# Learning to approximate industrial problems by operations research classic problems

Axel Parmentier

# Learning to approximate industrial problems by operations research classic problems

Axel Parmentier*

December 5, 2019

**Abstract**

Practitioners of operations research often consider difficult variants of well-known optimization problems, and struggle to find a good algorithm for their variants while decades of research have produced highly efficient algorithms for the well-known problems. We introduce a "machine learning for operations research" paradigm to build efficient heuristics for such variants of well-known problems. If we call the difficult problem of interest the hard problem, and the well known one the easy problem, we can describe our paradigm as follows. First, use a machine learning predictor to turn an instance of the hard problem into an instance of the easy one, then solve the instance of the easy problem, and finally retrieve a solution of the hard problem from the solution of the easy one.

Using this paradigm requires to learn the predictor that transforms an instance of the hard problem into an instance of the easy one. We show that the problem of learning such a predictor from a training set containing instances of the hard problem and their optimal solutions can be reformulated as a structured learning problem, whose structured prediction problem is the easy problem. This provides algorithms to learn our predictor if the easy problem has been considered as a structured prediction problem in the literature, and a methodology to build the learning algorithm if not.

We illustrate our paradigm and learning methodology on path problems. To that purpose, we introduce a maximum likelihood technique to train a structured prediction model which uses a shortest path problem as prediction problem. Using our paradigm, this enables to approximate an arbitrary path problem on an acyclic digraph (the hard problem) by a usual shortest path problem (the easy problem). Since path problems play an important role as pricing subproblems of column generation approaches, we also introduce matheuristics that leverage our approximations in that context. Numerical experiments show their efficiency on two stochastic vehicle scheduling problems.

**Keywords:** Machine Learning for Operations Research, Structured Learning, Path problem, Column Generation, Matheuristic, Stochastic Vehicle Scheduling Problem

## 1 Introduction

### 1.1 A novel "ML for OR" paradigm

In the past few years, machine learning (ML) techniques have become increasingly popular to speed-up the resolution of operations research (OR) problems. Consider a generic optimization problem

$$\min_{x \in \mathcal{X}(\Gamma)} f(x; \Gamma),$$

*Ecole des Ponts Paristech, axel.parmentier@enpc.fr

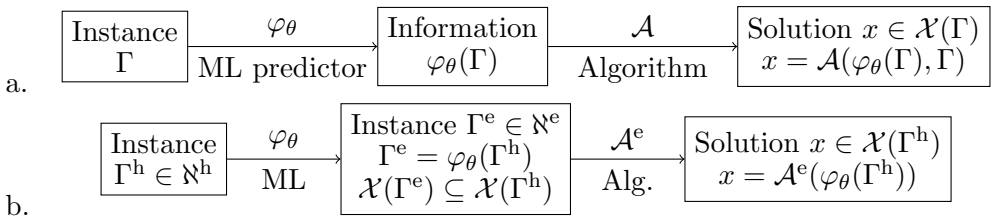


Figure 1: a. General scheme of ML approaches to OR problems. b. The paradigm we propose.

where $\Gamma$ is an instance of the problem, $\mathcal{X}(\Gamma)$ is the set of feasible solutions of $\Gamma$, and $x \mapsto f(x;\Gamma)$ the objective function of $\Gamma$. In contrast with the common use, we mention the instance $\Gamma$ explicitely. This will be handy because ML schemes typically consider a set of instances simultaneously to extract information that is relevant for any instance of the problem and not just one instance. Figure 1.a illustrates the general scheme of algorithms that exploit ML techniques to solve OR problems: First, an ML predictor $\varphi_\theta$ extracts relevant information $\varphi_\theta(\Gamma)$ on instance $\Gamma$, and second an optimization algorithm $\mathcal{A}$ uses this information to solve the problem. The ML predictor $\varphi_\theta$ belongs to a family $(\varphi_\theta)_{\theta\in\Theta}$, and the objective of the learning algorithm is to find the parameter $\theta$ in $\Theta$ that leads to the best performance of the algorithm $\mathcal{A}$ on the set of instances of interest. The methodologies identified by Bengio et al. [5] in their survey differ by the kind of ML predictor $\varphi_\theta$ and algorithm $\mathcal{A}$ they use. For instance, end-to-end learning approaches use a deep neural network as $\varphi_\theta$ and a simple greedy heuristic as $\mathcal{A}$, while other approaches use a simple feature based ML predictor $\varphi_\theta$ to find a good parametrization $\varphi_\theta(\Gamma)$ of an advanced combinatorial optimization solver $\mathcal{A}$.

In this work, we consider the following situation, which we have frequently encountered in the practice of OR. We want to solve large instances of a "hard" problem

$$\min_{x\in\mathcal{X}^{\mathrm{h}}(\Gamma^{\mathrm{h}})} f^{\mathrm{h}}(x;\Gamma^{\mathrm{h}}), \tag{h}$$

and we have an algorithm $\mathcal{A}^{\mathrm{h}}$ that can solve only moderate size instances of our problem. But our hard problem is a variant of an "easy" problem

$$\min_{x\in\mathcal{X}^{\mathrm{e}}(\Gamma^{\mathrm{e}})} f^{\mathrm{e}}(x;\Gamma^{\mathrm{e}}), \tag{e}$$

that is, a problem for which we know an efficient solution algorithm $\mathcal{A}^{\mathrm{e}}$ that can handle large instances. Again, we denote by $\Gamma^{\mathrm{h}}$ and $\Gamma^{\mathrm{e}}$ instances of the hard and the easy problem, respectively, by $\mathcal{X}^{\mathrm{h}}(\Gamma^{\mathrm{h}})$ and $\mathcal{X}^{\mathrm{e}}(\Gamma^{\mathrm{e}})$ their feasible solutions, and by $f^{\mathrm{h}}(\cdot;\Gamma^{\mathrm{h}})$ and $f^{\mathrm{e}}(\cdot;\Gamma^{\mathrm{e}})$ their objective functions. Let $\aleph^{\mathrm{h}}$ and $\aleph^{\mathrm{e}}$ be the set of instances of the hard and the easy problems, respectively.

To address that situation, we propose a novel "ML for OR" paradigm that uses the moderately efficient algorithm $\mathcal{A}^{\mathrm{h}}$ to learn a practically efficient heuristic for the hard problem of interest (h). It is illustrated on Figure 1.b and could be summarized as follows.

*Learn to approximate instances $\Gamma^{\mathrm{h}}$ of the hard problem (h) by instances of the easy problem (e).*

More precisely, we suggest to use a ML predictor $\varphi_\theta : \aleph^{\mathrm{h}} \to \aleph^{\mathrm{e}}$ to turn an instance $\Gamma^{\mathrm{h}}$ of the hard problem into an instance $\Gamma^{\mathrm{e}}$ of the easy problem (e) such that the set of feasible solutions $\mathcal{X}^{\mathrm{e}}(\Gamma^{\mathrm{e}})$ is contained in the set of feasible solutions $\mathcal{X}^{\mathrm{h}}(\Gamma^{\mathrm{h}})$ of $\Gamma^{\mathrm{h}}$. Then, we suggest to use the efficient algorithm $\mathcal{A}^{\mathrm{e}}$ to find the optimal solution of $\Gamma^{\mathrm{e}}$, and to return it as the solution of the

instance $\Gamma^{\mathrm{h}}$ of the hard problem. Of course, *if we want this paradigm to work in practice on a hard problem* (h)*, we must choose an easy problem* (e) *such that it is possible to approximate well an instance of* (h) *by an instance of* (e).

