Machine-learning-enhanced strategies to generate subtour elimination constraints for the symmetric traveling salesman problem

Fatih Burak Akçay, Maxence Delorme, Ulrich Pferschy

(1) Department of Econometrics and Operations Research, Tilburg University, The Netherlands
(2) Department of Operations and Information Systems, University of Graz, Austria
Corresponding author m.delorme@tilburguniversity.edu

Abstract

We present a machine learning (ML) component designed to enhance the well-known branch-and-cut (B&C) framework for the symmetric traveling salesman problem (TSP) in which only the subtour elimination constraints (SECs) violated by previously found integer solutions are considered. The objective of the ML component is to identify which SECs, from a pool of candidates, will be required during the B&C execution and to add these constraints before initializing the framework. To develop this component, we investigate several intermediate questions, including evaluating various B&C implementations to identify the most effective one, determining the theoretical minimum and maximum number of SECs required by a given B&C implementation, generating a pool of SEC candidates, creating labeled data, engineering SEC-specific input features, and training the ML agent. We demonstrate that our trained agent slightly increases the number of instances solvable by our best B&C implementation and reduces the average computation time for these instances by 23%. We also show that these improvements extend to TSP instances with features different from those used to train the agent, although not consistently. While these improvements are already noteworthy, we also provide a proof of concept showing that reductions of up to 65% could be achieved with a "perfect" component, highlighting the potential of ML-integrated agents within exact optimization approaches. Although we acknowledge that, even when ML-augmented, the resulting B&C implementation still performs below the state-of-the-art Concorde algorithm, the proposed methodology is generalizable to other routing problems with SECs.

Keywords: Travelling salesman; Branch-and-cut; Machine learning; Exact algorithms; Integer programming.

1 Introduction

Given a complete graph where each edge is associated with a non-negative length, the *symmetric traveling salesman problem* (TSP) requires finding a Hamiltonian cycle, i.e., a tour that visits every vertex exactly once, with the smallest total length. While the origin of the TSP can be traced back to the first half of the 19th century [3], the problem has continuously inspired the research community to this day, as witnessed by the numerous books, articles, and codes published on the topic, making it one of the most studied combinatorial optimization problems of all time. Indeed, the relatively simple structure of this $\mathcal{NP}$-hard problem [21] made it the perfect playground to develop and test all sorts of solution methods, including heuristics [6], metaheuristics [15], branch-and-bound algorithms [25], branch-and-cut algorithms [16], integer linear programming models [14], and deep reinforcement learning [13]. Besides theoretical motivations, the TSP is also studied for its numerous

direct practical applications from widely diverse fields, such as school bus routing, printed circuit board drilling, and genome sequencing [3].

For the last twenty years, the well-known *Concorde* optimization package [9] has been recognized as state-of-the-art for the TSP, a position that is likely to remain uncontested for at least a few more years. Whereas the source code of Concorde is freely available to any academic researcher, and the introduction of various "wrappers" has made the software accessible even to the less tech-savvy users, one must acknowledge that adapting it to solve one of the many close relatives of the TSP that frequently appear in routing problems remains a daunting task: navigating through 17 folders, 153 files, and 96 788 lines of code (counted using the `cloc` software on the `Concorde-03.12.19` distribution, restricted to source files `.c` and `.h`) spread over 700 functions [9], requires both significant expertise and a substantial investment of time. On the other hand, solution methods relying on *integer linear programming* (ILP) models have increased in popularity in the past years. The recent survey of Clautiaux and Ljubic [7] attributes this trend to (i) an increased efficiency of modern ILP solvers, (ii) the rise of modeling skills in operations research curricula, and (iii) the ease of implementation of ILP-based approaches.

ILP models for the TSP have been widely studied in the literature [34] and mostly adopt the same structure: a set of binary variables keeping track of the presence of the edges contained in the solution, a set of constraints ensuring that each vertex is linked to two edges, and a set of constraints ensuring that the edges selected in a solution do not form two or more disjoint *subtours*. Most ILP models differ in the way they model the latter, usually referred to as *subtour elimination constraints* (SECs) in the literature. For example, the well-known MTZ formulation [27] uses an additional set of variables to keep track of the order in which the vertices are visited and forbids subtours by ensuring that vertices are visited in increasing order. On the other hand, the DFJ formulation [14] enumerates every subset of vertices in which a subtour can be formed and adds one SEC for each of these subsets. While the DFJ formulation is known to have a better continuous relaxation, the resulting number of constraints, which is exponential in the number of vertices, quickly becomes prohibitive. As a result, the DFJ formulation is usually solved within a *branch-and-cut* (B&C) framework, where only a subset of SECs is considered: if the solution returned by the framework contains two or more subtours, then an equal number of SECs are violated, and at least one of these must be added to prevent the same solution from being generated again; otherwise, an optimal solution for the TSP has been found.

This framework has been widely investigated in the literature, with research directions reflecting the trends of their respective time periods. For example, older contributions [12, 16, 26] considered a version of the framework that extensively utilized fractional solutions to update the SECs, in line with the era of ad hoc branch-and-bound algorithms and early-generation general-purpose ILP solvers. Conversely, more recent contributions [1, 30] fully utilized the effectiveness of available ILP solvers by studying the performance of a version of the framework in which only integer solutions were used to derive the SECs. One key finding in these contributions was that the time needed to solve a TSP instance varied significantly depending on several features, including the number and the type of SECs added per subtour-containing integer solution previously found.

The recent progress of *machine learning* (ML) and its integration into optimization algorithms [5] has added another research direction to the framework. For example, Vo et al. [42] investigated the performance of an ML-enhanced branch-and-bound algorithm for the DFJ formulation where the agent is called after solving a node to decide whether to branch on a fractional variable or to add the violated SECs instead. Regarding broader TSP literature, papers closely related to our work are

discussed in the relevant sections; meanwhile, we refer the interested reader to existing comprehensive surveys [3, 10, 17, 23, 33] for a more exhaustive overview of the field.

In this work, we extend the research direction initiated by Pferschy and Staněk [30] and Aguayo et al. [1], investigating whether the version of the framework in which only integer solutions are used to derive the SECs can be enhanced via ML. Our objective is to develop a supervised learning agent capable of predicting which SECs from a pre-defined pool of candidates should be added at the start of the framework. To achieve this goal, we address a number of intermediate questions:

- What is the best version of the B&C framework? We analyze the impact of various rules for adding violated SECs, including some that have not been tested in previous studies [1, 30], and demonstrate that some of these rules consistently outperform the others. We also provide an updated empirical evaluation of the framework, showing that it can solve TSP instances with up to 1500 nodes.

- How can we obtain a set of labeled data? We first investigate how to determine the minimum number of SECs required in the DFJ formulation for a given instance such that *none* of the optimal solutions of the model contains subtours. We propose a set covering formulation for this task and derive optimal solutions for instances with up to 100 nodes. We also compare our approach with the enumerative algorithm recently proposed by Vercesi and Buchanan [39], highlighting why the latter, which computes the minimum number of SECs required in the DFJ formulation such that *at least one* optimal solution does not contain subtours, is not suitable for this task. We empirically show that one of our tested B&C frameworks identifies a number of SECs relatively close to that found by the set covering formulation, but in a much shorter computation time. Consequently, we use this framework to identify the *true* SECs in our training dataset (i.e., the ones to be labeled 1).

- How can we obtain a set of SEC candidates to choose from? We propose two distinct procedures designed to produce a reasonable number of SEC candidates: one based on the minimum spanning tree, and the other using information from the continuous relaxation of the DFJ model without any SECs. When combined, the pool of SEC candidates generated by these two procedures captures, on average, 76.8% of the true SECs on a set of benchmark instances. However, these true SECs are mixed with many *false* ones (i.e., those to be labeled 0), which constitute 95.03% of the pool on average.

- What is the maximum gain achievable by an ML agent? We simulate the performance of the best B&C framework when different proportions of true and false SECs are added at the start. Interestingly, our results show that, under the best configuration, computation time can be reduced by up to 65%.

- How can we train the ML agent? We first identify a set of 27 SEC-related features and then train a neural network to predict the label of each SEC, achieving a recall of 0.998 at a precision of 0.143.

- How does the trained ML agent perform? We test various agents with different complexities and recall-precision trade-offs on unseen benchmark TSP instances, demonstrating that mild but consistent increases in the number of instances solved to optimality, as well as decreases in the average computation time required to solve these instances by 23%, can be achieved.

Although below the theoretically achievable 65%, this reduction is comparable to those of other ML-enhanced optimization algorithms reported in the literature [28] for other combinatorial optimization problems.

The remainder of the manuscript is organized as follows. We introduce in Section 2 the necessary notation, provide a formal problem definition, and describe the experimental setup along with the instances used in the computational experiments, which are spread throughout the paper. We discuss in Section 3 the studied B&C framework, including various features (both existing and new) for tuning it. We present our set covering formulation aimed at identifying the minimum number of SECs needed to consistently solve a TSP instance and compare its performance with other methods in Section 4. We also investigate in that section the problem of finding the maximum number of SECs that could potentially be required by the described frameworks to solve a TSP instance. We outline in Section 5 the design our ML agent, while Section 6 details how the agent is trained and tested. Finally, conclusions are drawn in Section 7.

2 Problem description and experimental setup

We consider a complete graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each vertex $i \in \mathcal{V}$ is associated with an integer index, and each pair of vertices $i, j \in \mathcal{V}$ with $i < j$ is connected by an edge $e \in \mathcal{E}$ with a non-negative length l_e. We also denote by $|\mathcal{V}| = n$ the number of nodes in the graph and by $|\mathcal{E}| = m$ the number of edges in the graph. Using binary decision variables x_e taking value 1 if edge $e \in \mathcal{E}$ is selected in the solution and value 0 otherwise, the DFJ formulation for the TSP is as follows:

$$\min \quad z = \sum_{e \in \mathcal{E}} l_e x_e \tag{1}$$

$$\text{s.t.} \quad \sum_{\substack{e=(i,j)\in\mathcal{E} \text{ with} \\ i=v||j=v}} x_e = 2 \qquad v \in \mathcal{V}, \tag{2}$$

$$\sum_{\substack{e=(i,j)\in\mathcal{E} \text{ with} \\ i,j\in\mathcal{S}}} x_e \leq |\mathcal{S}| - 1 \quad \mathcal{S} \subset \mathcal{V}, |\mathcal{S}| \geq 3, \tag{3}$$

$$x_e \in \{0, 1\} \qquad e \in \mathcal{E}. \tag{4}$$

Objective function (1) minimizes the total length of the solution, constraints (2) ensure that each vertex is connected to exactly two edges, and constraints (3) prevent subtours. Note that SECs can also be defined as:

$$\sum_{\substack{e=(i,j)\in\mathcal{E} \text{ with} \\ i\in\mathcal{S}, j\notin\mathcal{S}||i\notin\mathcal{S}, j\in\mathcal{S}}} x_e \geq 2 \quad \mathcal{S} \subset \mathcal{V}, |\mathcal{S}| \geq 3. \tag{5}$$

As there are $O(2^n)$ SECs (both when (3) and (5) are used), the DFJ model is usually solved with a subset $\mathcal{C}$ of these constraints, typically initialized as $\mathcal{C} = \emptyset$ and dynamically updated. Such an approach has been widely used when solving TSP instances in which the triangle inequality holds, also referred to as *metric* TSP instances. For example, Vercesi and Buchanan [39] showed that only 4 out of the (approximatively) 2^{42} SECs were actually needed to solve the (reduced) 42-city TSP instance discussed in the seminal paper of Dantzig et al. [14].

As far as benchmark instances are concerned, we used two groups of test instances derived from the literature. The first group is a subset of the "Symmetric traveling salesman problem (TSP)" available at the TSPLIB website [32], from which we selected all instances with less than 1500 vertices (88 in total). The second group consists of all 40 instances from the recently introduced HardTSPLIB [41], which are available in an online repository [40] and were specifically designed to exhibit large integrality gaps.

Given the need for ML-based approaches to have large training datasets, we followed the strategy adopted by recent ML-related contributions [13,20,31,42], which is to use dataset generators from (or mimicking) the DIMACS TSP Challenge [18,19]. We used the original code of two such generators:

- Generator `portgen` creates Euclidean TSP instances of a given size n. For each vertex, both the x- and y-coordinates are generated from a discrete uniform distribution $[0, 1\,000\,000]$.
- Generator `portcgen` creates clustered Euclidean TSP instances of a given size n where each cluster is composed of 100 vertices. For each of the $\frac{n}{100}$ clusters, both the x- and y-coordinates of the cluster center are drawn from a discrete uniform distribution $[0, 1\,000\,000]$. Each vertex is then randomly assigned to one of the clusters, and its x- and y-coordinates are sampled from a normal distribution centered at the corresponding cluster center. The standard deviation of this distribution depends on the total number of vertices, such that instances with more vertices result in more compact clusters.

Given an instance file produced by one of these generators (consisting of a set of n pairs of coordinates), we compute the length of each edge as the Euclidean distance between its two endpoints rounded to the nearest integer. For testing purposes, we generated $2 \times 15 \times 10 = 300$ instances: for each generator and for each n value in the set $\{100, 200, \ldots, 1500\}$, we created 10 instances.

As far as the experimental setup is concerned, all computational tests described in the following sections were executed on a single thread of a virtual machine `AMD EPYC-Milan Processor` with 2.25 GHz and 185 GB of RAM memory operating under `Ubuntu 22.04`. All tested approaches were coded in `C++`, and all ILP models were solved using `Gurobi 11.0.3` with default parameters unless stated otherwise. The only exception was `MIPGap`, which was set to 0 to ensure that the solver did not prematurely terminate before solving an instance to optimality. Each run was subject to a time limit of 3600 seconds. The neural networks were trained using `python` packages `tensorflow/keras` and `scikit-learn` for data splitting and evaluation. All our instances and source codes are available for download at `https://github.com/fatih-burak/ML-TSP`.

