.

An enlargement of the set of features, collected during the creation of the dataset, is already planned, so that the learned classifier relies less on the point coordinates and more on the synthesized features (e.g. number of edges crossed by the segment $\overline{P_i P_j}$ related to the constraint `nocrossing`$(i, Next_i, j, Next_j)$).

Finally, instead of learning a classifier that chooses the set of `nocrossing` constraints that should be imposed a priori, one could use more dynamic strategies, like removing during search the `nocrossing` constraints that result less effective because they have not obtained significant pruning in the last activations. One source of inspiration could be the strategies used in SAT solvers to forget some of the nogoods [21].

References

1. Amrehn, M., Mualla, F., Angelopoulou, E., Steidl, S., Maier, A.: The random forest classifier in weka: Discussion and new developments for imbalanced data (2019)
2. Applegate, D., Bixby, R.E., Chvátal, V., Cook, W.J.: TSP cuts which do not conform to the template paradigm. In: Jünger, M., Naddef, D. (eds.) Computational Combinatorial Optimization, Optimal or Provably Near-Optimal Solutions [based on a Spring School, Schloß Dagstuhl, Germany, 15-19 May 2000]. Lecture Notes in Computer Science, vol. 2241, pp. 261–304. Springer (2001)

3. Arora, S.: Polynomial time approximation schemes for euclidean TSP and other geometric problems. In: Proceedings of 37th Conference on Foundations of Computer Science. pp. 2–11 (Oct 1996)
4. Beldiceanu, N., Simonis, H.: Modelseeker: Extracting global constraint models from positive examples. In: Bessiere et al. [8], pp. 77–95. https://doi.org/10.1007/978-3-319-50137-6
5. Benchimol, P., van Hoeve, W.J., Régin, J., Rousseau, L., Rueher, M.: Improved filtering for weighted circuit constraints. Constraints **17**(3), 205–233 (2012)
6. Bertagnon, A., Gavanelli, M.: 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), https://aaai.org/ojs/index.php/AAAI/article/view/5498
7. Bessiere, C., Daoudi, A., Hebrard, E., Katsirelos, G., Lazaar, N., Mechqrane, Y., Narodytska, N., Quimper, C., Walsh, T.: New approaches to constraint acquisition. In: Bessiere et al. [8], pp. 51–76. https://doi.org/10.1007/978-3-319-50137-6
8. Bessiere, C., De Raedt, L., Kotthoff, L., Nijssen, S., O'Sullivan, B., Pedreschi, D. (eds.): Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach, Lecture Notes in Computer Science, vol. 10101. Springer (2016). https://doi.org/10.1007/978-3-319-50137-6
9. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (Oct 2001). https://doi.org/10.1023/A:1010933404324
10. Caseau, Y., Laburthe, F.: Solving small TSPs with constraints. In: Naish, L. (ed.) Logic Programming, Proceedings of the Fourteenth International Conference on Logic Programming, Leuven, Belgium, July 8-11, 1997. pp. 316–330. MIT Press (1997)
11. Davis, J., Goadrich, M.: The relationship between precision-recall and ROC curves. In: European Conference on Machine Learning (ECML 2006). pp. 233–240. ACM (2006)
12. Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., Rousseau, L.: Learning heuristics for the TSP by policy gradient. In: van Hoeve, W.J. (ed.) Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26-29, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10848, pp. 170–181. Springer (2018)
13. Edelsbrunner, H., Rote, G., Welzl, E.: Testing the necklace condition for shortest tours and optimal factors in the plane. Theor. Comput. Sci. **66**(2), 157–180 (1989). https://doi.org/10.1016/0304-3975(89)90133-3
14. Fages, J., Lorca, X.: Improving the asymmetric TSP by considering graph structure. CoRR **abs/1206.3437** (2012), http://arxiv.org/abs/1206.3437
15. Fages, J., Lorca, X., Rousseau, L.: The salesman and the tree: the importance of search in CP. Constraints **21**(2), 145–162 (2016)
16. Flood, M.M.: The traveling-salesman problem. Operations Research **4** (1956)
17. Focacci, F., Lodi, A., Milano, M.: Embedding relaxations in global constraints for solving TSP and TSPTW. Ann. Math. Artif. Intell. **34**(4), 291–311 (2002)
18. Focacci, F., Lodi, A., Milano, M.: A hybrid exact algorithm for the TSPTW. INFORMS Journal on Computing **14**(4), 403–417 (2002)
19. Francis, K.G., Stuckey, P.J.: Explaining circuit propagation. Constraints **19**(1), 1–29 (2014). https://doi.org/10.1007/s10601-013-9148-0

20. Garey, M.R., Graham, R.L., Johnson, D.S.: Some NP-complete geometric problems. In: Proceedings of the Eighth Annual ACM Symposium on Theory of Computing. pp. 10–22. STOC '76, ACM, New York, NY, USA (1976)
21. Gent, I.P., Miguel, I., Moore, N.C.A.: An empirical study of learning and forgetting constraints. AI Commun. **25**(2), 191–208 (2012). https://doi.org/10.3233/AIC-2012-0524
22. Held, M., Karp, R.M.: The traveling-salesman problem and minimum spanning trees. Operations Research **18**, 1138–1162 (1970)
23. Isoart, N., Régin, J.C.: Integration of structural constraints into tsp models. In: International Conference on Principles and Practice of Constraint Programming. pp. 284–299. Springer (2019)
24. Johnson, D.S., McGeoch, L.A.: Experimental analysis of heuristics for the stsp. In: The traveling salesman problem and its variations, pp. 369–443. Springer (2007)
25. Kaya, L.G., Hooker, J.N.: A filter for the circuit constraint. In: Benhamou, F. (ed.) Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4204, pp. 706–710. Springer (2006)
26. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. In: Bessiere et al. [8], pp. 149–190. https://doi.org/10.1007/978-3-319-50137-6
27. Lombardi, M., Milano, M.: Boosting combinatorial problem modeling with machine learning. In: Lang, J. (ed.) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. pp. 5472–5478. ijcai.org (2018). https://doi.org/10.24963/ijcai.2018/772
28. Lombardi, M., Milano, M., Bartolini, A.: Empirical decision model learning. Artif. Intell. **244**, 343–367 (2017). https://doi.org/10.1016/j.artint.2016.01.005
29. Mitchell, J.S.B.: Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. SIAM J. Comput. **28**(4), 1298–1309 (1999)
30. Pesant, G., Gendreau, M., Potvin, J., Rousseau, J.: An exact constraint logic programming algorithm for the traveling salesman problem with time windows. Transportation Science **32**(1), 12–29 (1998)
31. Provost, F.J., Fawcett, T.: Robust classification for imprecise environments. Machine Learning **42**(3), 203–231 (2001)
32. Régin, J.: A filtering algorithm for constraints of difference in CSPs. In: Hayes-Roth, B., Korf, R.E. (eds.) Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1. pp. 362–367. AAAI Press / The MIT Press (1994)
33. Schimpf, J., Shen, K.: Ecl$^{\mathrm{i}}$ps$^{\mathrm{e}}$ - from LP to CLP. TPLP **12**(1-2), 127–156 (2012)
34. Witten, I.H., Frank, E., Hall, M.A.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann, 3 edn. (2011)