The main question that must be answered to make this approach work is the following one: *How to choose $\theta$ in such a way that $\mathcal{A}^{\mathrm{e}}$ applied on $\varphi_\theta(\Gamma^{\mathrm{h}})$ provides a good solution of $\Gamma_h$?*

We propose a general methodology which addresses the question of choosing $\theta$ by reformulating it as a structured learning problem. Structured learning is a branch of supervised learning where predictions are done by solving a structured prediction problem. In our case, the structured prediction problem corresponds to the easy problem (e). Structured learning theory provides two approaches to formulate the structured learning problem as an optimization problem: one based on the maximum likelihood principle, and the other one on loss minimization. In both case, the structure of the optimization problem obtained highly depends on the structured prediction problem used. Ad-hoc algorithms must therefore be developed for each type of easy problem used. Structured prediction problems for which structured learning algorithms exist notably include MAP problems in probabilistic graphical models, minimum weight matching in bipartite graphs, and some clustering problems [21]. In this paper, we illustrate our methodology and propose structured learning algorithms in the case of path problems.

### 1.2 Easy and hard problems considered in the paper

**Path problems** Let $D = (V, A)$ be an acyclic digraph with vertices $v$ in $V$ and arcs $a$ in $A$. Let $o$ and $d$ be origin and destination vertices, respectively, and $\mathcal{P}_{od}$ the set of $o$-$d$ paths in $D$. Finally let $c$ be a mapping from $\mathcal{P}_{od}$ to $\mathbb{R} \cup \{+\infty\}$. We will consider as hard problem the following generic path problem on an acyclic digraph

$$\min_{P \in \mathcal{P}_{od}} c(P). \tag{1}$$

This generic path problem notably encompasses difficult problems such as resource constrained shortest path problems or stochastic path problems as special cases. We introduce algorithms to approximate it by a usual shortest path problem

$$\min_{P \in \mathcal{P}_{od}} \hat{c}_a, \tag{2}$$

which can efficiently be solved by dynamic programming along a topological order.

**Path partition problems** Many vehicle scheduling problems (VSP) can be modeled using the set partitioning formulation

$$\min_{z} \quad \sum_{P \in \mathcal{P}_{od}} c(P) z_P, \tag{3a}$$

$$\text{s.t.} \quad \sum_{P \ni v} z_P = 1, \qquad \forall v \in V \backslash \{o, d\}, \tag{3b}$$

$$z_P \in \{0, 1\}, \qquad \forall P \in \mathcal{P}_{od}, \tag{3c}$$

where $D = (V, A)$ is an acyclic directed graph, $V$ is its set of vertices, and $A$ its set of arcs. $o$ is an *origin* vertex, $d$ a *destination* vertex, and $\mathcal{P}_{od}$ is the set of $o$-$d$ paths. Given an $o$-$d$ path $P$, we denote by $c(P)$ its cost in $\mathbb{R} \cup \{+\infty\}$. Typically, $V \backslash \{o, d\}$ is a set of tasks $v$ with fixed begin and end times that must be operated. The set $A$ contains all pairs $(u, v)$ such that a vehicle can operate $v$ after $u$. An $o$-$v$ path $P = o, v_1, \ldots, v_k, d$ in $\mathcal{P}_{od}$ models a sequence of tasks $v_1, \ldots, v_k$ that can be operated by a vehicle. Note that the sequence of tasks corresponding to

some paths $P$ may not be feasible. For instance, the number of hours a driver can work on a given day may be limited. We model this with a cost $c(P)$ equal to $+\infty$. The binary variable $z_P$ is equal to 1 if and only if a vehicle operates the sequence of tasks encoded by $P$.

Since the number of *o-d* paths is exponentially large, the mixed integer linear program (MILP) (3) contains too many variables to be solved directly using a MILP solver. It is therefore solved using a column generation approach. Such approaches alternate the resolution of the linear relaxation of (3) restricted to a subset of paths or columns $\mathcal{P}' \subset \mathcal{P}_{od}$ of tractable size, and the resolution of the *pricing subproblem*, which generates column(s) that should be added to $\mathcal{P}'$. Typically, if we denote by $\boldsymbol{\nu} = (\nu_v)_{v \in V \setminus \{o,d\}}$ the dual variables[1] associated with Constraint (3b) and relax Constraint (3c), we obtain the *pricing subproblem*

$$\min \Big\{ c(P) - \sum_{v \in P} \nu_v \colon P \in \mathcal{P}_{od} \Big\}. \tag{4}$$

A column generation approach works well only if we can solve the pricing subproblem efficiently. Since the column generation algorithm only requires to find a column $P$ of negative reduced cost $c(P) - \sum_{v \in P} \nu_v$, the pricing subproblem can be solved approximately. We propose to address the case where we can solve only instances of (4) of moderate size using our paradigm. We consider the pricing subproblem as the hard path problem (1) and approximate it by a usual shortest path problem (2). We also assume that, given a path $P$ in $\mathcal{P}_{od}$, we can efficiently compute its cost $c(P)$.

### 1.3 Running examples on which the numerical experiments are performed

We now introduce two problems on which we will illustrate our methodology: the *stochastic vehicle scheduling problem* (stochastic VSP), and the *chance constrained capacitated vehicle scheduling problem* (chance constrained CVSP).

Using our formalism, we only need to define the digraph $D$ and the cost $c(P)$ for each *o-d* path $P$ to define a problem. Our two problems are on the same digraph. For each task $v$ in $V$, we suppose to have a scheduled begin time $t_v^{\mathrm{b}}$ in $\mathbb{Z}_+$ and a scheduled end time $t_v^{\mathrm{e}}$ in $\mathbb{Z}_+$. We suppose $t_v^{\mathrm{e}} > t_v^{\mathrm{b}}$ for each task $v$ in $V$. For each pair of task $(u, v)$, the travel time to reach task $v$ from task $u$ is $t_{(u,v)}^{\mathrm{tr}}$. Task $v$ can be scheduled after task $u$ on a path only if

$$t_v^{\mathrm{b}} \geq t_u^{\mathrm{e}} + t_{(u,v)}^{\mathrm{tr}}. \tag{5}$$

There is an arc between $u$ and $v$ only if (5) is satisfied. There is also an *o-v* arc and a *v-d* arc for each $v$ in $V \setminus \{o, d\}$. The fact that $t_v^{\mathrm{e}} > t_v^{\mathrm{b}}$ ensures that $D$ is acyclic. We now explain how $c(P)$ is defined in each problem.

**Stochastic VSP.** In the stochastic VSP, we consider tasks that may be delayed. When a vehicle is delayed on a given task, it propagates delay to the next task it is supposed to operate. The objective is to minimize the total cost of delay. A typical application of the problem is *aircraft routing*, which aims at building the sequences of flight legs operated by the airplanes of an airline. Tasks correspond to flight legs, vehicles to airplanes, and the objective is to build the sequences operated by airplanes so as to minimize the total cost of delay.

More formally, we define the *stochastic VSP* as follows. Let $\Omega$ be a set of scenarios. For each task $v$, we have a random begin time $\xi_v^{\mathrm{b}}$ and a random end time $\xi_v^{\mathrm{e}}$, and for each arc $(u, v)$, we have a random travel time $\xi_{(u,v)}^{\mathrm{tr}}$. Hence, $\xi_v^{\mathrm{b}}(\omega)$, $\xi_v^{\mathrm{e}}(\omega)$, and $\xi_{(u,v)}^{\mathrm{tr}}(\omega)$ are respectively the begin

[1] Duals $\nu_v$ are defined only for $v \in V \setminus \{o, d\}$. For ease of notations, we complete $\boldsymbol{\nu}$ with $\nu_o$ and $\nu_d$, which are both defined to be equal to 0.

time of $v$, end time of $v$, and travel time between $u$ and $v$ under scenario $\omega$. We define $\xi^{\mathrm{e}}_o = 0$ and $\xi^{\mathrm{b}}_d = +\infty$.