3 B&C framework implementation

While it is widely accepted that it is more effective to solve the DFJ formulation within a B&C framework, different strategies have been proposed in the literature to do so. Even under the assumption that only integer solutions are used to identify violated SECs, there are various possible implementations. For example, Pferschy and Staněk [30] discussed whether SECs of type (3), (5), or a combination of the two should be used. Their experiments showed that a strategy in which (3) is used when $|\mathcal{S}| \leq \frac{2n+1}{3}$ and (5) is used otherwise was particularly effective. Another decision asks whether to use the delayed cut generation method or to add the violated SECs on-the-fly. In the delayed approach, one solves the DFJ model with a fixed subset of SECs $\mathcal{C}$ to optimality before updating $\mathcal{C}$, if needed (i.e., if the optimal solution found still contains subtours). After each update,

the model is solved again. In the on-the-fly approach, the DFJ model is solved only once, but with a dynamic $\mathcal{C}$ that is updated every time a new incumbent solution is found during the branch-and-bound search. This is typically done using *lazy constraints* and *callback functions* in general-purpose ILP solvers. In the experiments of Pferschy and Staněk [30], the delayed approach was used, whereas in the experiments of Aguayo et al. [1], SECs were added dynamically. In the delayed approach, one also needs to decide which solutions are used to derive the violated SECs, as one can usually access all integer solutions found by the ILP solver during its search. Pferschy and Staněk [30] suggested that using every incumbent solution found by the solver to identify the violated SECs was more effective than using only the optimal solution. However, strategies between these two extremes were not tested. Finally, given a solution containing k subtours, one must also decide which of the associated SECs should be added: Aguayo et al. [1] empirically showed that adding all k SECs was more effective than adding only one (they tried adding the one with the largest cardinality and the one containing a fixed arbitrary node). However, strategies between these two extremes were not tested.

We report in the following the outcome of experiments performed on various implementations of the B&C framework, each leading to what one could consider as a separate solution method. For the delayed approach, we tested three ways to select the solutions used to derive the violated SECs and three ways to forbid a given solution with subtours, resulting in nine combinations. For each feature, we tested the two extreme options previously proposed in the literature, as well as a newly introduced intermediate one. Regarding the solution(s) used to derive the violated SECs, we tested (i) using only the optimal solution (identified by the attribute "OS" in the following), (ii) using all solutions found except the first one ("ASEF"), and (iii) using all solutions ("AS"). Note that because a given SEC can be violated in multiple solutions, any duplicates are removed. Regarding the way to forbid a given solution with subtours, we tested (i) adding only the smallest violated SEC ("SM"), (ii) adding all violated SECs except those involving strictly more than half of the nodes ("50"), and (iii) adding all violated SECs ("100"). For the on-the-fly approach, we tested the three ways to forbid a given solution with subtours. For the resulting twelve strategies, we tried the two types of SECs (3) and (5) (identified by the attributes "CT1" and "CT2", respectively, where "CT" stands for "cut type"). As suggested by Pferschy and Staněk [30], we also tested using (3) for SECs involving fewer than $\frac{2n+1}{3}$ vertices and (5) otherwise ("CT3"). The results obtained by these implementations on the 89 + 40 + 300 = 429 tested TSP instances are displayed in Table 1, where, for each feature combination, we report the number of instances solved to optimality (column "#opt") and the average CPU time in seconds over all runs, including those terminated due to the time limit of 3600 seconds (column "T(s)").

Table 1: Performance measures of the tested B&C framework implementations on all 429 instances.

	Delayed						On-the-fly	
	OS		ASEF		AS			
	#opt	T(s)	#opt	T(s)	#opt	T(s)	#opt	T(s)
CT1-SM	191	2207.7	233	1919.1	235	1929.2	216	2082.9
CT1-50	280	1539.1	**316**	1324.1	**311**	1354.2	262	1670.7
CT1-100	279	1550.4	**317**	1351.3	**315**	1389.0	257	1735.1
CT2-SM	181	2305.6	216	2071.0	207	2122.4	211	2086.1
CT2-50	257	1727.2	271	1653.7	272	1685.6	258	1662.1
CT2-100	260	1707.4	270	1688.7	262	1730.0	264	1673.1
CT3-100*	281	1549.2	**319**	1341.2	**314**	1377.5	258	1695.9

*By design, CT3-SM and CT3-50 are identical to CT1-SM and CT1-50, respectively.

First, we clearly observe that the worst-performing implementations of the framework (highlighted in dark gray in the table) are those that add only the smallest violated SEC to forbid a solution. These

are the only ones that solved at most 235 instances to optimality. Then, we distinguish between the on-the-fly versions of the framework and those that use delayed cuts: whereas the former (highlighted in gray in the table) never solved more than 264 of the tested instances to optimality, the latter often significantly surpassed this number. Next, we notice a clear underperformance of the solution methods that use only SECs of type (5) (in light gray in the table), as these solved at most 272 instances to optimality, whereas the others always exceeded that number. Finally, it appears that the approaches that use only the optimal solution to derive the violated SECs were clearly below, solving at most 281 instances to optimality compared to 311 for the remaining approaches (in bold in the table). Comparing the best six approaches, it seems like not using the first solution found by the solver to derive violated SECs is slightly beneficial. Based on our observations, this could be explained by the fact that the first solution found by the solver (which is often derived from a heuristic run prior to solving the root node relaxation) has a value that is on average more than ten times larger than that of the optimal solution of the model. This difference, which was observed throughout most (if not all) iterations of the framework, may explain why the SECs identified in this low-quality solution are unlikely to be needed.

While **CT3-100-ASEF** seems to obtain the best results overall, we highlight that, due to the inherent randomness of ILP solvers, changing the random seed used by the solver could increase or decrease the number of instances solved to optimality by a few (see, e.g., the experiments of Akçay and Delorme [2] on this matter in the context of packing problems with contiguity constraints). It is also worth mentioning that 344 instances out of 429 could be solved by at least one of the tested versions. We report in Table 2 the number of instances solved (column "#opt") depending on the maximum number of versions of the framework considered (column "#vers"). These results are obtained by solving a maximum coverage problem, where the goal is to cover as many elements (here, TSP instances that were solved to optimality) as possible using a fixed number of sets (here, implementations of the framework). In particular, we highlight that **CT1-50-ASEF** seems to produce results that complement those of **CT3-100-ASEF**, as the former solves nine instances that the latter could not. Adding further approaches produces only marginal gains: four additional instances were solved when also considering the results of **CT2-100-OTF**, three more when also considering those of **CT3-100-AS**, and two more when also considering those of **CT2-100-OS**. In total, ten approaches were required to solve the 344 instances.

Table 2: Number of instances solved depending on the number of implementations considered

#vers	#opt	CT1				CT2			CT3		
		50-ASEF	50-AS	50-OTF	100-OTF	SM-OTF	100-OS	100-OTF	100-AS	100-OTF	100-ASEF
1	319										x
2	328	x									x
3	334	x						x			x
4	337	x						x	x		x
5	339	x					x	x	x		x
						...					
10	344	x	x	x	x	x	x	x	x	x	x

We conclude this section by analyzing the trade-off between the number of iterations ("#iter") and the number of SECs ("#SEC") needed by the tested implementations of the framework. This should provide relevant insights to use later in our supervised learning agent when balancing false positives and false negatives: decreasing the number of SECs can be achieved by minimizing false positives, whereas decreasing the number of iterations can be achieved by minimizing false negatives.

To ensure a fair comparison, we report in Table 3 the values averaged over the 173 instances that were solved to optimality by every approach. For the on-the-fly versions of the framework, we report the average number of calls to the callback function ("#cal"), which is the same as the average number of integer solutions found during the branch-and-bound search. For the sake of clarity, we repeat the highlights from the previous table.

Table 3: Performance measures on the 173 instances solved by all B&C implementations

	Delayed									On-the-fly		
	OS			ASEF			AS					
	T(s)	#iter	#SEC	T(s)	#iter	#SEC	T(s)	#iter	#SEC	T(s)	#cal	#SEC
CT1-SM	298.0	63.4	62.4	119.5	35.0	115.1	123.0	34.9	118.8	235.0	173.0	163.1
CT1-50	56.3	14.8	83.5	37.2	**9.2**	**150.8**	36.7	**9.3**	**158.1**	72.6	40.5	204.2
CT1-100	56.8	13.3	90.1	47.2	**8.8**	**167.7**	52.5	**8.7**	**175.5**	93.0	38.8	226.4
CT2-SM	449.9	63.5	62.5	195.0	36.5	119.3	225.7	36.5	132.9	230.6	174.8	164.6
CT2-50	89.6	14.9	83.6	70.9	9.5	175.3	83.9	9.6	189.7	73.3	39.5	204.3
CT2-100	81.2	13.4	90.3	76.9	8.8	190.2	88.6	8.9	206.1	77.4	38.0	223.4
CT3-100	57.9	13.3	90.1	40.1	**8.6**	**167.4**	42.1	**8.7**	**174.3**	67.8	37.3	222.0

Focusing on the on-the-fly versions of the framework, we observe that these typically have a large average number of SECs and a large average number of calls to the callback function. However, we notice a trade-off between these two metrics, as the versions with the lowest average number of SECs are also the ones with the highest average number of calls to the callback function.

Concerning the approaches using delayed cuts, the worst ones (with attribute "SM", highlighted in dark gray in the table) have the highest average number of iterations and the lowest average number of SECs. Note, however, that while the average number of SECs is reduced by roughly a third in the "SM" versions of the framework compared to those with attributes "50" and "100", the average number of iterations is about four times higher, indicating that the trade-off is not particularly advantageous. Regarding the type of SECs used, it is interesting to observe that it neither influences the average number of iterations nor the average number of SECs required. Mathematical models using SECs of type (5) simply take more time to be solved by an ILP solver than the others. When it comes to the solution(s) used to derive the violated SECs, using only the optimal one (attribute "OS") roughly halves the average number of SECs required, while it increases the average number of iterations by around 50% compared to the other strategies (attributes "ASEF" and "AS"), highlighting a more favorable trade-off than the one obtained when using "SM" over "50" and "100". Nevertheless, approaches with attribute "OS" underperform. Focusing on the approaches with attributes "ASEF" and "AS", we observe that the former tend to use fewer SECs on average than the latter, while requiring about the same average number of iterations, confirming the hypothesis that the SECs derived from the first (and low-quality) solution found by the solver can safely be ignored.

4 Minimum and maximum number of SECs required

The experiments in the previous section showed that, for a given TSP instance, different B&C framework implementations result in different numbers of SECs being added before reaching an optimal solution. While we observed a negative correlation between the number of SECs and the number of iterations required, it is also of interest to determine the possible range of these values. From an ML perspective, one would want to minimize the number of SECs that the agent needs to identify. We saw that **CT1-SM-OS** required the lowest number of SECs on average, but is this the best we can achieve? Specifically, what would be the minimum number of SECs (referred to as "$c^{\min}$" hereafter)

such that the optimal solution of the DFJ formulation returned by an ILP solver never contains subtours? On the other hand, it would also be interesting to tackle the opposite question, namely determining the maximum number of SECs ("$c^{\max}$") that might be required. Beyond its theoretical relevance, this would allow us to compare the number of SECs used by a (possibly ML-enhanced) B&C implementation with both the best possible outcome (via $c^{\min}$) and the worst-case scenario (via $c^{\max}$).

4.1 Minimum number of SECs $c^{\min}$

As for determining $c^{\min}$, the problem was recently tackled by Vercesi and Buchanan [39], who proposed an enumerative algorithm that iteratively solves the DFJ formulation with tailored subsets of SECs. These subsets are stored in a queue $\mathcal{Q}$ initialized with one element composed of the empty set. Using a breadth-first strategy, the algorithm selects and removes the first element (or subset of SECs) $\mathcal{C}$ from the queue and solves the resulting restricted DFJ formulation. If a solution with objective value z strictly below the optimal one, say z^*, is found, then it must contain subtours. For each SEC $\mathcal{S}$ that would forbid such a solution (including the SECs consisting of the union of nodes contained in two or more subtours), a new element $\{\{\mathcal{S}\} \cup \mathcal{C}\}$ is added to $\mathcal{Q}$. As soon as a solution with objective value z^* is found, then the subset of SECs $\mathcal{C}$ it considers must be minimal. The authors highlighted that solving the DFJ formulation with the SECs identified by their procedure would always produce a solution with value z^* and that at least one optimal solution of the model would not contain subtours. However, since there may also exist solutions with value z^* that do contain subtours, it is not guaranteed that every optimal solution of the resulting model is subtour-free. We highlight this behaviour in Figure 1, where we display the three optimal solutions (with objective value $z = z^* = 426$) of the DFJ formulation for the TSPLIB *eil51* instance when only the two SECs suggested by Vercesi and Buchanan [39], namely $\{7, 23, 24, 43\}$ and $\{3, 20, 35, 36\}$, are considered. The solution depicted on the left of the figure does not contain any subtours and is therefore optimal for the TSP, whereas the solutions shown in the middle and on the right both contain subtours that are not forbidden by either $\{7, 23, 24, 43\}$ or $\{3, 20, 35, 36\}$ (highlighted in pink in the figure). A third SEC, namely $\{1, 2, 5, 6, 11, 16, 22, 32, 35, 46, 50\}$ (shown in green in the figure), is needed in the DFJ formulation to ensure that every optimal solution of the model is subtour-free. From now on, we refer to the value computed with the approach of Vercesi and Buchanan [39] as $c^{\min}_{LB}$, as it corresponds to a lower bound on our definition of $c^{\min}$.

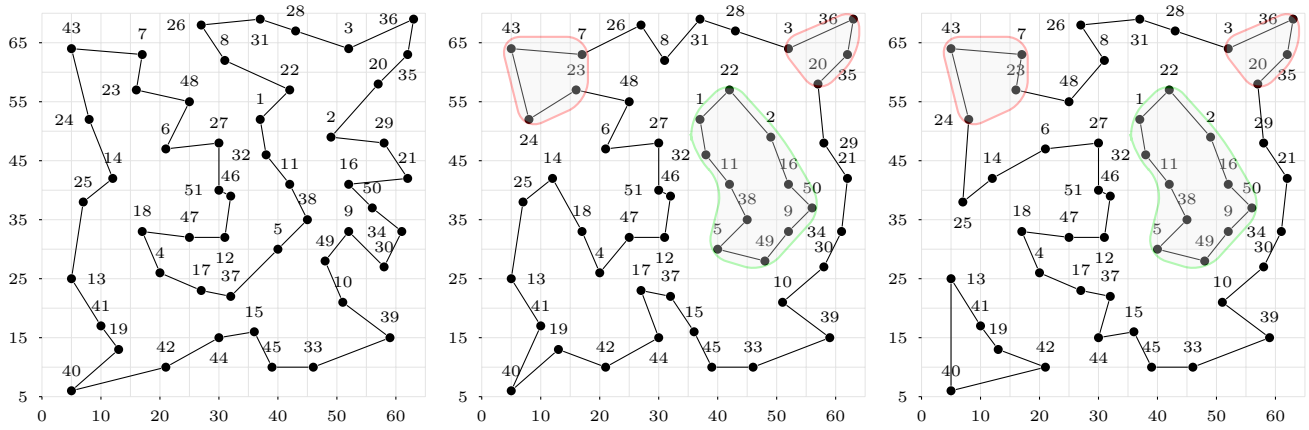
We thus developed another procedure that first enumerates every solution of the DFJ formulation with objective value $z \leq z^*$. Once all these solutions are found, we keep only those that contain subtours and solve a set covering problem where the objective is to cover each solution using the minimum number of SECs. Note that a similar approach using only a subset of these solutions was suggested by Vercesi and Buchanan [39] to obtain a lower bound for their $c^{\min}_{LB}$.