Given an $o$-$v$ path $P$, we define recursively the end-time $\tau_P$ of $P$ as follows.

$$\tau_P = \begin{cases} 0, & \text{if } P \text{ is the empty path in } o, \\ \xi^{\mathrm{e}}_v + \max(\tau_Q + \xi^{\mathrm{tr}}_a - \xi^{\mathrm{b}}_v, 0), & \text{if } P = Q + a \text{ for some path } Q \text{ and arc } a. \end{cases} \tag{6}$$

Equation (6) models that fact that a task can be operated by a vehicle only when the vehicle has finished the previous task: The vehicle finishes $Q$ at $\tau_Q$, and arrives in $v$ at $\tau_Q + \xi^{\mathrm{tr}}_a$ with delay $\max(\tau_Q + \xi^{\mathrm{tr}}_a - \xi^{\mathrm{b}}_v, 0)$. The total delay $\Delta_P$ along a path $P$ is therefore defined recursively by

$$\Delta_P = \begin{cases} 0, & \text{if } P \text{ is the empty path in } o, \\ \Delta_Q + \max(\tau_Q + \xi^{\mathrm{tr}}_a - \xi^{\mathrm{b}}_v, 0), & \text{if } P = Q + a \text{ for some path } Q \text{ and arc } a. \end{cases} \tag{7}$$

Finally, we define the cost of an $o$-$d$ path $P$ as

$$c(P) = c^{\mathrm{veh}} + c^{\mathrm{del}}\mathbb{E}(\tau_P) \tag{8}$$

where $c^{\mathrm{veh}}$ in $\mathbb{Z}_+$ is the cost of a vehicle and $c^{\mathrm{del}}$ in $\mathbb{Z}_+$ is the cost of a unit delay. Practically, we use a finite set of scenarios $\Omega$, and compute the expectation as the average on this set.

**Chance constrained CVSP.** The chance constrained CVSP consists in picking-up goods of size $\rho_v$ with vehicles of capacity $R$ in $\mathbb{Z}_+$. If the capacity of the vehicle remains constant, the goods size $\rho_v$ is random. The cost of an $o$-$d$ path $P$ is

$$c(P) = \begin{cases} c^{\mathrm{veh}}, & \text{if } \mathbb{P}\Big(\sum_{v \in P} \rho_v \leq R\Big) \geq 1 - \alpha, \\ +\infty, & \text{otherwise,} \end{cases} \tag{9}$$

where again $c^{\mathrm{veh}}$ is the cost of a vehicle. In other words, a path is *feasible* if the vehicle has enough capacity to pick-up all the goods on the path with probability $1-\alpha$. A typical application of this problem is *garbage collection*. A task corresponds to a client. The vehicles collecting garbage visit the same sequence of clients every day, but the volume of garbage loaded by each client varies from one day to another.

**Why these problems.** We have chosen these problems for two reasons. First, they are difficult. State of the art algorithms for the pricing subproblems of the stochastic VSP and the chance constrained CVSP are label algorithms. These algorithms rely on a smart enumeration of all the solutions using dominance relations and bounds to discard partial solutions. The dimension of the label strongly impacts the performance of the algorithms due to the curse of dimensionality, and only instances of moderate size can be solve when the dimension is larger than 20. In our case, the dimension of the label is $1 + |\Omega|$ where $\Omega$ is the set of scenarios. For the sample average approximation to be relevant, the number of scenarios $|\Omega|$ must be large, typically a few hundred. Hence, only small instances of our problems can be solved using exact solvers.

Second, we have chosen these problems because we expect our paradigm to perform well on the first one, and not well on the second one. Indeed, our paradigm makes sense only if the hard problem can be well approximated by the easy problem. At first sight, it seems reasonable to approximate the pricing subproblem of the stochastic VSP by a usual shortest path problem. Indeed, delay propagates on arcs, and it would not seem to surprising that the cost $P$ of a path

might be approximated by $\sum_{a \in P} \hat{c}_a$. On the contrary, in the chance constrained CVSP, there is a threshold effect: the path remains feasible until the size of the goods collected reaches the capacity. This threshold effect cannot be captured by a usual shortest path problem. We will see that these intuitions are confirmed by the numerical experiments.

### 1.4 Organization of the paper

The paper is organized as follows. Section 2 provides a literature review and details our contributions. Section 3 starts with a brief introduction to structured learning and then introduces our structured learning methodology to find the best approximation of the hard problem (h) by the easy one (e). The section then provides a step-by-step description of how to apply this methodology to a new problem, and details these steps on the case of path problems and path partition problems, with examples on the applications of Section 1.3. The two next sections introduce the new algorithms required to apply the methodology to path problems and partition problems. More specifically, Section 4 explains how to perform structured learning when using a usual shortest path problem as structured prediction problem, and Section 5 explains how to derive practically efficient algorithms for the path partition problem (3) from an approximation of its pricing subproblem by a usual shortest path problem. Section 6 then provides numerical results showing the efficiency of the approach on our running examples. Finally, Section 7 concludes and provides research directions.

## 2 Literature review and contributions

### 2.1 Literature review

**Leveraging ML to solve combinatorial optimization problems.** Bengio et al. [5] survey the recent attempts at leveraging ML to solve combinatorial optimization problems. They identify three main methodologies. First, "end-to-end learning methods" [4, 9, 12, 13, 16, 18, 23] train the ML model to directly output a solution. Using our terminology of Figure 1.a, $\phi_\theta$ is a ML predictor, generally based on deep learning, that outputs a vector from which a solution can immediately be reconstructed using a simple heuristic. These approaches focus mainly but not only on the traveling salesman problem. Second, "learning meaningful properties of optimization problems" [6, 14, 17] approaches use the ML predictor $\phi_\theta$ to predict information that will be exploited by an advanced optimization algorithm designed to solve the problem of interest. Finally, in "ML alongside optimization algorithms" approaches [3, 12], a ML predictor is regularly called to take heuristic decisions within an optimization algorithm. Our work is somewhere inbetween the three methodologies. When the hard problem is the problem of interest, our work can be compared to the end-to-end learning methods and the learning meaningful properties approaches. Indeed, the algorithm that we use to reconstruct the optimal solution, i.e., the solution algorithm $\mathcal{A}^{\mathrm{e}}$ that we use to solve the easy instance $\Gamma^{\mathrm{e}}$ exploits much more the structure of the problem than the greedy heuristics of end-to-end learning approaches, but has not been conceived to find solutions of our problem of interest, as is the case in "learning meaningful properties of optimization problems" approaches. Finally, when used to approach the pricing subproblem of a column generation, our approach can be considered as a "ML alongside optimization algorithms approach".

Bengio et al. [5] identify three main challenges that must be addressed to improve ML approaches to combinatorial optimization problems. The first challenge is to develop approaches that work when finding feasible solutions is difficult. In its current version, our work does not address well that challenge, in the sense that, if our approach scales very well on the stochastic VSP, where finding a good solution is easy, it does not scale well on the chance constrained

CVSP due to the feasibility constraint. We believe that this comes from our choice of the usual shortest path problem as easy problem (e), which is not adapted to model capacity constraints, and that our paradigm could potentially address such feasibility problems by choosing an easy problem (e) that can better model feasibility constraints. The second challenge is to find neural network architectures that suit to combinatorial optimization problems. In this work, we bypass this challenge by using a feature based ML approach. Developing a feature free version of our approach would require to address this challenge, and we let it to future work. Finally, the third challenge is to develop models that scale to instances larger than those on which the ML approach has been trained. We claim that our approach is one way of meeting that challenge: As we will see in the numerical experiments section, our approach scales very well on instances order of magnitude larger that those on which our ML predictor was trained.