To enumerate every solution of the DFJ formulation with objective value $z \leq z^*$, a straightforward approach is to solve model (1),(2),(4) to optimality, iteratively excluding each solution found by adding a no-good cut until a solution with value $z > z^*$ is returned. Given a solution $\bar{x}$, the associated no-good cut is defined as:

$$\sum_{\substack{e \in \mathcal{E} \text{ with} \\ \bar{x}_e = 1}} x_e \leq n - 1. \tag{6}$$

While such an approach is valid, it is not particularly effective as it generates many solutions involving

Figure 1: Three optimal solutions of the DFJ formulation for the *eil51* instance in which only SECs $\{7, 23, 24, 43\}$ and $\{3, 20, 35, 36\}$ are considered.



(a) Solution 1 (without subtours) (b) Solution 2 (with subtours) (c) Solution 3 (with subtours)

the exact same set of subtours—for example, the solution composed of subtours $1 \to 2 \to 3 \to 4 \to 1$ and $5 \to 6 \to 7 \to 5$, and the solution composed of subtours $1 \to 3 \to 4 \to 2 \to 1$ and $5 \to 6 \to 7 \to 5$. To avoid this issue, given a solution k that includes the set of subtours $\mathcal{C}(k)$, we instead add the cut:

$$\sum_{\mathcal{S} \in \mathcal{C}(k)} \sum_{\substack{e=(i,j) \in \mathcal{E} \text{ with} \\ i,j \in \mathcal{S}}} x_e \leq n - 1. \tag{7}$$

Note that it may also be necessary to forbid an optimal TSP solution (i.e., with objective value z^* and no subtours) if other solutions with the objective value z^* remain to be found. Since optimal TSP solutions cannot be excluded using (7) (as $|\mathcal{C}(k)| = 1$ in that case, meaning that the only subset $\mathcal{S}$ it contains includes all vertices), we exclude them using (6) instead.

After collecting all solutions in the set $\mathcal{X}$, we identify, for each solution $k \in \mathcal{X}$, the set of subtours $\mathcal{C}(k)$ it contains. We then gather all unique subtours into the set $\mathcal{C}$, defined as $\mathcal{C} = \bigcup_{k \in \mathcal{X}} \{\mathcal{C}(k)\}$. For a given solution $k \in \mathcal{X}$, we also define $\mathcal{U}(k)$ as the set of all subtours s such that adding the SEC associated with s would forbid solution k. For example, if $\mathcal{C}(k) = \big\{\{1,2,3\},\{4,5,6\},\{7,8,9\}\big\}$, then $\mathcal{U}(k) = \big\{\{1,2,3\},\{4,5,6\},\{7,8,9\},\{1,2,3,4,5,6\},\ \{1,2,3,7,8,9\},\{4,5,6,7,8,9\}\big\}$. Introducing binary decision variable x_s ($s \in \mathcal{C}$) taking the value 1 if the SEC associated with the subset of nodes s is included in the minimum subset of SECs, we solve the following set covering model:

$$\min \quad c^{\min} = \sum_{s \in \mathcal{C}} x_s \tag{8}$$

$$\text{s.t.} \quad \sum_{s \in \mathcal{U}(k) \cap \mathcal{C}} x_s \geq 1 \quad k \in \mathcal{X}, \tag{9}$$

$$x_s \in \{0, 1\} \qquad s \in \mathcal{C}. \tag{10}$$

Objective function (8) minimizes the number of selected SECs, whereas constraints (9) ensure that each solution is forbidden by at least one of the selected SECs. We highlight that, for a given solution $k \in \mathcal{X}$, there may exist subtours $s \in \mathcal{U}(k)$ that do not belong to $\mathcal{C}$. For example, if $\mathcal{C}(k) = \big\{\{1,2,3\},\{4,5,6\},\{7,8,9\}\big\}$, then $\{1,2,3,4,5,6\}$ always belongs to $\mathcal{U}(k)$ but only belongs to $\mathcal{C}$ if there exists another solution $k' \in \mathcal{X}$ such that $\{1,2,3,4,5,6\} \in \mathcal{C}(k')$.

We provide an example of the computation of $c^{\min}$ for the TSPLIB instance *ulysse16* in Table 4. The set $\mathcal{X}$ consists of nine solutions, and only four SECs (highlighted in different colors in the table) are needed to eliminate them. Therefore, solving the DFJ formulation with these four SECs always returns a solution with no subtours.

Table 4: Values related to the computation of $c^{\min}$ for instance *ulysses16*.

z	$C(k)$			
6113	{1 2 3}	{4 5 6 8 9 10 14}	{0 7 15}	{11 12 13}
6136	{0 1 2 3 7 15}	{4 5 6 8 9 10 14}	{11 12 13}	
6205	{1 2 3}	{4 5 6 8 9 10 11 12 13 14}	{0 7 15}	
6228	{0 1 2 3 7 15}	{4 5 6 8 9 10 11 12 13 14}		
6698	{1 2 3}	{4 5 6 8 9 10 14}	{0 7 11 12 13 15}	
6743	{0 1 2 3 7}	{4 5 6 8 9 10 14}	{11 12 13 15}	
6770	{1 2 3}	{0 4 5 6 7 8 9 10 11 12 13 14 15}		
6777	{0 1 2 3 7 11 12 13 15}	{4 5 6 8 9 10 14}		
6825	{0 1 2 3 7}	{4 5 6 8 9 10 11 12 13 14 15}		
6859*	{0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15}			

4.2 Maximum number of SECs $c^{\max}$

As for determining $c^{\max}$, the problem has, to the best of our knowledge, never been studied in the literature before. In this work, we define $c^{\max}$ as the maximum number of SECs that may be needed to solve a given TSP instance using a B&C implementation that adds only the SECs corresponding to (some of) the subtours contained in an optimal solution of the DFJ formulation. We used this definition of $c^{\max}$ because:

- If $c^{\max}$ is not tied to a specific B&C implementation, then a straightforward approach would be to take a solution with subtours and objective value $z \leq z^*$ for the TSP instance of interest, and then add all $O(2^n)$ SECs except those corresponding to the subtours in that solution. This would lead to a situation in which almost all SECs must be added before finding a solution without subtours.

- If one considers a B&C implementation in which solutions with objective value $z > z^*$ may also be used to derive the SECs—such as those with attributes "ASEF", "AS", and "OTF"—then the set of solutions that may be visited becomes extremely large (note that the $n!$ feasible TSP solutions form only a small subset of this larger set), and the worst-case scenario (though extremely unlikely in practice) is again a value close to $O(2^n)$.

In the following, we consider the same set of solutions $\mathcal{X}$ with subtours and objective value $z \leq z^*$ as before, but we now assume an ordering over these solutions, where $o(k)$ represents the position in which solution k would be visited by the B&C implementation, ranging from 1 to $|\mathcal{X}|$. We also refer to this ordering as the *index* of a solution. Given two solutions k and k' with objective values z_k and $z_{k'}$, then $o(k) < o(k')$ if $z_k < z_{k'}$. In this setting, we consider that ties are broken randomly. In other words, if two solutions have the same objective value, then it is assumed that one will always be found before the other. The problem is now to determine the maximum number of SECs that can be added under the constraints that (i) a SEC can only be added if it is violated by a visited solution and (ii) a solution can only be visited if no SECs forbidding it was added when visiting a solution with a lower index. Introducing now binary variables x_{sk} ($k \in \mathcal{X}, s \in \mathcal{C}(k)$) taking the value

1 if SEC s is added when visiting solution k, an ILP model determining $c^{\max}$ is as follows:

$$\max \quad c^{\max} = \sum_{k \in \mathcal{X}} \sum_{s \in \mathcal{C}(k)} x_{sk} \tag{11}$$

$$\text{s.t.} \quad \sum_{s \in \mathcal{U}(k) \cap \mathcal{C}} \sum_{\substack{k' \in \mathcal{X} \text{ with} \\ o(k') \leq o(k),\, s \in \mathcal{C}(k')}} x_{sk'} \geq 1 \quad k \in \mathcal{X}, \tag{12}$$

$$x_{s'k} \leq 1 - \sum_{\substack{k' \in \mathcal{X} \text{ with} \\ o(k') < o(k),\, s \in \mathcal{C}(k')}} x_{sk'} \quad k \in \mathcal{X}, s \in \mathcal{U}(k) \cap \mathcal{C}, s' \in \mathcal{C}(k), \tag{13}$$

$$x_{sk} \in \{0, 1\} \quad k \in \mathcal{X}, s \in \mathcal{C}(k). \tag{14}$$

Objective function (11) maximizes the number of selected SECs, whereas constraints (12) ensure that every solution k is forbidden by at least one SEC added when visiting k or a solution k' such that $o(k') < o(k)$ $(k, k' \in \mathcal{X})$. Constraints (13) ensure that a SEC can only be added for a given solution k if the solution is visited, or in other words, if none of the SECs s $(s \in \mathcal{U}(k) \cap \mathcal{C})$ that would forbid k from being generated were added when visiting a solution k' such that $o(k') < o(k)$ $(k, k' \in \mathcal{X})$.

We also provide an example of the computation of $c^{\max}$ for the TSPLIB instance *ulysse16* in Table 5. Naturally, the set $\mathcal{X}$ consists of the same nine solutions, but in this case a total of twelve SECs are added to eliminate them. These SECs are highlighted in gray in the table in the row corresponding to the solution at which they are added to the formulation. Recall that a SEC can only be added for solutions that are actually visited (these are marked with a checkmark in column "vis."). For example, the second solution (i.e., with $o(k) = 2$ and objective value $z = 6136$) is not visited because it contains subtour $\{11, 12, 13\}$, for which the corresponding SEC was already added when visiting the first solution.

Table 5: Values related to the computation of $c^{\max}$ for instance *ulysses16*.

$o(k)$	z	vis.	$C(k)$			
1	6113	✓	{1 2 3}	{4 5 6 8 9 10 14}	{0 7 15}	{11 12 13}
2	6136	✗	{0 1 2 3 7 15}	{4 5 6 8 9 10 14}	{11 12 13}	
3	6205	✗	{1 2 3}	{4 5 6 8 9 10 11 12 13 14}	{0 7 15}	
4	6228	✓	{0 1 2 3 7 15}	{4 5 6 8 9 10 11 12 13 14}		
5	6698	✓	{1 2 3}	{4 5 6 8 9 10 14}	{0 7 11 12 13 15}	
6	6743	✓	{0 1 2 3 7}	{4 5 6 8 9 10 14}	{11 12 13 15}	
7	6770	✓	{1 2 3}	{0 4 5 6 7 8 9 10 11 12 13 14 15}		
8	6777	✓	{0 1 2 3 7 11 12 13 15}	{4 5 6 8 9 10 14}		
9	6825	✓	{0 1 2 3 7}	{4 5 6 8 9 10 11 12 13 14 15}		
10	6859*	✓	{0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15}			

4.3 Empirical results for $c^{\min}$ and $c^{\max}$

We report in Table 6 various values related to $c^{\min}$ and $c^{\max}$ for the subset of TSP instances for which we were able to determine $\mathcal{X}$ within three days of computation time (the time required to solve models (8)-(10) and (11)-(14) was always negligible compared to the time needed to compute $\mathcal{X}$). For each instance, we provide (in order) the dataset, the name, the number of vertices, the optimal solution value, the number of optimal solutions, the number of solutions with subtours and $z \leq z^*$, the number of solutions with subtours and $z = z^*$, the value of $c^{\min}_{LB}$ computed with the approach of Vercesi and Buchanan [39] (to which we also applied a three-day time limit), the value of $c^{\min}$ computed by model (8)-(10), the number of optimal solutions for model (8)-(10), the value of $c^{\max}$ computed by model (11)-(14), the number of SECs required by the B&C implementation

CT1-SM-OS, and the number of SECs that are common between those added in **CT1-SM-OS** and those that constitute $c^{\min}$ (we report the maximum value if there is more than one optimal solution for model (8)-(10)).