**Structured learning.** Nowozin et al. [19] provide an introduction to structured learning. Taskar et al. [21] detail many different combinatorial optimization problems that can be used as structured prediction problems within structured learning approaches, but do not consider shortest path problems. To the best of our knowledge, no structured learning approach has been proposed with a usual shortest path problem as structured prediction problem. However, paths have already been used by Joulin et al. [11] to encode the solution of a prediction problem in the context of image and video co-localization. They propose a Franck-Wolfe algorithm on the flow polytope to solve the learning problem. However, the objective function of their learning problem is specific to their application and does not suit to a generic structured learning setting.

**Stochastic and chance constrained VSP.** The chance constrained CVRP and stochastic VRP have been well studied and remain especially challenging. We refer the reader to the surveys of [10] and Vidal et al. [22, Section 2.6] for a review of the seminal and recent contributions, respectively. To the best of our knowledge, the chance constrained CVSP and stochastic VSP have been considered mainly in the context of airline problems. In their seminal work on delay in the aircraft routing problem, Lan et al. [15] considered a stochastic VSP where flight legs delay are independent. The successive contributions either use this independence assumptions or heuristic approaches based on simulation [1, 7, 8, 24]. Scenario approaches enable to better take into account the dependence between tasks delays, and remain challenging.

## 2.2 Contributions

Our contributions are as follows.

1. We introduce a new ML for OR paradigm: "If a hard combinatorial optimization problem is a variant of an easy one, then use a ML predictor to approximate the hard problem by the easy one". This paradigm enables to exploit the combinatorial structure of the problem considered.

2. We provide a structured learning methodology that enables to learn the ML predictor required by our paradigm.

3. We introduce a maximum likelihood approach for structured learning problems that use a usual shortest path problem on an acyclic digraph as structured prediction problem.

4. We provide algorithms that implement our methodology and enable to approximate arbitrary path problems on an acyclic digraph by a usual shortest path problem, and arbitrary path partition problems on an acyclic digraph by a flow problem.

5. We also introduce an approach to scale-up column generation approaches with difficult path problems as pricing subproblems. It takes the form of a column generation matheuristic that, instead of solving the true pricing subproblem, solves the approximation by a usual shortest path problem obtained using our ML for OR approach.

6. We provide numerical experiments that test the efficiency of the approach on the stochastic VSP and on the chance constrained CVSP. They notably show that our approach leads to efficient algorithms to solve the stochastic VSP. Such algorithms are practically relevant in the context of the aircraft routing problem.

# 3 Our structured learning methodology

## 3.1 Background on structured learning

Structured learning is a branch of supervised learning, which aims at learning a function

$$\begin{array}{rcl} h : \mathcal{S} & \to & \mathcal{T} \\ s & \mapsto & t \end{array}$$

from a training set $(s_1, t_1), \ldots, (s_n, t_n)$ where $t_i$ is a noisy observation of $h(s_i)$. Structured learning deals with supervised learning problems where, for any $s$ in $\mathcal{S}$, we know that $h(s)$ takes its value in a set $\mathcal{T}(s)$ that is finite, combinatorially large, and structured. To predict the value $h(s)$ on a new instance $s$, we solve the optimization problem

$$t^* = \hat{h}_g(x) := \operatorname*{arg\,min}_{t \in \mathcal{T}(s)} g(s, t)$$

where the statistical model $\hat{h}_g$ is defined through an auxiliary *evaluation function* $g$. In this paper, we restrict ourselves to generalized linear models $\{g_{\boldsymbol{\theta}} : \boldsymbol{\theta} \in \Theta\}$ of the form $g_{\boldsymbol{\theta}}(s, t) = \langle \boldsymbol{\theta} | \boldsymbol{\phi}(s, t) \rangle$, where $\boldsymbol{\phi} : \mathcal{X} \to \mathbb{R}^d$ is a features vector. This gives predictions of the form

$$t^* = \hat{h}_{\boldsymbol{\theta}}(s) := \operatorname*{arg\,min}_{t \in \mathcal{T}(s)} \langle \boldsymbol{\theta} | \boldsymbol{\phi}(s, t) \rangle. \tag{10}$$

Problem (10) is referred as the *structured prediction problem*. Given a training set $(s_i, t_i)_{i \in [n]}$, the objective of the *structured learning problem* is to learn a parameter $\boldsymbol{\theta}$ such that $\hat{h}_{\boldsymbol{\theta}}$ is a good approximation of $h$.

**Approximate problems using structured learning.** We can now reformulate the problem of finding parameter $\boldsymbol{\theta}$ such that $\varphi_{\boldsymbol{\theta}}(\Gamma^{\mathrm{h}})$ is the "best" approximation of $\Gamma^{\mathrm{h}}$ by an instance of (e) as a structured learning problem. The mapping $h : \mathcal{S} \to \mathcal{T}$ that we want to approximate is the mapping that associates to each instance $\Gamma^{\mathrm{h}}$ of the hard problem an optimal solution $x^* \in \mathcal{X}^{\mathrm{h}}(\Gamma^{\mathrm{h}})$. The structured prediction problem we use is

$$\min_{x \in \mathcal{X}^{\mathrm{e}}(\varphi_{\boldsymbol{\theta}}(\Gamma^{\mathrm{h}}))} f^{\mathrm{e}}(x; \varphi_{\boldsymbol{\theta}}(\Gamma^{\mathrm{h}})). \tag{11}$$

and the corresponding structured prediction problem enables to find the parameter $\boldsymbol{\theta}$ such that an optimal solution of (11) is the "best prediction" of $x^*$. In other words, the structured prediction problem coincides with the problem we want to solve.

This leads to the following methodology to learn $\boldsymbol{\theta}$. First, generate a set of moderate size instances $\Gamma^{\mathrm{h}}_1, \ldots, \Gamma^{\mathrm{h}}_n$ of the hard problem (h). Second, use algorithm $\mathcal{A}^{\mathrm{h}}$ to find optimal solutions $x^*_1, \ldots, x^*_n$. Finally, use a structured learning algorithm to solve the structured learning problem and find the $\boldsymbol{\theta}$ that gives the best approximation.

### 3.2 How to apply our ML for OR paradigm on your problem

Suppose that you want to apply our ML for OR paradigm on your hard problem of interest (h). This means that you already have the prerequisites of the method, i.e.,

1. an easy problem (e) that you want to use to approximate (h),
2. and a set of training instances from the family of instances of (h) that you want to solve with their solutions.

The solutions of the training instances are typically computed with an algorithm $\mathcal{A}^{\mathrm{h}}$ that can solve moderately large instances of (h). Applying our methodology will then require to define

3. a parametrized family of mapping $(\varphi_{\boldsymbol{\theta}})_{\boldsymbol{\theta}\in\Theta}$ from the set of instances of (h) to the set of instances of (e),
4. and an algorithm for the resulting structured learning problem.

If your easy problem (e) has already been considered as structured prediction problem in the structured learning literature, then you can directly use the corresponding structured learning algorithm, Otherwise, you will have to develop your own structured learning algorithm.