Table 6: Values related to $c^{\min}$ and $c^{\max}$ for a subset of instances

Instance			Opt. TSP sol.		DFJ sol. $\mathcal{X}$		$c^{\min}$			$c^{\max}$	**CT1-SM-OS**	
dataset	name	n	z^*	# sol.	$\#z \leq z^*$	$\#z = z^*$	$c^{\min}_{\text{LB}}$	#SEC	# sol.	#SEC	#SEC	#same as $c^{\min}$
TSPLIB	burma14	14	3323	1	3	0	2	2	1	5	3	2
	ulysses16	16	6859	1	9	0	4	4	1	12	6	4
	gr17	17	2085	1	28	0	5	5	1	23	11	5
	gr21	21	2707	1	0	0	0	0	1	0	0	0
	ulysses22	22	7013	1	38	0	5	5	1	27	10	5
	gr24	24	1272	2	3	0	1	1	1	6	2	1
	fri26	26	937	2	40	1	4	4	1	32	6	3
	bayg29	29	1610	1	8	0	4	4	1	12	4	4
	bays29	29	2020	1	11	0	5	5	3	17	5	5
	dantzig42	42	699	1	12	0	4	4	1	42	5	4
	swiss42	42	1273	4	51	0	3	3	3	46	4	3
	att48	48	10628	1	111	0	10	10	3	76	15	10
	gr48	48	5046	2	166	0	11	11	3	86	15	11
	hk48	48	11461	1	83	0	8	8	1	72	11	8
	eil51	51	426	2	6	1	2	3	1	11	3	3
	berlin52	52	7542	1	5	0	2	2	1	52	3	2
	brazil58	58	25395	–	–	–	11	–	–	–	24	–
	st70	70	675	–	–	–	12	–	–	–	20	–
	eil76	76	538	20	8	4	2	5	1	13	3	3
	gr96	96	55209	1	3518	0	20	20	2	1736	30	19
	rat99	99	1211	1	61	8	7	8	3	72	12	7
	eil101	101	629	97	53	9	4	9	1	64	5	2
Hard TSPLIB	10001	10	1000	44	7	0	7	7	2	14	7	7
	10007	10	1000	42	6	0	6	6	1	12	6	6
	10008	10	1000	53	7	0	7	7	2	14	7	7
	10010	10	1000	51	7	0	7	7	2	14	7	6
	11675	11	1000	54	8	0	8	8	1	16	8	8
	14850	14	1000	99	23	0	19	19	2	36	19	19
	15002	15	1000	454	31	0	22	22	1	42	22	22
	15005	15	1000	443	27	2	21	23	1	44	21	21
PORTGEN	100-9	100	7332622	1	3023	0	–	17	19	776	23	16
PORTCGEN	100-9	100	3255553	1	23	0	4	4	3	25	5	4

First, we highlight that our approach could not find $c^{\min}$ for two instances—*brazil58* and *st70*—within the time limit, whereas the procedure of Vercesi and Buchanan [39] could determine $c^{\min}_{LB}$. The opposite situation also arose for one 100-vertex instance generated with `portcgen`. Second, we confirm that the only instances for which $c^{\min}$ differs from $c^{\min}_{LB}$ are those for which there exists at least one solution with $z = z^*$ containing subtours. For these instances, we also reran model (8)-(10) after excluding those solutions from $\mathcal{X}$ and consistently obtained $c^{\min} = c^{\min}_{LB}$. Third, we observe that $c^{\max}$ is often up to one order of magnitude larger than $c^{\min}$ but is always far below the theoretical $O(2^n)$ maximum. Finally, we note that the number of SECs required by **CT1-SM-OS** is often very close to both $c^{\min}_{LB}$ and $c^{\min}$. In TSP instances for which there exists at least one solution with $z = z^*$ containing subtours, it is possible for **CT1-SM-OS** to terminate after adding even fewer SECs than $c^{\min}$ if at least one such solution is not encountered. Because of the extensive computation time required to determine $c^{\min}_{\text{LB}}$ and $c^{\min}$, in the remainder of the paper we will use the set of SECs included by **CT1-SM-OS** upon termination—denoted by $\mathcal{C}^*$—as a proxy for the minimum set of SECs.

5 Designing the ML component

In this section, we describe the creation of a supervised learning agent capable of determining $\mathcal{C}^*$. We first motivate its relevance by simulating the performance of our best B&C framework implementation under both perfect and imperfect knowledge of $\mathcal{C}^*$, and then proceed to describe the agent itself.

5.1 Proof of concept

To compute $\mathcal{C}^*$ for a given TSP instance, we let **CT1-SM-OS** run until an optimal solution was found. Using a three-day time limit per instance, we were able to obtain the desired subset for 299 out of the 429 instances in our dataset. To estimate the potential gains achievable by enhancing our B&C implementations with an ML agent capable of predicting $\mathcal{C}^*$, we reran **CT3-100-ASEF**, our top-performing variant, after pre-populating the set of SECs $\mathcal{C}$. Of course, one cannot expect an ML agent to achieve 100% accuracy: some elements of $\mathcal{C}^*$ (the true SECs) will likely be missing, whereas some elements not in $\mathcal{C}^*$ (the false SECs) will likely be included. As a result, we pre-populated $\mathcal{C}$ using the (simulated) output of an ML agent with five different *recall* levels and eight different *precision* levels [29]. Regarding recall—the ratio between the number of true SECs identified by the agent and the total number of true SECs $|\mathcal{C}^*|$—we tested 0.2, 0.4, 0.6, 0.8, and 1. Regarding precision—the proportion of true SECs among all SECs identified by the agent—we tested 1, $\frac{4}{5}$, $\frac{2}{3}$, $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{5}$, $\frac{1}{7}$, and $\frac{1}{9}$. For example, given an instance with $|\mathcal{C}^*| = 50$, an ML agent with recall 0.6 and precision $\frac{1}{7}$ would pre-populate $\mathcal{C}$ with 30 true SECs ($\frac{30}{50} = 0.6$) and 180 false SECs ($\frac{30}{30+180} = \frac{1}{7}$).

We report in Table 7 the results obtained for the $5 \times 8 = 40$ resulting recall-precision level combinations. For each combination, the true SECs are randomly selected from $\mathcal{C}^*$, while the false SECs are generated as follows: first, we draw a random integer in the range $[3, \frac{n}{2}]$; then, we randomly select that many vertices to form a subtour and add the corresponding SEC if it does not belong to $\mathcal{C}^*$ (otherwise, the process is repeated until a false SEC is obtained). At the bottom of the table, we also report the results obtained by **CT3-100-ASEF** without pre-populating $\mathcal{C}$ (the benchmark we aim to outperform). We highlight in green in the table all recall-precision combinations resulting in computation times significantly faster than the benchmark (with p-value < 0.05 and effect size $r > 0.3$ according to a two-tailed Wilcoxon signed-rank test), and in red those resulting in computation times significantly slower than the benchmark (with p-value < 0.05 and effect size $r < -0.3$ according to the same test). These thresholds were chosen based on the literature [8, 11]; the p-value determines statistical significance, while the r-value measures effect size, with $|r| \geq 0.3$ representing a moderate to large effect.

Table 7: Performance measures of **CT3-100-ASEF** after pre-populating $\mathcal{C}$ (false SECs are generated randomly) on the subset of 299 instances.

	Recall																			
	0.2				0.4				0.6				0.8				1			
Precision	#opt	T(s)	#iter	#SEC	#opt	T(s)	#iter	#SEC	#opt	T(s)	#iter	#SEC	#opt	T(s)	#iter	#SEC	#opt	T(s)	#iter	#SEC
1	285	536.3	9.6	354.7	288	502.6	8.2	326.7	291	420.3	6.5	288.6	293	355.7	4.6	248.4	292	246.8	1.1	140.2
4/5	288	546.1	9.6	365.9	288	521.5	8.0	338.7	289	460.6	6.7	318.0	291	390.6	4.7	277.5	294	248.9	1.1	173.6
2/3	286	570.5	9.7	378.6	289	550.6	8.3	357.3	289	486.2	6.6	337.9	292	407.5	4.7	307.9	294	267.9	1.1	209.6
1/2	288	578.3	9.9	394.1	291	558.7	8.4	394.6	285	543.0	6.6	385.7	288	478.6	4.6	361.6	289	299.6	1.1	277.8
1/3	283	634.3	9.8	424.1	276	686.4	8.1	447.2	272	686.0	6.5	466.1	270	706.0	4.6	469.0	290	419.0	1.1	416.5
1/5	277	718.0	9.5	477.8	251	927.5	7.7	537.4	242	1062.0	6.1	601.2	245	1066.0	4.5	675.6	277	617.7	1.1	692.4
1/7	261	853.9	9.2	521.0	235	1124.7	7.3	630.9	225	1254.2	5.9	757.8	234	1210.5	4.2	881.9	273	742.5	1.1	968.3
1/9	253	973.4	8.9	565.0	220	1290.1	7.3	727.3	217	1382.7	5.8	905.2	221	1337.8	4.2	1094.7	263	863.5	1.1	1244.5
Without pre-populating $\mathcal{C}$					286	579.5	10.9	379.7												

The results show that pre-populating $\mathcal{C}$ with the help of an ML agent can have a significant effect. In the case of a perfect agent (i.e., with recall and precision equal to 1), the average computing time is reduced by more than half. In the case of a highly effective agent, say with recall 0.8 and precision $\frac{2}{3}$, it is reduced by nearly a third. Conversely, for a poorly performing agent, say with recall 0.6 and precision $\frac{1}{9}$, the average computing time is doubled.

5.2 Types of input and output

To transform this proof of concept into a functional ML component, a key challenge is to determine the appropriate output format for the agent. While a natural approach would be to return subsets of vertices that constitute SECs in $\mathcal{C}^*$, ML architectures are generally not well suited for generating a non-fixed number of discrete sets. As a result, we propose using the agent for a *classification task*, which is a well-established strength of supervised learning [37]. In our setting, the ML agent takes a single SEC as input—the features of which are described later in this section—and produces a binary output: 1 if the candidate is a true SEC, and 0 otherwise.

That being said, calling the agent on each of the $O(2^n)$ possible SECs is impractical. We therefore propose two procedures designed to generate a restricted pool of SEC candidates on which the ML agent will perform its classification task. An ideal pool contains as many true SECs as possible (to maximize potential recall) while including as few false SECs as possible (to minimize the number of false positives).

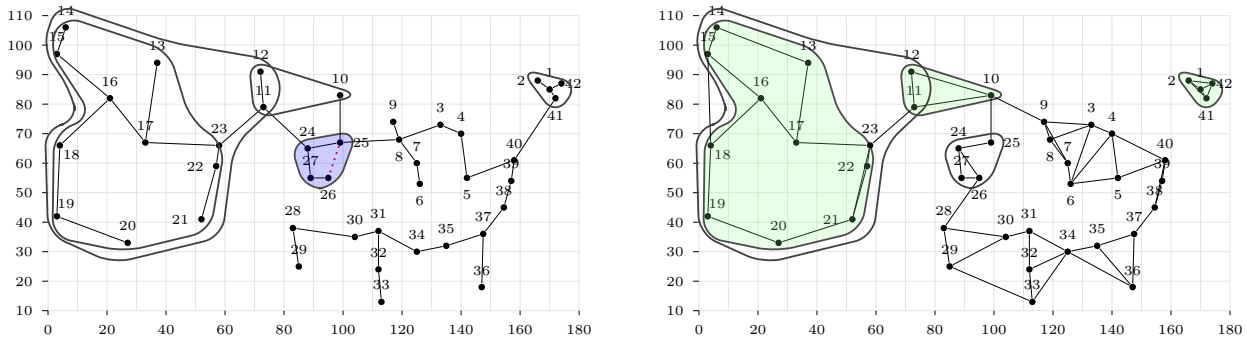
Our first procedure is based on the minimum spanning tree (MST). For a given TSP instance, we first compute the MST $\mathcal{M}$ using Prim's algorithm. Then, for each edge $e \in \mathcal{E} \setminus \mathcal{M}$, we identify the unique cycle formed by $\mathcal{M} \cup \{e\}$ and include the corresponding SEC in a candidate set $\mathcal{P}_1$ if it contains at most $\frac{n}{2}$ vertices. This threshold is imposed because, by design, **CT1-SM-OS** (and hence $\mathcal{C}^*$) never includes a SEC with more than $\frac{n}{2}$ vertices. This procedure produces $|\mathcal{E}| - (|\mathcal{V}| - 1)$ potential candidates before filtering out those that exceed the size threshold.

Our second procedure is based on linear programming and reduced costs. For a given TSP instance, we first solve the continuous relaxation of the DFJ model without any SECs. We then construct the set $\mathcal{R}$ containing all edges whose associated variables have a non-positive reduced cost. (Note that for bounded variables, separate reduced costs exist for the lower and upper bounds; Gurobi combines these into a single value because at most one of them can be non-zero.) Finally, we identify cycles within the subgraph induced by $\mathcal{R}$ and include the corresponding SECs in a candidate set $\mathcal{P}_2$.

We provide an illustration of these two procedures in Figure 2 for instance *dantzig42*. Here, $\mathcal{C}^* = \big\{\{1, 2, 41, 42\}, \{10, 11, 12\}, \{11, 12, \ldots, 23\}, \{13, 14, \ldots, 23\}, \{24, 25, 26, 27\}\big\}$. (Note that, as highlighted in Table 6, $c_{\min} = 4$ for this instance: although $\{10, 11, 12\}$ belongs to $\mathcal{C}^*$, it is not required for $c_{\min}$.) In each subfigure, the SECs in $\mathcal{C}^*$ are circled in black. Additionally, those identified by the first and second procedures are colored in blue and green, respectively. Figure 2a shows an MST $\mathcal{M}$ for the instance. Since edges $(24, 25), (26, 27)$, and $(27, 24)$ belong to $\mathcal{M}$, the SEC $\{24, 25, 26, 27\}$ is identified and added to $\mathcal{P}_1$ when the edge $e = (25, 26)$ is considered. Conversely, since neither edge $(10, 11)$ nor $(10, 12)$ belong to $\mathcal{M}$, SEC $\{10, 11, 12\}$ is not identified by this procedure. Figure 2b shows the graph $\mathcal{R}$ constructed as described above. Since edges $(10, 11), (11, 12)$, and $(12, 10)$ belong to $\mathcal{R}$, they form a cycle that results in the SEC $\{10, 11, 12\}$ being added to $\mathcal{P}_2$. Conversely, since edge $(25, 26)$ does not belong to $\mathcal{R}$, SEC $\{24, 25, 26, 27\}$ is not identified by this procedure. For this instance, there are 815 SECs in $\mathcal{P}_1$, 2725 SECs in $\mathcal{P}_2$, 3518 SECs in $\mathcal{P}_1 \cup \mathcal{P}_2$, 4 SECs in $\mathcal{C}^* \cap (\mathcal{P}_1 \cup \mathcal{P}_2)$, and 1 SEC in $\mathcal{C}^* \setminus (\mathcal{P}_1 \cup \mathcal{P}_2)$.

We report in Table 8 results similar to those in Table 7, but with false SECs randomly selected from $\mathcal{P}_1 \cup \mathcal{P}_2 \setminus \mathcal{C}^*$ (as would happen if the pool of SEC candidates were generated by the two aforementioned procedures). Selecting false SECs from $\mathcal{P}_1 \cup \mathcal{P}_2$ is expected to yield more meaningful results (in the sense of being closer to those one might expect from our ML component) and improved performance compared to the purely random generation discussed in Section 5.1. Note that the

Figure 2: Pool of SEC candidates generated by the two procedures for the *dantzig42* instance



(a) MST graph

(b) Reduced-cost graph

number of false SECs was capped by the cardinality of $\mathcal{P}_1 \cup \mathcal{P}_2 \setminus \mathcal{C}^*$, which means that the required count for certain recall-precision combinations could not be met for all instances.