You might want to apply this approach to a problem (h) that is a subproblem of your problem of interest. Typically, this subproblem could be the separation problem of a family of cuts in a branch and cut algorithm, or the pricing subproblem of a column generation approach. In that case, one last step would be to design

5. an algorithm for your problem of interest that can exploit your heuristic algorithm for the subproblem (h).

In the rest of this section, we detail these different steps on path problems and partition problems.

### 3.3 Application to path problems

We now explain how to approximate hard path problems by a usual shortest path problem using our methodology, and take the pricing subproblems of the stochastic VSP and of the chance constrained CVSP as examples.

**Step 1: identify easy and hard problem** In order to address a wide range of path problems, we use as hard problem h the following *generic path problem*

$$\min\left\{c(P;\mathcal{D})\colon P\in\mathcal{P}_{od}^{D}\right\} \quad \text{with} \quad \mathcal{D}=(D,o,d,b) \quad \text{and} \quad b\in B(D,o,d). \tag{12}$$

An instance $\mathcal{D}$ of the problem is a tuple $(D,o,d,b)$, where $D=(V,A)$ is an acyclic directed graph $(V,A)$, $o$ and $d$ are respectively an origin and a destination vertex in $V$, and the mapping $B:(D,o,d)\mapsto B(D,o,d)$ defines the set of possible parameterizations $b$ of digraph $(D,o,d)$. We denote by $\mathcal{P}_{od}^{D}$ the set of $o$-$d$ paths in $D$. Parameter $b$ provides additional information on the instance that may not be contained in digraph $D$ alone. We denote by $\mathfrak{D}$ the set of possible instances $\{(D,o,d,v)\colon D \text{ a digraph}, o,d\in V^2, b\in B(D,o,d)\}$. The mapping $c$ associates to each instance $\mathcal{D}$ a cost function

$$\begin{array}{rccl} c(\cdot;\mathcal{D}):& \mathcal{P}_{od}(D) & \to & \mathbb{R}\cup\{+\infty\},\\ & P & \mapsto & c(P;\mathcal{D}), \end{array} \tag{13}$$

which assigns a cost $c(P;\mathcal{D})$ to each $o$-$d$ path $P$. A path $P$ is *feasible* if $c(P;\mathcal{D}) < +\infty$.

This rather abstract setting enables to model a wide range of path problems. For instance, the usual shortest path problem is obtained by choosing $B(D,o,d) = \mathbb{R}^A$, and

$$c\colon \big(P;(D,o,d,(b_a)_{a\in A})\big) \mapsto \sum_{a\in P} b_a.$$

The pricing subproblem of the stochastic VSP defined in Section 1.3 is obtained by using $\Big(\big(\xi^{\mathrm{b}}_v(\omega)\big)_{v\in V}, \big(\xi^{\mathrm{e}}_v(\omega)\big)_{v\in V}, \big(\xi^{\mathrm{tr}}_{(u,v)}(\omega)\big)_{(u,v)\in A}\Big)_{\omega\in\Omega}$ as parameter $b$, and the mapping $c$ defined in Equation (8).

We propose to approximate our generic path problem (12) by the usual shortest path problem

$$\min\Big\{\sum_{a\in P}\hat{c}(a;\mathcal{D})\colon P\in\mathcal{P}^D_{od}\Big\}. \tag{14}$$

**Step 2, training set.** We suppose to have a *training set* $(\mathcal{D}_1),\dots,(\mathcal{D}_n)$, where $\mathcal{D}_i$ is an instance of Problem (12), and we compute the solution $P_i$ of each instance $\mathcal{D}_i$ using a path algorithm.

In the case of the pricing subproblems of the stochastic VSP and the chance constrained CVSP, we explain in Section 6.1 how we generate instances of these problems. We solve these pricing subproblems using the monoid resource constrained shortest path framework and algorithms we recently introduced [20].

**Step 3, define mapping $\varphi_{\boldsymbol{\theta}}$.** The structured learning problem leverages the training sample to choose the mapping $\hat{c}$ in such a way that (14) "approximates well" Problem (12). Practically, for a given problem, we define a *feature map*

$$\boldsymbol{\phi}:\begin{array}{ccc}\mathfrak{A} & \to & \mathbb{R}^d\\ (a,\mathcal{D}) & \mapsto & \boldsymbol{\phi}(a,\mathcal{D})\end{array}\quad\text{where}\quad \mathfrak{A}=\left\{(a,\mathcal{D})\colon\left|\begin{array}{l}\mathcal{D}\in\mathfrak{D},\ \ \mathcal{D}=(D,o,d,b)\\ D=(V,A),\ \ a\in A\end{array}\right.\right\} \tag{15}$$

that associates to each arc $a$ of an instance $\mathcal{D}$ a vector of features $\boldsymbol{\phi}(a,\mathcal{D})$. We restrict ourselves to approximate cost functions $\hat{c}(a,D)$ in the family $(c_{\boldsymbol{\theta}})_{\boldsymbol{\theta}\in\mathbb{R}^d}$ of functions

$$c_{\boldsymbol{\theta}}(a,\mathcal{D}) = \langle\boldsymbol{\theta}|\boldsymbol{\phi}(a,\mathcal{D})\rangle \tag{16}$$

where $\langle\cdot|\cdot\rangle$ is the usual scalar product on $\mathbb{R}^d$.

Remark that the approximate cost functions in the family $(c_{\boldsymbol{\theta}})_{\boldsymbol{\theta}\in\mathbb{R}^d}$ have access to information on arc $a$ and instance $\mathcal{D}$ only through $\boldsymbol{\phi}(a,\mathcal{D})$. The feature map $\boldsymbol{\phi}$ must therefore be chosen in such a way that $\boldsymbol{\phi}(a,\mathcal{D})$ contains the information on $(a,\mathcal{D})$ that is relevant to approximate (12).

For instance, on the stochastic VSP, for an arc $a=(u,v)$ we use the reduced cost $-\nu_u$, the travel time $t^{\mathrm{tr}}_a$, the deciles of the distribution of the slack $\max(\xi^{\mathrm{e}}_u+\xi^{\mathrm{tr}}_a-\xi^{\mathrm{b}}_v,0)$ between $u$ and $v$, as well as the value of its cumulative distribution function in -100, -50, -20, -10, 0, 10, 50, 100, 200, and 500 – time is expressed in minutes. For the chance constrained CVSP, we use quantiles of the distributions of $\rho_u$ and $\rho_u+\rho_v$. Quantiles of $\rho_u+\rho_v$ provide information on the correlation between $\rho_u$ and $\rho_v$ that is not available in the quantiles of $\rho_u$ alone.

**Step 4 structured learning algorithm** Our structured learning problem therefore consists in finding a parameter $\boldsymbol{\theta}$ in $\mathbb{R}^d$ such that the shortest path problem

$$\min\Big\{\langle\boldsymbol{\theta}|\boldsymbol{\phi}(a,\mathcal{D})\rangle\colon P\in\mathcal{P}^D_{od}\Big\}. \tag{17}$$

approximates problem (12) as well as possible. To the best of our knowledge, Problem (17) has not been considered as a structured prediction problem in the literature. Section 4 develops a structured learning approach to that problem.