Table 8: Performance measures of **CT3-100-ASEF** after pre-populating $\mathcal{C}$ (false SECs are randomly selected from $\mathcal{P}_1 \cup \mathcal{P}_2 \setminus \mathcal{C}^*$) on the subset of 299 instances.

	Recall																	
	0.2				0.4				0.6				0.8				1			
Precision	#opt	T(s)	#iter	#SEC	#opt	T(s)	#iter	#SEC	#opt	T(s)	#iter	#SEC	#opt	T(s)	#iter	#SEC	#opt	T(s)	#iter	#SEC
1	290	540.7	9.6	355.7	294	492.1	8.1	321.6	290	438.0	6.5	287.5	288	385.9	4.7	245.2	292	249.6	1.1	140.2
4/5	289	537.0	9.7	359.9	292	476.8	8.1	330.8	290	463.2	6.4	307.2	290	380.9	4.5	271.9	295	235.4	1.0	174.0
2/3	288	520.3	9.5	366.3	289	506.0	8.2	346.8	290	449.0	6.7	327.0	295	369.4	4.6	291.2	293	255.2	1.1	209.1
1/2	289	535.7	9.8	378.9	296	460.3	8.3	368.6	296	416.2	6.7	360.4	294	363.0	4.7	344.5	295	222.6	1.1	277.3
1/3	289	534.7	9.8	402.4	291	479.0	8.3	416.5	295	395.2	6.7	436.5	298	364.2	4.8	445.1	296	201.0	1.1	408.8
1/5	291	531.6	9.8	445.4	290	487.0	8.2	513.8	294	433.1	6.6	578.9	295	392.2	4.8	648.2	295	243.2	1.1	666.0
1/7	291	534.2	9.7	497.2	291	512.1	8.4	611.3	294	466.6	6.7	732.1	295	408.6	4.8	849.0	295	233.6	1.1	923.2
1/9	289	571.1	9.9	545.0	289	527.4	8.1	708.1	291	478.6	6.7	879.7	290	438.2	4.8	1051.6	296	239.1	1.1	1179.7
Without pre-populating $\mathcal{C}$					286	579.5	10.9	379.7												

Interestingly, the results show that, regardless of the recall-precision performance achieved by the ML component, the average computing time is reduced (although not always significantly) when pre-populating $\mathcal{C}$ with SECs coming from $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{C}^*$. Even more noteworthy is the fact that there are now some precision-recall combinations that slightly outperform the perfect agent: adding SECs from $\mathcal{P}_1$ and $\mathcal{P}_2$ that are not in $\mathcal{C}^*$ (i.e., false SECs which are, a priori, not useful) further reduces the computing time. Further investigation showed that these additional cuts help increase the dual bound more quickly. From an ML perspective, this suggests that increasing the number of false positives may not be problematic if it also leads to a higher number of true positives.

5.3 Implementing the procedures

While the two proposed procedures are conceptually intuitive, their implementation is less straightforward, especially under the natural goal of keeping the overall computation time as low as possible: indeed, the time saved by the ML component should not be offset by the time required to generate $\mathcal{P}_1$ and $\mathcal{P}_2$.

In the MST procedure, the task is to identify the cycle in a unicyclic graph (a connected graph that contains exactly one cycle). Whereas this task could be completed using a simple depth-first search (DFS) algorithm, the necessity of repeating such a procedure for every edge not included in the MST demands a more efficient implementation. In practice, we aim to identify the unique path in the MST $\mathcal{M}$ between every vertex pair $v, v' \in \mathcal{V}$ corresponding to an edge $(v, v') \in \mathcal{E} \setminus \mathcal{M}$, as this

path combined with the edge itself forms the desired cycle. We employed a lowest common ancestor (LCA) [4] approach to retrieve these paths. We first transformed the MST into a rooted tree by selecting an arbitrary vertex as the root, then performed a single DFS traversal to compute both the parent and the depth of each vertex. Finally, for each required pair, we determined the path by moving upwards through the ancestors of each vertex until their LCA is reached. While the overall procedure is computationally fast, the number of generated cycles (and thus SEC candidates) can be very large (as it is quadratic in the number of vertices). This may increase the computation time for the ML component, since a set of input features must be calculated for every single SEC candidate. To mitigate this issue, we also devised a similar but truncated strategy for building $\mathcal{P}_1$, in which we consider only the edges $e \in \mathcal{E} \setminus \mathcal{M}$ whose length is below a certain instance-dependent threshold. After running several preliminary tests to determine the most suitable type of threshold, we opted for $\alpha\frac{\tilde{z}}{n}$, where $\tilde{z}$ is the objective value of the continuous relaxation of the DFJ formulation without any SECs, and α is a constant (the higher its value, the more edges fall below the resulting threshold and, consequently, the more SECs are included in $\mathcal{P}_1$). The empirical performance of the MST procedure (corresponding to $\alpha = \infty$) and of its truncated versions for $\alpha = 5, 10$, and 20 is displayed in Table 9, where, for each version, we report the average computing time, the average and median numbers of SECs identified overall, and the average proportion of true SECs identified.

Table 9: Performance of $\mathcal{P}_1$-related strategies on the subset of 299 instances.

α	T (s)	Avg. cycles	Med. cycles	Match (%)
5	0.0	8447.2	6893	67.6
10	0.0	25 742.5	18 537	68.8
20	0.1	66 783.1	37 955	68.9
∞	0.3	169 627.8	79 401	68.9

We observe that the MST procedure is relatively effective: on average, it runs in 0.3s and detects 68.9% of the SECs in $\mathcal{C}^*$. The truncated versions greatly alleviate the main drawback of the MST procedure, namely the large number of SECs identified overall, reducing this number by a factor of 20 when $\alpha = 5$, at the cost of a decrease of slightly more than 1% in the proportion of true SECs identified. We find that the truncated version with $\alpha = 10$ provides a better compromise, with only a minor loss of 0.1% in the proportion of true SECs identified (a more acceptable loss than that with $\alpha = 5$), while still reducing the average number of SECs identified overall by a factor of 6.5 compared to the complete version.

In the reduced cost procedure, the task is to identify all cycles in a graph. Following the framework described by Lee [24], we proceeded as follows. First, we determine a *cycle basis* $\mathcal{B}_1$ of $\mathcal{R}$. This is achieved by computing a spanning tree $\mathcal{M}'$ of $\mathcal{R}$ and, as in the first procedure, looping over each edge $e \in \mathcal{R} \setminus \mathcal{M}'$ and adding the unique cycle formed by $\mathcal{M}' \cup \{e\}$ to $\mathcal{B}_1$. The cycles in $\mathcal{B}_1$ are called *fundamental cycles*. It is a well-known property in graph theory that any cycle in $\mathcal{R}$ can be represented as the symmetric difference (referred to as *XOR sum* hereafter) of a subset of these fundamental cycles. However, computing all $O(2^{|\mathcal{B}_1|})$ possible XOR sums is not practical. As a result, we restrict ourselves to a heuristic set $\mathcal{B}$ formed by the union of sets $\mathcal{B}_1, \ldots, \mathcal{B}_K$, where each set $\mathcal{B}_i$ $(i = 2, \ldots, K)$ is constructed by taking the XOR sum of every tuple of i distinct fundamental cycles, provided the result is itself a cycle. For each (distinct) cycle in $\mathcal{B}$, we add the corresponding SEC to $\mathcal{P}_2$ if the cycle is *simple* (i.e., each vertex is visited at most once) and contains no more $\frac{n}{2}$ vertices.

Once again, the rapidly increasing computation times of such a procedure required additional efforts to ensure an efficient implementation. First, as the graph resulting from $\mathcal{R}$ is not necessarily

connected, we decomposed the set into connected components using DFS. Each of these components was then further decomposed into biconnected components (maximal subgraphs that remain connected after removing any single vertex and its incident edges) using the procedure described by Tarjan [38]. Note that a cycle may only exist within a biconnected component. Afterwards, for each biconnected component, the same strategy as that used for the MST procedure was employed to determine a cycle basis (this time looping only over the edges belonging to that component). Fundamental cycles were then encoded as bitsets, a data structure that is more effective than vectors of integers or booleans for performing the operations needed to build sets $\mathcal{B}_2, \ldots, \mathcal{B}_K$.

We now discuss additional $\mathcal{P}_2$-related strategies aimed at either further reducing the computation time required to compute the set or increasing the number of true SECs identified. The empirical performance of each of these strategies is reported in Table 10. First, we tested various values of K, with higher values expected to identify more true SECs but also require more computation time. Second, we attempted to reduce the number of cycles stored in each set $\mathcal{B}_i$ $(i = 2, \ldots, K)$ by constructing the sets iteratively: for each i, we built $\mathcal{B}_i$ by taking the XOR sum of every pair (b, b') where $b \in \mathcal{B}_1$ and $b' \in \mathcal{B}_{i-1}$ provided that (i) b and b' share at least one edge, (ii) the resulting cycle is simple, and (iii) it contains $\frac{n}{2}$ vertices or fewer (note that we previously applied these reductions only when generating SECs from $\mathcal{B}$, as the XOR sum operator can reduce the number of vertices or make a cycle simple). This pruning procedure (identified as "**P**" in the table) is expected to decrease the computation time at the cost of identifying fewer true SECs. Third, we varied the number of edges initially included in $\mathcal{R}$. To increase the size of the set, we added all edges with reduced cost below a small positive threshold, namely 0.025% ("+") and 0.05% ("++") of the instance's continuous relaxation value. To decrease the size of the set, we had to employ a different strategy, as using a negative threshold would simply remove too many edges. Hence, we searched for a meaningful way to remove from $\mathcal{R}$ a certain instance-dependent number of edges with reduced cost 0. To this end, we modeled the relationship between the number of edges included by the baseline procedure and the total number of edges in the instance (see Figure 3) with a log-log regression (shown in black in the figure). We then removed the longest edge with reduced cost 0 from $\mathcal{R}$ as long as the size of the set remained above the fitted line ("-"). To reduce the size of $\mathcal{R}$ even further ("--"), we repeated this procedure using a second, artificially generated line with a smaller power coefficient (shown in red in the figure). Naturally, both the computation time and the number of identified SECs are positively correlated with the size of $\mathcal{R}$. Finally, we experimented with building the cycle basis using an MST instead of a random spanning tree. We tested two approaches: one using actual edge lengths ("**L**") and another using the reduced costs ("**R**") associated with the edges. This change is not expected to impact the average computing time, but it may increase the number of identified SECs. For each $\mathcal{P}_2$-related strategy, we report metrics similar to those in Table 9, together with the average proportion of true SECs identified, both when the strategy is combined with the MST procedure and when it is combined with its truncated version with $\alpha = 10$.

Regarding the reduced-cost procedure, we observe substantial variation in both computation time and number of generated SECs, depending on the strategy used. Overall, our findings are as follows: (i) setting K to 3 offers the best compromise, balancing low average computation time with the detection of many true SECs; (ii) applying the pruning procedure is largely beneficial, as it significantly reduces computation time while only slightly decreasing the number of true SECs detected; (iii) reducing the size of $\mathcal{R}$ so that it never exceeds the black fitted line is also advantageous, greatly lowering the number of generated SECs with almost no impact on the number of true SECs

Figure 3: Number of edges with non-positive reduced cost versus total number of edges (logarithmic scale).

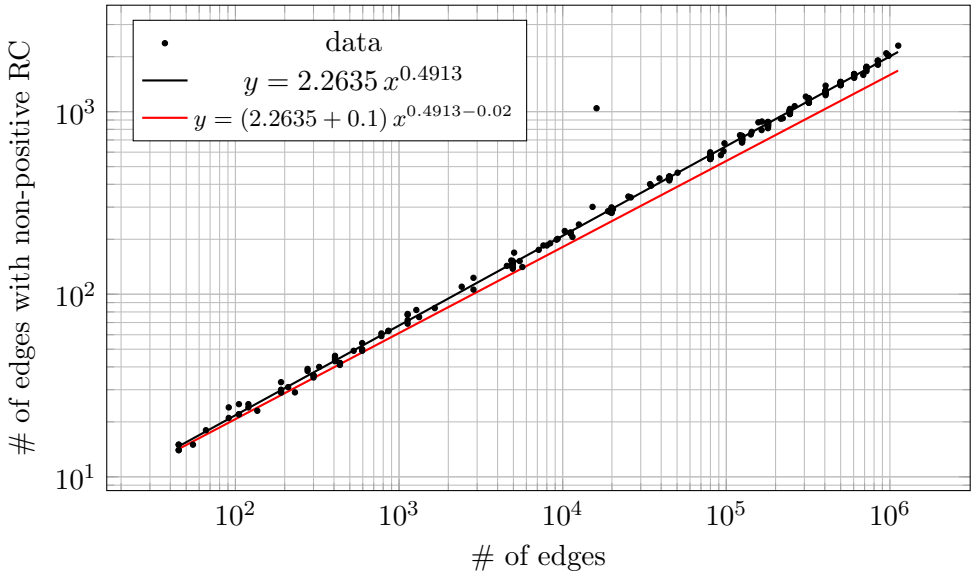


Table 10: Performance of $\mathcal{P}_2$-related strategies on the subset of 299 instances.

Strategy				T(s)	Avg. cycles	Med. cycles	Match (%)	Comb.match (%)	
K	**P**	$\lvert\mathcal{R}\rvert$	MST					$\alpha = 10$	$\alpha = \infty$
1				0.7	205.9	162.0	35.3	71.9	72.0
2				0.7	964.4	432.0	48.4	74.9	75.0
3				6.7	31 023.9	1549.0	51.2	76.3	76.4
4*				480.3	151 100.2	6361.5	52.2	76.9	77.0
2	✓			0.7	964.2	432.0	48.2	74.9	75.0
3	✓			1.3	31 023.5	1549.0	50.9	76.3	76.4
4*	✓			6.2	151 099.4	6361.5	51.9	76.9	77.0
5**	✓			84.3	1 860 671.2	23561.0	52.3	77.4	77.5
3	✓	−−		0.7	417.6	268.0	41.1	73.8	74.1
3	✓	−		0.7	3682.0	1498.0	50.5	76.0	76.1
3	✓	+		6.7	163 001.8	6585.0	56.7	77.4	77.5
3	✓	++		38.6	885 310.7	16928.0	58.7	77.4	77.5
3	✓		L	3.9	102 497.5	1320.0	52.6	77.2	77.3
3	✓		R	1.5	26 377.9	1380.0	51.9	76.8	76.9
3	✓	−	L	0.7	2656.4	1223.0	51.7	76.8	76.8

* Instance `brg180` reached the memory limit and was excluded from the reported calculations.
** Instances `brg180`, `portgen-1400-4`, and `nrw1379` reached the memory limit and were excluded from the reported calculations.

detected; (iv) using an MST based on edge length to build the cycle basis also appears to be slightly beneficial. Finally, combining all these strategies produces positive results as well, with an average computation time below one second and an average proportion of true SECs detected reaching 51.7%, and even 76.8% when combined with the truncated MST procedure.