**Step 5** does not apply since we directly approximate the path problem of interest and not one of its subproblems

### 3.4 Application to path partition problems

We now explain how to apply our methodology to path partition problems (3). On these problems, we do not use our methodology directly on the master problem (3), but apply it instead to the pricing subproblem (4), which we consider as our hard problem (h). Since this pricing subproblem is a path problem (12), we approximate it by a usual shortest path problem (14). Steps 1 to 4 are therefore identical to those considered in Section 3.3. After performing these steps, we obtain an approximation of the pricing subproblem by a usual shortest path problem. We then have to perform Step 5 to use this approximation to design efficient algorithms for the master problem (3). Section 5 introduces two approaches: one which retrieves from the approximation of the pricing subproblem by a shortest path problem an approximation of the master problem by a flow problem, and one which performs a heuristic column generation.

## 4 Structured learning for path problems

In this section, we provide a maximum likelihood algorithm for the structured learning problem corresponding to the structured prediction problem (17). We start by introducing some background on maximum likelihood approaches in structured learning.

### 4.1 Background on probabilistic structured learning

Recall that structured learning aims at learning a function

$$\begin{array}{rcl} h : \mathcal{S} & \mapsto & \mathcal{T} \\ s & \to & t \end{array}$$

from a training set $(s_1, t_1), \ldots, (s_n, t_n)$ where $t_i$ is a noisy observation of $h(s_i)$. Given a new $s$, the prediction of $h(s)$ is done by solving the structured prediction problem

$$t^* = \hat{h}_{\boldsymbol{\theta}}(x) := \arg\max_{t \in \mathcal{T}(s)} \langle \boldsymbol{\theta} | \boldsymbol{\phi}(s,t) \rangle.$$

The objective of the *structured learning problem* is to learn a parameter $\boldsymbol{\theta}$ such that $\hat{h}_{\boldsymbol{\theta}}$ is a good approximation of $h$.

We now describe the maximum likelihood approach to structured learning. Given $s$ in $\mathcal{S}$ and $\boldsymbol{\theta}$ in $\mathbb{R}^d$, we endow $\mathcal{T}$ with the probability distribution

$$p(t|s,\boldsymbol{\theta}) = \begin{cases} \frac{1}{Z(s,\boldsymbol{\theta})} e^{-\langle \boldsymbol{\theta} | \boldsymbol{\phi}(s,t) \rangle}, & \text{if } t \in \mathcal{T}(s), \\ 0, & \text{otherwise,} \end{cases} \quad \text{where} \quad Z(s,\boldsymbol{\theta}) = \sum_{t' \in \mathcal{T}(s)} e^{-\langle \boldsymbol{\theta} | \boldsymbol{\phi}(s,t') \rangle}. \tag{18}$$

The collection of distributions $(\mathbb{P}(\cdot|s,\boldsymbol{\theta}))_{\boldsymbol{\theta}}$ is the *exponential family* on $\mathcal{T}$ defined by the features mapping $\boldsymbol{\phi}(s,\cdot)$, and $Z(s,\boldsymbol{\theta})$ is the associated *partition function*.

The *regularized maximum conditional likelihood* learning consists in choosing the parameter

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \quad \text{with} \quad \mathcal{L}(\boldsymbol{\theta}) := \lambda\|\boldsymbol{\theta}\|^2 + \sum_{i=1}^{n} \langle \boldsymbol{\theta} | \boldsymbol{\phi}(s_i, t_i) \rangle + \sum_{i=1}^{n} \log Z(s_i, \boldsymbol{\theta}), \tag{19}$$

where $\lambda > 0$ is the regularization parameter. When $\lambda = 0$, we obtain the *maximum conditional likelihood* learning.

The rationale behind this learning problem is the following. We recall that the Kullback-Leibler divergence

$$\mathrm{KL}(p\|q) = \sum_x p(x) \log\Big(\frac{p(X)}{q(X)}\Big).$$

between two distributions $p$ and $q$ on a finite space $S$ evaluates how different $p$ and $q$ are. The more different the two distributions, the larger the Kullback-Leibler divergence. If we denote by $q(s)$ and $q(s,t)$ the empirical distribution on $\mathcal{S}$ and $\mathcal{S} \times \mathcal{T}$ induced by our training sample $(s_1, t_1), \ldots, (s_n, t_n)$, the optimal solution $\boldsymbol{\theta}^*$ of the maximum conditional likelihood learning problem is the parameter that minimizes the Kullback-Leibler divergence between the empirical distribution $q(s,t)$ and the learned distribution $q(s)p(t|s,\boldsymbol{\theta})$ on $\mathcal{S} \times \mathcal{T}$.

$$\lambda = 0 \quad \text{implies} \quad \boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathrm{KL}\Big(q(s,t)\Big\|q(s)p(t|s,\boldsymbol{\theta})\Big).$$

The regularization term $\lambda\|\boldsymbol{\theta}\|^2$ enables to prevent overfitting.

To solve the learning problem (19) numerically, we use a BFGS algorithm. To that purpose, we need to compute the gradient of $\mathcal{L}$ with respect to $\boldsymbol{\theta}$.

**Proposition 1.** [19, Section 5.2] *$\mathcal{L}(\boldsymbol{\theta})$ is a smooth convex function with gradient*

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = 2\lambda\boldsymbol{\theta} + \sum_{i=1}^{n} \Big( \boldsymbol{\phi}(s_i, t_i) - \mathbb{E}_{t \sim p(t|s_i, \boldsymbol{\theta})} \boldsymbol{\phi}(s_i, t) \Big), \tag{20}$$

*where* $\mathbb{E}_{t \sim p(t|s_i,\boldsymbol{\theta})} \boldsymbol{\phi}(s_i, t) = \frac{1}{Z(s,\boldsymbol{\theta})} \sum_{t \in \mathcal{T}(s)} \boldsymbol{\phi}(s,t) e^{-\langle \boldsymbol{\theta} | \boldsymbol{\phi}(s,t) \rangle}$.

### 4.2 Probabilistic structured learning using shortest path problems

We now introduce our structured learning approach to the structured prediction problem (17) using shortest paths. In that case, $\mathcal{S} = \mathfrak{D}$, and given an instance $s = \mathcal{D} = (D, o, d, b)$, we have $\mathcal{T}(s) = \mathcal{P}_{od}^D$. The approximate cost function (16) leads to the exponential family

$$p(P|\mathcal{D}, \boldsymbol{\theta}) = \frac{1}{Z(\mathcal{D}, \boldsymbol{\theta})} \exp\Big( - \Big\langle \boldsymbol{\theta} \Big| \sum_{a \in P} \boldsymbol{\phi}(a, \mathcal{D}) \Big\rangle \Big) \quad \text{on } \mathcal{P}_{od}^D, \tag{21}$$

where

$$Z(\mathcal{D}, \boldsymbol{\theta}) = \sum_{P \in \mathcal{P}_{od}^D} e^{-\langle \boldsymbol{\theta} | \sum_{a \in P} \boldsymbol{\phi}(a, \mathcal{D}) \rangle}. \tag{22}$$

We therefore obtain the regularized conditional likelihood

$$\mathcal{L}(\boldsymbol{\theta}) = \lambda\|\boldsymbol{\theta}\|^2 + \sum_{i=1}^{n} \Big\langle \boldsymbol{\theta} \Big| \sum_{a \in P_i} \boldsymbol{\phi}(a, \mathcal{D}) \Big\rangle + \sum_{i=1}^{n} \log Z(\mathcal{D}_i, \boldsymbol{\theta}), \tag{23}$$

and gradient

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = 2\lambda\boldsymbol{\theta} + \sum_{i=1}^{n} \Big( \sum_{a \in P_i} \boldsymbol{\phi}(a, \mathcal{D}) - \mathbb{E}_{P \sim p(P|\mathcal{D}_i, \boldsymbol{\theta})} \sum_{a \in P} \boldsymbol{\phi}(a, \mathcal{D}_i) \Big). \tag{24}$$

Given the number of $o$-$d$ paths, we cannot enumerate them explicitely when computing $Z(\mathcal{D}, \boldsymbol{\theta})$, $\mathcal{L}(\boldsymbol{\theta})$ and $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$. In the rest of the section, we show that these quantities can easily be computed by dynamic programming.