From this point onward, the fine-tuned reduced-cost procedure and the truncated MST procedure with $\alpha = 10$ are used to compute the sets $\mathcal{P}_1$ and $\mathcal{P}_2$. (Note that this was already the case for the results presented in Table 8.) At this stage, it is important to highlight that the recall level achieved by any ML agent must be scaled by 0.768 to obtain its effective recall, since 23.2% of the SECs are not detected by our two procedures and therefore cannot possibly be classified as true SECs.

5.4 Input features

The final aspect of our ML component to address is the determination of input features, or in other words, the descriptive attributes associated with a given SEC $\mathcal{S}$ that would allow our ML agent to classify $\mathcal{S}$ as either a true or a false SEC. In the following, we describe the feature set used in our

study. The first three features are:

- Feature 1: whether $\mathcal{S}$ belongs to $\mathcal{P}_1$;
- Feature 2: whether $\mathcal{S}$ belongs to $\mathcal{P}_2$;
- Feature 3: the number of vertices contained in $\mathcal{S}$.

The remaining features depend on specific edge weights. In practice, for each of these features, we consider three types of weights w_e: the edge length l_e, the complement $1 - \tilde{x}_e$ (where $\tilde{x}_e$ is the value of the variable associated with edge e in the continuous relaxation of the DFJ formulation without SECs), and the corresponding reduced cost $\tilde{r}_e$.

The next five features are computed using the subgraph induced by $\mathcal{S}$ (i.e., the subgraph containing all edges for which both endpoints belong to $\mathcal{S}$), which we denote as $\mathcal{G}(\mathcal{S})$:

- Feature 4: the weight of an MST of $\mathcal{G}(\mathcal{S})$; we let $\mathcal{M}(\mathcal{S})$ denote the set of edges in this MST;
- Feature 5: the weight of a nearest-neighbor (NN) heuristic tour starting from a randomly chosen vertex of $\mathcal{G}(\mathcal{S})$; we let $\text{NN}(\mathcal{S})$ denote the set of edges in this tour;
- Feature 6: the total weight of the edges in $\mathcal{G}(\mathcal{S})$; we let $\mathcal{E}(\mathcal{S})$ denote the set of these edges;
- Feature 7: for each vertex $v \in \mathcal{S}$, we identify the minimum-weight edge incident to it—denoted by $e_1(v, \mathcal{S})$—and sum the weights of all such edges;
- Feature 8: for each vertex $v \in \mathcal{S}$, we identify the two minimum-weight edges incident to it—letting $e_2(v, \mathcal{S})$ denote the second of these two edges—and sum the weights of all edges $e_1(v, \mathcal{S})$ and $e_2(v, \mathcal{S})$;

The last three features are computed using both $\mathcal{G}(\mathcal{S})$ and the complete instance graph $\mathcal{G}$.

- Feature 9: for each vertex $v \in \mathcal{S}$, we identify the two minimum-weight edges incident to it that do not belong to $\mathcal{G}(\mathcal{S})$—denoted by $e_1(v, \bar{\mathcal{S}})$ and $e_2(v, \bar{\mathcal{S}})$, respectively—and sum the weights of all such edges;
- Feature 10: for each vertex $v \in \mathcal{S}$, we subtract the weight of $e_1(v, \bar{\mathcal{S}})$ from that of $e_1(v, \mathcal{S})$ and sum the resulting values;
- Feature 11: for each vertex $v \in \mathcal{S}$, we identify its K nearest neighbors—letting $\mathcal{K}(v)$ denote the set of these neighbors—and count how many vertices in $\mathcal{K}(v)$ do not belong to $\mathcal{S}$. These counts are then summed over all vertices in $\mathcal{S}$.

Each of the resulting $3 + 3 \times 8 = 27$ features is multiplied by two scaling factors—a form of normalization—to account for differences in SEC size, instances size, and instance type. The first scaling factor is feature-specific, while the second depends on the type of edge weight used. These features, along with their multiplicity (three for edge-weight-dependent features, one otherwise), values, and respective scaling factors, are summarized in Table 11. Regarding the weight-dependent scaling factor, it is set to 1 if $w_e = 1 - \tilde{x}_e$ and to $\frac{n}{\tilde{z}}$ if $w_e = \tilde{r}_e$ or $w_e = l_e$ (we recall that $\tilde{z}$ is the objective value of the continuous relaxation of the DFJ formulation without SECs).

Table 11: Summary of the input features used by the ML classifier.

ID	#	Name	Value	Scaling factor
1	1	In $\mathcal{P}_1$	1 if $\mathcal{S} \in \mathcal{P}_1$, 0 otherwise	1
2	1	In $\mathcal{P}_2$	1 if $\mathcal{S} \in \mathcal{P}_2$, 0 otherwise	1
3	1	Size	$\lvert\mathcal{S}\rvert$	$\frac{1}{n}$
4	3	MST weight	$\sum_{e \in \mathcal{M}(\mathcal{S})} w_e$	$\frac{1}{\lvert\mathcal{S}\rvert-1}$
5	3	NN tour weight	$\sum_{e \in \mathrm{NN}(\mathcal{S})} w_e$	$\frac{1}{\lvert\mathcal{S}\rvert}$
6	3	Total internal edge weight	$\sum_{e \in \mathcal{E}(\mathcal{S})} w_e$	$\frac{2}{\lvert\mathcal{S}\rvert \cdot (\lvert\mathcal{S}\rvert-1)}$
7	3	Sum of 1-min internal edge weights	$\sum_{v \in \mathcal{S}} w_{e_1(v,\mathcal{S})}$	$\frac{1}{\lvert\mathcal{S}\rvert}$
8	3	Sum of 2-min internal edge weights	$\sum_{v \in \mathcal{S}} \left(w_{e_1(v,\mathcal{S})} + w_{e_2(v,\mathcal{S})}\right)$	$\frac{2}{\lvert\mathcal{S}\rvert}$
9	3	Sum of 2-min external edge weights	$\sum_{v \in \mathcal{S}} \left(w_{e_1(v,\bar{\mathcal{S}})} + w_{e_2(v,\bar{\mathcal{S}})}\right)$	$\frac{2}{\lvert\mathcal{S}\rvert}$
10	3	Sum of 1-min internal minus 1-min external edge weights	$\sum_{v \in \mathcal{S}} \left(w_{e_1(v,\mathcal{S})} - w_{e_1(v,\bar{\mathcal{S}})}\right)$	$\frac{1}{\lvert\mathcal{S}\rvert}$
11	3	Sum of external K nearest neighbors	$\sum_{v \in \mathcal{S}} \lvert\mathcal{K}(v) \setminus \mathcal{S}\rvert$	$\frac{1}{\lvert\mathcal{S}\rvert \cdot K}$

6 Training and testing the ML component

In this section, we outline the training process for our ML agent. We first describe the dataset generation procedure, then discuss the architecture selection for the neural network and the methodology used to evaluate both training performance and feature importance. Finally, we assess the computational improvements achieved by the best implementation of the B&C framework after integrating the resulting ML component using the original 429 benchmark instances.

6.1 Creating the training dataset

We used the `portgen` and `portcgen` instance generators to create new instances for the dataset used to train our ML agent. For each $n \in \{100, 200, \ldots, 1000\}$, we created 50 instances with each generator. Each instance was then solved using **CT1-SM-OS** with a 12-hour time limit. If an instance was solved to optimality, then all SECs in $\mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{C}^*$ were included in the dataset along with their corresponding features; SECs in $\mathcal{C}^*$ were labeled 1, while the others were labeled 0. If an instance remained unsolved, then only the SECs actually added by **CT1-SM-OS** before the time limit was reached were included and labeled 1, as these are known to belong to $\mathcal{C}^*$. The other SECs from $\mathcal{P}_1 \cup \mathcal{P}_2$ were not included because their true labels are unknown; some might have been added by **CT1-SM-OS** later in the search, and thus would also belong to $\mathcal{C}^*$. In total, 801 out of the $2 \times 10 \times 50 = 1000$ instances could be solved within the time limit (466 from `portgen` and 335 from `portcgen`), resulting in a dataset with 23 866 515 rows, of which 0.46% were labeled 1.

This dataset, which we refer to as **merged** hereafter, is partitioned into four segments—**portgen-S**, **portgen-M**, **portcgen-S**, and **portcgen-M**—according to the instance generator used and the instance size ("S" if $n \leq 500$, and "M" otherwise). These segments are used to evaluate the generalizability of our ML agent.

6.2 Neural network architecture

Neural networks and their architectural variants form a class of supervised learning models widely used for classification tasks in both ML [36] and operations research applications [28]. Among these,

multilayer perceptrons (MLPs), in which the input data is processed through a sequence of layers, are often considered the simplest variant [5].

Following this approach, we evaluated five candidate architectures: a linear baseline (a single-layer sigmoid) and four MLPs with increasing depth and width, namely MLP(32, 16), MLP(32, 16, 8), MLP(256, 128, 64, 32), and MLP(512, 256, 128, 64, 32). For instance, MLP(32, 16) has two hidden layers: the first has 32 neurons and the second has 16. Whereas complex network architectures can learn more difficult patterns, they are also prone to overfitting and increased computational cost.

All models were trained with identical settings: an Adam optimizer [22] with a learning rate of 10^{-3}, 10 training epochs, and a batch size of 2^{15}. The MLP architectures used tanh activations for hidden layers and a sigmoid activation for the output layer. We estimated performance via 10-fold stratified cross-validation, with all models evaluated on identical splits to ensure a fair comparison. In each of the 10 iterations, the agent was trained on 9 folds (the training set) and generated predictions on the single held-out fold (the testing set). From these results, we computed the precision–recall (PR) curve and average precision (AP), the latter serving as a numerical estimate of the area under the PR curve. To deal with class imbalance, we applied fold-specific class weights. Letting n_0 and n_1 denote the number of samples labeled 0 and 1 in the training split, respectively, we assigned a weight of 1 to samples with label 1 and a weight of $\frac{n_1}{n_0}$ to samples with label 0.

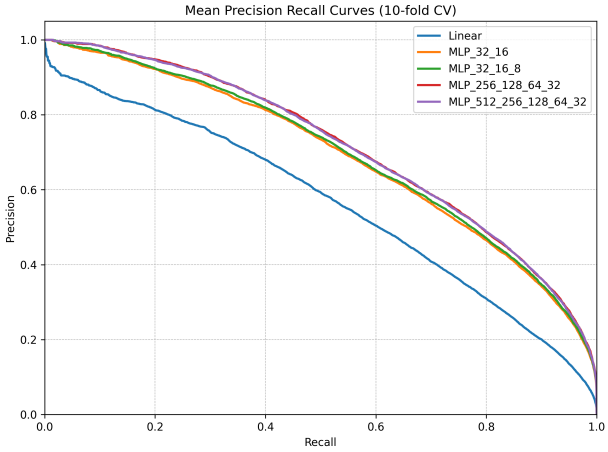
We show in Figure 4 the PR curve (averaged over the 10 folds) obtained by each network architecture when trained on the **portgen-S** segment (top) and the **merged** dataset (bottom). Since some PR curves are very close to one another, we also provide a zoomed-in view of the high-recall region on the right-hand side of the figure. Additionally, we report in Table 12 the mean AP for each architecture, along with the mean precision achieved when the recall is set to 0.99 and the mean recall achieved when the precision is set to $\frac{1}{7}$. For each measure, we also provide the standard deviation. These two values are derived from Table 8, where we observed that one should aim for a very high recall, but also that performance started to deteriorate when the precision dropped below $\frac{1}{7}$.

Table 12: Performance measures for each of the tested network architectures.

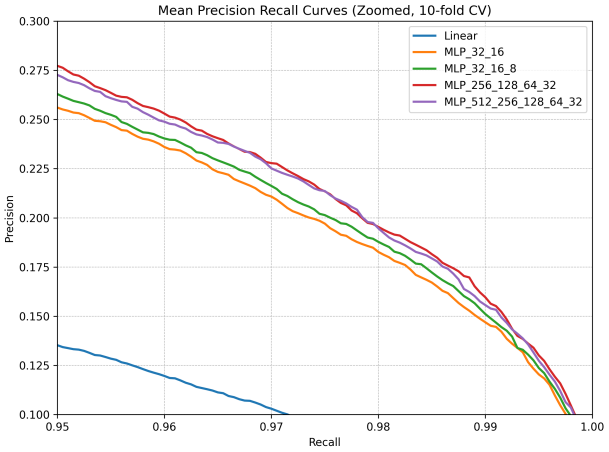
Architecture	portgen-S						merged					
	AP		P@R(0.99)		R@P(1/7)		AP		P@R(0.99)		R@P(1/7)	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Linear	0.5607	0.0498	0.0609	0.0249	0.9409	0.0249	0.6937	0.0074	0.1418	0.0040	0.9897	0.0006
MLP(32,16)	0.6898	0.0120	0.1469	0.0081	0.9910	0.0019	0.7646	0.0036	0.2118	0.0051	0.9975	0.0004
MLP(32,16,8)	0.6955	0.0095	0.1511	0.0081	0.9918	0.0014	0.7653	0.0037	0.2140	0.0035	0.9977	0.0003
MLP(256,128,64,32)	0.7133	0.0092	0.1596	0.0064	0.9930	0.0017	0.7655	0.0042	0.2143	0.0068	0.9979	0.0006
MLP(512,256,128,64,32)	0.7125	0.0099	0.1557	0.0042	0.9931	0.0012	0.7640	0.0030	0.2117	0.0089	0.9976	0.0006

While it is clear that the linear architecture underperforms compared to the MLPs, its results remain surprisingly strong, particularly on the **merged** dataset. Regarding the MLP architectures, performance improves marginally with each increase in size up to the MLP(256, 128, 64, 32) configuration. Beyond this threshold, further increases in network complexity appear to slightly reduce all performance measures on both **portgen-S** and **merged**, although the associated standard deviations suggest that these differences may not be statistically significant. Consequently, the MLP(256, 128, 64, 32) architecture is used for the neural network from this point onward.