**An exponential family of $o$-$d$ paths.** Let $D = (V, A)$ be an acyclic directed graph, and $(r_a)$ and $\varphi_a$ be vectors in $\mathbb{R}^d$ for each arc $a$ in $A$. For any path $P$ in $D$, let

$$r_P = \sum_{a \in P} r_a \quad \text{and} \quad \varphi_P = \sum_{a \in P} \varphi_a.$$

Given two vertices $u$ and $v$, we define $\mathcal{P}_{uv}$ to be the set of $u$-$v$ paths in $D$. We consider the distribution on $\mathcal{P}_{uv}$

$$p_{uv}(P) = \frac{1}{Z_{uv}} \exp\left( - \sum_{a \in P} r_a \right) = \frac{1}{Z_{uv}} e^{-r_P} \quad \text{where} \quad Z_{uv} = \sum_{P \in \mathcal{P}_{uv}} e^{-r_P}. \tag{25}$$

If $u = v$, then $\mathcal{P}_{uv} = \{P_u\}$ where $P_u$ is the path with no arcs in $u$,

$$Z_{uu} = 1, \quad \text{and} \quad \mathbb{E}_{P \sim p_{uv}} \varphi_P = 0. \tag{26}$$

If there is no $u$-$v$ paths, then $Z_{uv} = 0$ and $\mathbb{E}_{P \sim p_{uv}} \varphi_P = 0$. For any vertex $v$, we denote by $\delta^-(v)$ the set of arcs incoming to $v$.

**Proposition 2.** *Let $u$ and $v$ be two distinct vertices in $V$ such that there exists a $u$-$v$ path. We have*

$$Z_{uv} = \sum_{a=(s,v) \in \delta^-(v)} Z_{us} e^{-r_a}, \quad \textit{and} \quad \mathbb{E}_{P \sim p_{uv}} \varphi_P = \sum_{a=(s,v) \in \delta^-(v)} \frac{Z_{us} e^{-r_a}}{Z_{uv}} \Big( \varphi_a + \mathbb{E}_{Q \sim p_{us}} \varphi_Q \Big).$$

Remark that using Equation (26) and Proposition (2), we can compute $Z_{uv}$ and $\mathbb{E}_P$ by dynamic programming along a topological ordering.

*Proof.* The first part of the proof comes from the fact that any $u$-$v$ path $P$ can be decomposed into a path $Q$ followed by an arc $a$ with $a$ in $\delta^-(v)$.

$$Z_{uv} = \sum_{P \in \mathcal{P}_{uv}} e^{-\sum_{a \in P} r_a} = \sum_{a=(s,v) \in \delta^-(u)} e^{-r_a} \sum_{Q \in P_{us}} e^{-\sum_{a' \in Q} r_{a'}} = \sum_{a=(s,v) \in \delta^-(u)} e^{-r_a} Z_{us}.$$

Furthermore

$$\begin{aligned}
\mathbb{E}_{P \sim p_{uv}} \varphi_P &= \frac{1}{Z_{uv}} \sum_{P \in \mathcal{P}_{uv}} e^{-\sum_{a \in P} r_a} \sum_{a \in P} \varphi_a \\
&= \frac{1}{Z_{uv}} \sum_{a=(s,v) \in \delta^-(v)} \sum_{Q \in \mathcal{P}_{us}} e^{-r_a} e^{-\sum_{a' \in Q} r_{a'}} \Big( \varphi_a + \sum_{a' \in Q} \varphi_{a'} \Big) \\
&= \frac{1}{Z_{uv}} \sum_{a=(s,v) \in \delta^-(v)} e^{-r_a} \Big( \varphi_a \sum_{Q \in \mathcal{P}_{us}} e^{-r_Q} + \sum_{Q \in \mathcal{P}_{us}} e^{-r_Q} \varphi_Q \Big) \\
&= \frac{1}{Z_{uv}} \sum_{a=(s,v) \in \delta^-(v)} e^{-r_a} \Big( Z_{us} \varphi_a + Z_{us} \mathbb{E}_{Q \sim p_{us}} \varphi_Q \Big),
\end{aligned}$$

which gives the result. □

**Computing $\mathcal{L}(\boldsymbol{\theta})$ and $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$, and solving the learning problem.** Consider now a fixed $\boldsymbol{\theta}$ and a fixed training sample $(\mathcal{D}_i, P_i)$. Then, defining $r_a = \langle \boldsymbol{\theta} | \boldsymbol{\phi}(a, \mathcal{D}) \rangle$, the distribution $p(P|\mathcal{D}_i, \boldsymbol{\theta})$ of Equation (21) and $p_{od}(P)$ of Equation (25) coincide. Hence, $Z(\mathcal{D}_i, \boldsymbol{\theta})$ can be computed by dynamic programming along a topological ordering using Equation (26) and the first part of Proposition (2). And each component of the vector $\mathbb{E}_{P\sim p(P|\mathcal{D}_i,\boldsymbol{\theta})} \sum_{a\in P} \boldsymbol{\phi}(a, \mathcal{D}_i)\Big)$ can be computed by dynamic programming using Equation (26) and the second part of Proposition (2). Hence, using these algorithms, we can compute $\mathcal{L}(\boldsymbol{\theta})$ and $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$ and solve the learning problem (19) using a BFGS algorithm.

*Remark* 1. Depending on $\boldsymbol{\theta}$, the value of the partition function $Z(\mathcal{D}, \boldsymbol{\theta})$ can be tiny or huge. We therefore store its logarithm to avoid numerical errors in implementations. △

# 5 Heuristics for path partition problems leveraging the pricing subproblem approximation

In this section, we focus on Step 5 of our methodology, and explain how to exploit our approximation of path problems by usual shortest path problems to derive matheuristics for the path partition problem (3). We denote by $\mathcal{M}$ be an instance of the master problem (3). Given a vector of duals $\boldsymbol{\nu} = (\nu_v)_{v\in V\setminus\{o,d\}}$ associated with the cover constraint (3b), we denote by $\mathcal{D}(\mathcal{M}, \boldsymbol{\nu}) = (D, o, d, b(\boldsymbol{\nu}))$ the corresponding instance of the pricing subproblem (4). Following Step 1 to 4 of our structured learning methodology, we suppose to have learned a parameter $\boldsymbol{\theta}$ leading to a good approximation of the pricing subproblem (4) by the usual shortest path problem

$$\min_{P\in\mathcal{P}_{od}} \Big\langle \boldsymbol{\theta} \Big| \sum_{a\in P} \boldsymbol{\phi}\big(a, \mathcal{D}(\mathcal{M}, \boldsymbol{\nu})\big) \Big\rangle. \tag{28}$$