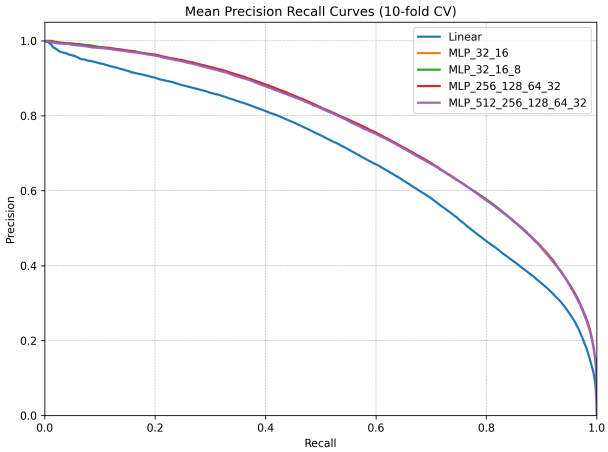
Figure 4: Averaged PR curve for each of the tested network architectures.



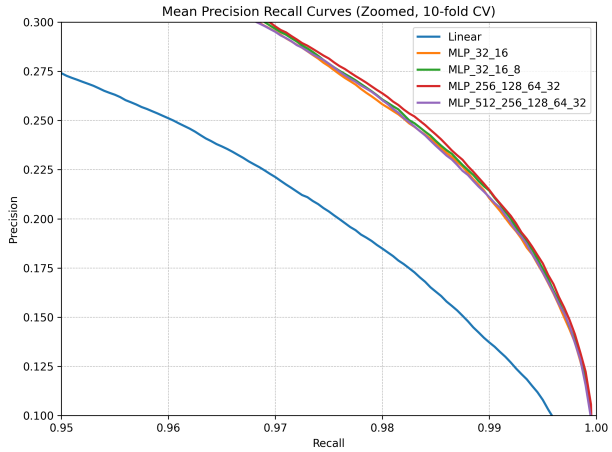
(a) **portgen-S**, zoomed-out view.



(b) **portgen-S**, zoomed-in view.



(c) **merged**, zoomed-out view.



(d) **merged**, zoomed-in view.

6.3 Training performance

With the network architecture fixed, we then investigated the generalizability of our ML agent. To this end, we partitioned each of the four segments into training and testing sets, both of which remained fixed throughout the subsequent experiments. We then trained several agents using the training set from one or more segments. These agents were then evaluated on the testing set of all segments. The results of these experiments are reported in Table 13, which specifies the segment(s) used to train each agent, the total number of training samples, and the three performance measures previously described that were achieved on each testing set (the highest value in each column is highlighted in bold).

We first investigate whether agents trained on a specific segment are the ones that perform best on that same segment. To ensure a fair comparison, given that the training sets differ in size, we trained the four segment-specific agents using the same number of samples, namely 2 800 000. As can be observed in the first four rows of the table, while performance differences between agents are relatively small, each segment-specific agent typically outperforms the other three on its respective segment across all three performance measures.

We then determine whether increasing the size of the training set further improves performance. To do so, we trained four additional segment-specific agents using (this time) all available samples

Table 13: Performance measures for agents trained on different segments.

training set					test set														
					portgen-S			portgen-M			portcgen-S			portcgen-M			merged		
portgen-S	**portgen-M**	**portcgen-S**	**portcgen-M**	**N** (10^6)	**AP**	**P@R(0.99)**	**R@P(1/7)**	**AP**	**P@R(0.99)**	**R@P(1/7)**	**AP**	**P@R(0.99)**	**R@P(1/7)**	**AP**	**P@R(0.99)**	**R@P(1/7)**	**AP**	**P@R(0.99)**	**R@P(1/7)**
✓				2.8	0.702	0.162	0.993	0.742	0.195	0.996	0.698	0.142	0.989	0.818	0.279	0.998	0.647	0.054	0.933
	✓			2.8	0.698	0.155	0.992	0.742	0.195	**0.997**	0.699	0.145	0.990	0.821	0.291	0.998	0.648	0.050	0.936
		✓		2.8	0.699	0.151	0.992	0.741	0.189	0.996	0.709	0.153	0.991	0.827	0.299	0.999	0.754	0.201	0.997
			✓	2.8	0.696	0.157	0.993	0.741	0.196	**0.997**	0.704	0.151	0.991	0.828	0.302	0.999	0.752	0.206	0.997
✓				3.4	0.706	**0.168**	0.994	0.747	0.202	0.996	0.703	0.157	0.992	0.825	0.293	0.998	0.654	0.057	0.941
	✓			9.1	0.707	0.163	0.993	0.749	0.205	**0.997**	0.707	0.146	0.991	0.826	0.295	0.998	0.650	0.042	0.921
		✓		2.9	0.694	0.162	0.992	0.735	0.195	**0.997**	0.701	0.152	0.991	0.821	0.292	0.999	0.749	0.203	**0.998**
			✓	3.7	0.703	0.167	0.994	0.745	0.204	**0.997**	0.709	0.153	0.992	0.830	0.302	0.999	0.757	0.209	**0.998**
✓	✓			12.5	0.708	**0.168**	0.992	0.750	0.207	**0.997**	0.707	0.158	0.992	0.828	0.304	0.998	0.650	0.034	0.918
		✓	✓	6.6	0.704	0.155	0.992	0.746	0.203	**0.997**	**0.717**	0.154	0.992	**0.833**	0.314	0.999	0.762	0.214	**0.998**
✓	✓	✓	✓	19.1	**0.712**	0.167	**0.995**	**0.751**	**0.212**	**0.997**	0.716	**0.173**	**0.994**	**0.833**	**0.323**	**1.000**	**0.764**	**0.222**	**0.998**

from their respective training sets. A pairwise comparison between the first and second groups of four rows confirms that larger training sets tend to improve performance, although the gains are modest. This suggests that further increases in the size of the training set would likely result in only marginal gains.

Next, we investigate whether expanding the training set by including additional segments further improves performance. To do so, we trained three more agents: one using the training sets of **portgen-S** and **portgen-M**, a second using the training sets of **portcgen-S** and **portcgen-M**, and a third using the training sets of all four segments. As shown in the last three rows of the table, the agent trained on all segments achieves the best performance overall, as well as on each individual segment. Based on these results, we chose to continue our experiments using the agent trained on all segments.

6.4 Feature importance

Whereas the performance of our ML agent thus far meets expectations (an effective recall of $0.998 \times 0.768 = 76.6\%$ at a precision of $\frac{1}{7}$), one may wonder which of the proposed features contribute most to the performance of the agent and whether all 27 features are necessary to achieve such results.

To answer these questions, we performed *recursive feature elimination*, which iteratively removes the "least relevant" feature until only one remains (or until the performance drop becomes significant, depending on the chosen stopping criterion). A natural way to measure the relevance of a given feature is to train the model without it and evaluate the resulting decrease in performance. Unfortunately, this approach is computationally expensive: since the number of models to train at each step equals the number of remaining features, it would require training $27 + 26 + \cdots + 1 = 378$ models in total (multiplied by a constant factor k when using k-fold stratified cross-validation).

Instead, we used *permutation feature importance* [35], which measures a feature's contribution to a pre-trained model and works as follows. Given a trained model, the contribution of a feature is estimated by randomly permuting its values in the validation set (thereby breaking the association between the feature and the label) and computing the resulting decrease in performance. The advantage of this approach is that only one model needs to be trained at each step, resulting in only 27

models in total (again multiplied by a constant factor k when using k-fold stratified cross-validation). The drawback, however, is that a feature's relevance to a pre-trained model does not perfectly correlate with the performance of a newly trained model that excludes that feature. Still, this approach allowed us to achieve competitive performance using considerably fewer features.

For each of the 27 iterations, we first trained an MLP(256,128,64,32) via 5-fold stratified cross-validation on the current feature subset. We then measured the importance of each remaining feature using the aforementioned procedure, repeating the permutation step 5 times per feature to reduce variance (resulting in $5 \times 5 = 25$ measurements per feature), and removed the feature with the lowest average relevance. We report in Figure 5 the average model performance (measured again as average AP, precision at recall 0.99, and recall at precision $\frac{1}{7}$) after each elimination step as a function of the number of retained features.

Interestingly, we observe that the first 14 features (up to and including 8.1) can be removed with little impact on model performance. This subset includes all features with IDs 1, 2, 3, 8, and 11. An additional 5 features (up to and including 6.2) can be removed with only a minor performance decrease (at this stage, all features with ID 9 have also been removed). Regarding the remaining 8 features, it is worth noting that they include all three weight types, all three features with ID 10, and two features with ID 6. Finally, we note that a model using only features 6.3 (the most important feature according to all three measures), 7.3, and 10.3 already achieves decent performance, highlighting the predictive power of the reduced costs for this task. Once again, we emphasize that a feature's lack of importance in a trained model does not imply a lack of intrinsic predictive value; rather, it suggests that the other features already capture the information provided by that feature.

6.5 Results on the benchmark instances

To conclude our experiments, we embedded the ML component into the B&C framework implementation **CT3-100-ASEF** and compared its performance with that of the non-augmented version (previously reported in Table 1) using the benchmark instances (which, it is worth reminding, were not used to train the ML agent). We evaluated six distinct ML components in total: three using the full set of 27 features (with attribute "27F" in the table), and three using only the 8 most important features ("8F"). For each subset of three components, we varied the threshold to achieve recall levels of 0.99, 0.999, and 0.9999, respectively ("99R", "999R", and "9999R"). The performance metrics for these six ML-enhanced B&C framework implementations on the 299 instances for which $\mathcal{C}^*$ is known are summarized in Table 14. For each dataset, we report the number of instances, the average size of $\mathcal{C}^*$, and the average number of SECs in $\mathcal{C}^*$ that also belong to $\mathcal{P}_1 \cup \mathcal{P}_2$ ("#det"). In addition, for each tested approach, we provide the number of instances solved to optimality, the average computation time, the average number of iterations, and the average number of SECs added. We further specify the average number of SECs added at the start of the framework by the ML agent ($\#\text{SEC}_F$) and the average number of these SECs that belong to $\mathcal{C}^*$ ("TP"). Finally, we report ML-related performance metrics, namely the average actual recall ("Re_A"), recall ("Re"), and precision ("Pr"). For the sake of comparison, we also report performance metrics obtained using the standard standalone Linux-compatible executable file of Concorde [9].

Table 14: Performance measures of the ML-enhanced versions of **CT3-100-ASEF** on the subset of 299 instances.

				Concorde		ML-27F-99R									ML-27F-999R									ML-27F-9999R								
Dataset	#inst	$\lvert\mathcal{C}^*\rvert$	#det	#opt	$T(s)$	#opt	$T(s)$	#iter	#SEC	#SEC$_F$	TP	Re$_A$	Re	Pr	#opt	$T(s)$	#iter	#SEC	#SEC$_F$	TP	Re$_A$	Re	Pr	#opt	$T(s)$	#iter	#SEC	#SEC$_F$	TP	Re$_A$	Re	Pr
TSPLIB	73	71.0	49.7	73	13	73	92	4.7	367.8	294.5	45.5	0.7	0.9	0.2	73	95	4.8	631.7	561.2	48.4	0.7	1.0	0.1	73	98	4.8	1102.8	1031.2	49.1	0.7	1.0	0.1
HardTSPLIB	31	96.1	89.3	23	343	27	902	22.0	159.7	11.6	10.7	0.2	0.2	1.0	27	892	20.5	147.8	26.5	22.1	0.3	0.3	0.8	28	901	16.0	159.2	78.2	53.6	0.6	0.7	0.7
portgen	116	173.1	135.6	116	91	113	609	6.3	942.3	783.9	134.4	0.8	1.0	0.2	113	601	6.2	1589.1	1434.6	135.5	0.8	1.0	0.1	112	625	6.3	2719.2	2567.0	135.6	0.8	1.0	0.1
portcgen	79	165.1	107.2	79	17	79	315	8.2	811.6	604.8	105.8	0.7	1.0	0.2	78	289	8.1	1301.2	1103.7	107.1	0.7	1.0	0.1	77	382	8.0	2139.4	1939.0	107.2	0.7	1.0	0.1
Total	299	138.1	102.3	291	73	292	435	8.0	686.4	537.0	92.3	0.7	0.9	0.3	291	425	7.9	1129.9	987.9	95.0	0.7	0.9	0.2	290	461	7.4	1906.0	1768.1	98.5	0.7	1.0	0.1

		CT3-100-ASEF				ML-8F-99R									ML-8F-999R									ML-8F-9999R								
Dataset	#inst	#opt	$T(s)$	#iter	#SEC	#opt	$T(s)$	#iter	#SEC	#SEC$_F$	TP	Re$_A$	Re	Pr	#opt	$T(s)$	#iter	#SEC	#SEC$_F$	TP	Re$_A$	Re	Pr	#opt	$T(s)$	#iter	#SEC	#SEC$_F$	TP	Re$_A$	Re	Pr
TSPLIB	73	72	142	7.1	187.0	73	94	4.7	412.3	341.0	46.1	0.7	0.9	0.1	73	99	4.9	689.8	617.1	48.2	0.7	1.0	0.1	73	90	4.8	1170.5	1102.9	49.0	0.7	1.0	0.0
HardTSPLIB	31	28	823	23.0	156.8	29	763	21.2	161.7	32.9	23.4	0.4	0.4	0.8	29	721	17.1	170.2	79.9	51.3	0.6	0.7	0.7	29	651	8.5	185.8	150.1	80.6	0.9	0.9	0.5
portgen	116	111	829	9.6	482.8	113	581	6.3	1012.5	853.8	134.4	0.8	1.0	0.2	114	589	6.2	1657.4	1506.3	135.5	0.8	1.0	0.1	112	646	6.2	2858.4	2707.0	135.6	0.8	1.0	0.1
portcgen	79	75	522	11.5	493.9	78	354	8.3	862.3	659.2	105.8	0.7	1.0	0.2	79	347	8.1	1365.4	1166.1	107.1	0.7	1.0	0.1	78	331	8.1	2280.1	2081.0	107.2	0.7	1.0	0.0
Total	299	286	580	10.9	379.7	293	421	8.0	738.1	592.1	93.8	0.7	0.9	0.2	295	419	7.5	1189.8	1051.5	97.9	0.7	1.0	0.2	292	427	6.6	2016.4	1884.9	101.3	0.8	1.0	0.1

First and foremost, we observe that all six ML-enhanced versions of **CT3-100-ASEF** outperform the non-augmented version. It is also interesting to note that, for a fixed recall value, the ML component using 8 features outperforms that using all 27 features. This suggests that the latter may overfit the training data, an observation particularly evident when comparing the low recall levels of the **ML-27F** approaches on the HardTSPLIB dataset to those of the **ML-8F** approaches. Similarly, for a fixed number of features, the ML component with a recall set to 0.999 outperforms those set to 0.99 and 0.9999. This indicates that, while starting the framework with as many true SECs as possible is important, there is a tipping point where finding additional true SECs requires adding a disproportionately large number of false SECs, which becomes detrimental to the overall performance.

Regarding dataset-specific measures, **ML-27F-999R** performs particularly well on portcgen (-44% computing time compared to **CT3-100-ASEF**), TSPLIB (-33%), and portgen (-28%) instances, but less so on HardTSPLIB instances (+8%). Comparatively, the performance of **ML-8F-999R** is similar to that of **ML-27F-999R** for TSPLIB and portgen instances, slightly worse for portcgen instances, and better for HardTSPLIB instances.

Finally, we observe that Concorde remains undefeated, with a very low average computation time (73s compared to 419s for our best approach). While it solves fewer HardTSPLIB instances than **ML-8F-999R**, its average computation time even for that subset remains the lowest. Furthermore, we must highlight that the executable file of Concorde uses QSopt, a linear programming solver that does not perform as well as Gurobi.

We now report the performance metrics for the ML-enhanced frameworks on the full set of 429 instances in Table 15. In addition to the metrics previously introduced, we also report the average computation time calculated only over the instances that were solved by at least one of the tested ML-enhanced approaches (column "T(s)$_{\text{opt}\geq 1}$").

Table 15: Performance measures of the ML-enhanced versions of **CT3-100-ASEF** on all 429 instances.

		Concorde			ML-27F-99R					ML-27F-999R					ML-27F-9999R				
Dataset	#inst	#opt	$T(s)$	$T(s)_{\text{opt}\geq 1}$	#opt	$T(s)$	$T(s)_{\text{opt}\geq 1}$	#iter	#SEC	#opt	$T(s)$	$T(s)_{\text{opt}\geq 1}$	#iter	#SEC	#opt	$T(s)$	$T(s)_{\text{opt}\geq 1}$	#iter	#SEC
TSPLIB	89	87	130	15	78	558	217	5.2	1628.2	80	563	221	5.4	3287.9	79	572	231	5.3	4824.1
HardTSPLIB	40	30	1111	672	27	1509	986	20.4	186.7	27	1501	976	18.7	176.6	28	1508	986	15.3	250.0
portgen	150	150	319	101	115	1282	774	6.5	1150.2	117	1256	741	6.5	1937.8	115	1288	781	6.5	3315.4
portcgen	150	150	71	38	114	1344	936	8.4	1290.0	112	1342	934	8.6	2110.2	113	1379	977	8.5	3481.7
Total	429	417	267	110	334	1175	726	8.2	1208.4	336	1165	714	8.1	2114.0	335	1192	746	7.8	3400.7
		CT3-100-ASEF			ML-8F-99R					ML-8F-999R					ML-8F-9999R				
Dataset	#opt	$T(s)$	$T(s)_{\text{opt}\geq 1}$	#iter	#SEC	#opt	$T(s)$	$T(s)_{\text{opt}\geq 1}$	#iter	#SEC	#opt	$T(s)$	$T(s)_{\text{opt}\geq 1}$	#iter	#SEC	#opt	$T(s)$	$T(s)_{\text{opt}\geq 1}$	#iter	#SEC
TSPLIB	76	619	284	7.0	243.3	78	575	235	5.1	1643.3	77	593	255	5.4	3511.5	79	572	231	5.3	4824.1
HardTSPLIB	28	1448	910	21.6	186.4	29	1401	851	18.9	203.9	31	1232	640	17.4	285.1	28	1314	743	15.3	250.0
portgen	112	1457	986	9.6	585.1	116	1251	735	6.4	1230.7	121	1247	731	6.4	2014.1	115	1301	796	6.5	3315.4
portcgen	103	1625	1268	11.5	702.6	108	1475	1090	8.5	1354.8	108	1402	1004	8.4	2173.2	113	1463	1076	8.5	3481.7
Total	319	1341	923	10.9	518.1	331	1203	760	8.0	1264.0	337	1164	713	7.9	2219.2	335	1207	765	7.8	3400.7

Most of the comments made for the previous table also apply here, namely that: (i) all six ML-enhanced versions of **CT3-100-ASEF** outperform the non-augmented version, (ii) for a fixed number of features, the ML component with a recall set to 0.999 outperforms those set to 0.99 and 0.9999, (iii) **ML-27F-999R** performs particularly well on portcgen (-26% computing time compared to **CT3-100-ASEF** on the instances solved to optimality by at least one of the ML-enhanced approaches), portgen (-25%), and TSPLIB (-22%) instances, but less so on HardTSPLIB instances (+7%), (iv) the performance of **ML-8F-999R** is better than that of **ML-27F-999R** on HardTSPLIB instances, and (v) Concorde remains undefeated.

To conclude this section, it makes sense to compare the actual performance obtained by the ML component (Table 15) with the expected results (Table 8). With an average actual recall of 0.7 and precision of 0.2, we expected an average decrease in computation time between 25 and 32%. Both for **ML-27F-999R** and **ML-8F-999R**, however, the average reduction was approximately 23%. Such differences compared to our expectations can be explained by heterogeneity across datasets (for example, **ML-27F-999R** achieved the expected time reduction levels for portgen and portcgen instances, whereas **ML-8F-999R** did so for HardTSPLIB instances), but also within datasets. Indeed, the experiments in Table 8 assume the same recall and precision levels for all instances, whereas we observed significant variation in practice (for example, actual recall levels typically ranged between 0 and 1, largely due to the varying ability of our two procedures to generate a high-quality pool of SEC candidates $\mathcal{P}_1 \cup \mathcal{P}_2$). While minor time improvements are expected when the recall approaches 1, more significant deteriorations are expected when the recall decreases.

7 Conclusions

We developed in this study a machine learning (ML) component that predicts the necessity of specific subtour elimination constraints (SECs) when solving the symmetric traveling salesman problem (TSP) via a standard branch-and-cut (B&C) framework. After completing the necessary intermediate steps, including creating a pool of SEC candidates, generating labeled data, engineering input features, training the ML agent, and testing it on benchmark instances, we demonstrated that using our component to pre-populate the set of SECs before initiating the B&C process marginally increases the number of instances solved to optimality, while also reducing the average computation time needed to solve these instances by 23%. For certain versions of the component, this reduction in computation time holds even for instances with structures differing from those encountered during

the training phase.

From a problem perspective, our work empirically compares various B&C implementations for the TSP and highlights substantial performance variations among them. Furthermore, we address the under-explored questions of determining: (i) the minimum number of SECs required in the DFJ formulation for a given TSP instance to ensure that no optimal solution contains subtours, and (ii) the maximum number of SECs that a given B&C implementation might add during its execution. We also provide a timely update on the computational limits of the tested B&C implementations, showing that, when combined with a modern integer linear programming (ILP) solver, they can solve metric TSP instances with up to 1500 vertices to optimality. From an optimization perspective, we demonstrated that ML-based components can benefit exact solution methods as well, whereas the vast majority of prior research has focused on their integration into (or their use to replace) heuristic solutions.

Regarding future research, we believe that developing further approaches to pre-populate the set of SECs before launching a B&C framework is a relevant direction. Such frameworks are relatively easy to implement (especially compared to highly specialized solvers like Concorde), and we showed that computation time reductions of up to 65% are achievable. To continue on this path, additional methods are needed to generate a broader pool of SEC candidates, as our current procedures capture only 76.8% of the required SECs on average. Even if one managed to increase that value to 100%, an achievement that would theoretically always allow the B&C framework to terminate after a single iteration, we observed that even that single iteration could be completed faster by incorporating additional SECs that are, a priori, unneeded.

Finally, we do acknowledge that our primary expertise lies in the field of operations research; therefore, there is significant potential to further enhance the ML aspects of our component. This could be achieved by incorporating additional input features, exploring alternative neural network architectures (or even different supervised learning paradigms), and performing systematic hyperparameter tuning, among other refinements. Naturally, while we believe our results for the TSP are noteworthy, it is likely more impactful to extend the proposed approach to other routing problems. Indeed, the performance level of exact approaches for many (if not all) such problems remains far below the standards set by Concorde for the TSP.

Acknowledgments

This work used the Dutch national e-infrastructure with the support of the SURF Cooperative using grant no. EINF-13596.

References

[1] M.M. Aguayo, S.C. Sarin, and H.D. Sherali. Solving the single and multiple asymmetric traveling salesmen problems by generating subtour elimination constraints from integer solutions. *IISE Transactions*, 50:45–53, 2018.

[2] F. Akçay and M. Delorme. Solving the parallel processor scheduling and bin packing problems with contiguity constraints: mathematical models and computational studies. *European Journal of Operational Research*, 323:701–723, 2025.

[3] D.L. Applegate, R.E. Bixby, V. Chvátal, and W.J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[4] M.A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57:75–94, 2005.

[5] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290:405–421, 2021.

[6] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. *Operations Research Forum*, 3:20, 2022.

[7] F. Clautiaux and I. Ljubić. Last fifty years of integer linear programming: a focus on recent practical advances. *European Journal of Operational Research*, 324:707–731, 2024.

[8] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 2 edition, 1977.

[9] W. Cook. Concorde TSP Solver. `https://www.math.uwaterloo.ca/tsp/concorde/index.html`. Accessed 10 January 2025.

[10] W.J. Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, 2012.

[11] G.W. Corder and D.I. Foreman. *Nonparametric Statistics: A Step-by-Step Approach*. John Wiley & Sons, 2 edition, 2014.

[12] H. Crowder and M.W. Padberg. Solving large-scale symmetric travelling salesman problems to optimality. *Management Science*, 26:495–509, 1980.

[13] H. Dai, E.B. Khalil, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, volume 30, page 6351–6361, 2017.

[14] 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.

[15] L.M. Gambardella and M. Dorigo. Ant-Q: A reinforcement learning approach to the traveling salesman problem. In A. Prieditis and S. Russell, editors, *Machine Learning Proceedings 1995*, pages 252–260. Elsevier, 1995.

[16] M. Grötschel and O. Holland. Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming*, 51:141–202, 1991.

[17] G. Gutin and A.P. Punnen. *The Traveling Salesman Problem and Its Variations*. Springer, 2006.

[18] D.S. Johnson and L.A. McGeoch. 8th DIMACS Implementation Challenge: The Traveling Salesman Problem. website. Available at `http://archive.dimacs.rutgers.edu/Challenges/TSP/`.

[19] D.S. Johnson and L.A. McGeoch. Experimental analysis of heuristics for the STSP. In *The traveling salesman problem and its variations*, pages 369–443. Springer, 2002.

[20] C.K. Joshi, T. Laurent, and X. Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019. available at: `https://arxiv.org/pdf/1906.01227`.

[21] R.M. Karp. *Reducibility among combinatorial problems*. Springer, 2010.

[22] D.P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[23] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. J. Wiley, 1985.

[24] B.C. Lee. Algorithmic approaches to circuit enumeration problems and applications. Technical report, Cambridge, Mass.: Massachusetts Institute of Technology, Dept. of, 1982.

[25] J.D.C. Little, K.G. Murty, D.W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.

[26] P. Miliotis. Integer programming approaches to the travelling salesman problem. *Mathematical Programming*, 10:367–378, 1976.

[27] 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.

[28] M. Morabit, G. Desaulniers, and A. Lodi. Machine-learning–based column selection for column generation. *Transportation Science*, 55:815–831, 2021.

[29] K.P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[30] U. Pferschy and R. Staněk. Generating subtour elimination constraints for the TSP from pure integer solutions. *Central European Journal of Operational Research*, 25:231–260, 2017.

[31] M. Prates, P.H.C. Avelar, H. Lemos, L.C. Lamb, and M.Y. Vardi. Learning to solve NP-complete problems: A graph neural network for decision TSP. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4731–4738, 2019.

[32] G. Reinelt. TSPLIB95. website. Available at `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`.

[33] G. Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer, 1994.

[34] R. Roberti and P. Toth. Models and algorithms for the asymmetric traveling salesman problem: an experimental comparison. *EURO Journal on Transportation and Logistics*, 1:113–133, 2012.

[35] Scikit-learn developers. Permutation Feature Importance. `https://scikit-learn.org/stable/modules/permutation_importance.html`, 2026. Accessed 13 March 2026.

[36] L. Shen, L.R. Margolies, J.H. Rothstein, E. Fluder, R. McBride, and W. Sieh. Deep learning to improve breast cancer detection on screening mammography. *Scientific Reports*, 9(1):12495, 2019.

[37] M. Sokolova and G. Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing and Management*, 45(4):427–437, 2009.

[38] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[39] E. Vercesi and A. Buchanan. The Dantzig-Fulkerson-Johnson TSP formulation is easy to solve for few subtour constraints. Technical report, Universita della Svizzera italiana, 2025. Available at: `https://optimization-online.org/2024/09/the-dantzig-fulkerson-johnson-tsp-formulation-is-easy-to-solve-for-few-subtour-constraints/`.

[40] E. Vercesi, S. Gualandi, M. Mastrolilli, and L.M. Gambardella. HARDTSPLIB. website. Available at `https://github.com/eleonoravercesi/HardTSPLIB`.

[41] E. Vercesi, S. Gualandi, M. Mastrolilli, and L.M. Gambardella. On the generation of metric TSP instances with a large integrality gap by branch-and-cut. *Mathematical Programming Computation*, 15:389–416, 2023.

[42] T.Q.T. Vo, M. Baiou, V.H. Nguyen, and P. Weng. Improving subtour elimination constraint generation in branch-and-cut algorithms for the TSP with machine learning. In *International Conference on Learning and Intelligent Optimization*, pages 537–551, 2023.

Figure 5: Model performance as a function of the number of retained features. Subplots show mean AP (top), mean precision at 0.99 recall (middle), and mean recall at precision $\frac{1}{7}$ (bottom).