We start by showing that we can derive from (28) an approximate master problem. To that purpose, we restrict ourselves to vector of features of the form

$$\boldsymbol{\phi}(a, \mathcal{D}(\mathcal{M}, \boldsymbol{\nu})) = (\phi^{\mathrm{d}}(a, \mathcal{D}(\mathcal{M}, \boldsymbol{\nu})), \boldsymbol{\phi}^0(a, \mathcal{D})) \quad \text{where} \quad a = (u, v) \quad \text{and} \quad \phi^{\mathrm{d}}(a, \mathcal{D}(\mathcal{M}, \boldsymbol{\nu})) = -\nu_u,$$

with the convention that $\nu_o = 0$. We also split the vector of parameters $\boldsymbol{\theta}$ in $(\theta^{\mathrm{d}}, \boldsymbol{\theta}^0)$, with $\theta^{\mathrm{d}}$ the parameter corresponding the feature $-\nu$. We assume that $\theta^{\mathrm{d}} > 0$. The assumption is without loss of generality. We obtained $\theta$ satisfying $\theta^{\mathrm{d}} > 0$ in all our numerical experiments. Indeed, given the objective of the pricing subproblem (4), it would be very surprising to obtain a structured learning approximation with $\theta^{\mathrm{d}} \leq 0$. Consider now the following *approximate master problem*.

$$\min \quad \sum_{P\in\mathcal{P}} \left( \frac{1}{\theta^{\mathrm{d}}} \sum_{a\in P} \Big\langle \boldsymbol{\theta} \Big| \boldsymbol{\phi}\big(a, \mathcal{D}(\mathcal{M}, \mathbf{0})\big) \Big\rangle \right) z_P, \tag{29a}$$

$$\text{s.t.} \quad \sum_{P\ni v} z_P = 1, \qquad \forall v \in V\setminus\{o, d\}, \tag{29b}$$

$$z_P \in \{0, 1\}, \qquad \forall \in \mathcal{P}_{od}, \tag{29c}$$

**Proposition 3.** *The pricing subproblem of a column generation approach to the approximate master problem (29) is the usual shortest path problem (28), where $\boldsymbol{\nu}$ corresponds to the vector of duals associated to (29b).*

*Proof of Proposition 3.* The pricing subproblem corresponding to a dual solution $\boldsymbol{\nu}$ of (29) is

$$\min_{\mathcal{P}\in\mathcal{P}_{od}} \left( \frac{1}{\theta^{\mathrm{d}}} \sum_{a\in P} \Big\langle \boldsymbol{\theta} \Big| \boldsymbol{\phi}\big(a, \mathcal{D}(\mathcal{M}, \mathbf{0})\big) \Big\rangle \right) - \sum_{v\in P} \nu_v.$$

The equality

$$\left(\frac{1}{\theta^{\mathrm{d}}}\sum_{a\in P}\Big\langle \boldsymbol{\theta}\Big|\boldsymbol{\phi}\big(a,\mathcal{D}(\mathcal{M},\mathbf{0})\big)\Big\rangle\right)-\sum_{v\in P}\nu_v=\frac{1}{\theta^{\mathrm{d}}}\left(\sum_{a\in P}\Big\langle \boldsymbol{\theta}\Big|\boldsymbol{\phi}\big(a,\mathcal{D}(\mathcal{M},\boldsymbol{\nu})\big)\Big\rangle\right)$$

then gives the result. □

We introduce two kinds of algorithms: algorithms solving the initial master problem (3), and algorithms solving the approximate master problem (29). Since the solution set of the two problems coincide only if all the *o-d* paths are feasible, the second kind of algorithms are relevant only for problems such that $c_P < +\infty$ for all $P$ in $\mathcal{P}_{od}$. On our example, they will provide solutions of the stochastic VSP, but not of the chance constrained CVSP.

### 5.1 Flow approach to solve the approximate master problem

It is well-known that the set partitioning formulation (29) is the Dantzig-Wolfe reformulation of the following flow formulation.

$$\min \sum_{a\in A}\Big\langle \boldsymbol{\theta}\Big|\boldsymbol{\phi}\big(a,\mathcal{D}(\mathcal{M},\mathbf{0})\big)\Big\rangle t_a, \tag{30a}$$

$$\text{s.t.} \sum_{a\in\delta^-(v)} t_a = \sum_{a\in\delta^+(v)} t_a = 1, \quad \forall v\in V\backslash\{o,d\}, \tag{30b}$$

$$t_a\in\{0,1\}, \qquad \forall a\in A. \tag{30c}$$

Since flow matrices are totally unimodular, the formulation (30) is very well solved by off-the-shelf MILP solvers, which provides a way to solve efficiently large instances of the approximate problem (29) to optimality.

### 5.2 Column generation approaches to solve the initial master problem

Algorithm 1 describes the usual column generation algorithm, that enables to solve the linear relaxation of the master problem (3). Column generation theory ensures that, when the pric-

**Algorithm 1** Generic column generation algorithm

1: **Input:** an instance $\mathcal{M}$ of (3), $\mathcal{P}'$
2: **if** $\mathcal{P}'=\emptyset$ **then** $\mathcal{P}'\leftarrow\{$paths $o,v,d$ for $v\in V\backslash\{o,d\}\}$;
3: solve the linear relaxation of the master problem (4) restricted to $\mathcal{P}'$ using an LP solver;
4: $\boldsymbol{z},\boldsymbol{\nu}\leftarrow$ optimal solutions of the primal and the dual of the restricted master problem;
5: solve the pricing subproblem instance $\mathcal{D}(\mathcal{M},\boldsymbol{\nu})$;
6: $\mathcal{L}\leftarrow$ negative reduced cost paths found at Step 5;
7: $\mathcal{P}'\leftarrow\mathcal{P}'\cup\mathcal{L}$;
8: **if** $\mathcal{L}\neq\emptyset$ **then go to** Step 3;
9: **return** $\boldsymbol{z}$.

ing subproblem is solved to optimality, Algorithm 1 return an optimal solution $\boldsymbol{z}$ of the linear relaxation of the master problem (3). This is not the case in this paper, where the pricing subproblem considered is intractable. However, we still leverage Algorithm 1 to get a matheuristic that compute a (hopefully good) solution of the master problem (3). Two elements must be specified to define such a matheuristic: First, how the pricing subproblem is approximately solved, and second, how an integer solution of the master problem (3) is retrieved from the fractional solution returned by Algorithm 1.

**Approximate resolution of the pricing subproblem.** Let $\boldsymbol{\nu}$ be a vector of duals, $\mathcal{D}(\mathcal{M}, \boldsymbol{\nu})$ be the corresponding instance of the pricing subproblem, and $\hat{\mathcal{D}}(\mathcal{M}, \boldsymbol{\nu})$ its approximation by a usual shortest path problem. Given an $o$-$d$ path $P$, we denote by $\tilde{c}_P$ its reduced cost $c(P) - \sum_{v \in P} \nu_v$. Since $\hat{\mathcal{D}}(\mathcal{M}, \boldsymbol{\nu})$ is an approximation of $\mathcal{D}(\mathcal{M}, \boldsymbol{\nu})$, the reduced cost $\tilde{c}_{\hat{P}}$ of the optimal solution of $\hat{P}$ of $\hat{\mathcal{D}}(\mathcal{M}, \boldsymbol{\nu})$ may be positive even if there exists paths $P$ with negative reduced cost $\tilde{c}(P)$. In order to increase our chances to find a negative reduced cost path $P$ if such a path exist, we suggest to compute the $k$ best solutions of the usual shortest path problem $\hat{\mathcal{D}}(\mathcal{M}, \boldsymbol{\nu})$. These $k$ best solutions can be computed using K$^*$ algorithm [2], which is a natural generalization of the usual A$^*$ algorithm to find the $k$ shortest paths. However, even if K$^*$ algorithm is quite efficient, it is order of magnitudes slower than the usual dynamic programming algorithm on acyclic digraphs (which we denote by ADP). And furthermore, the larger $k$, the slower K$^*$. To get the best of the two algorithms, we therefore suggest to solve the pricing subproblem using Algorithm (2).

**Algorithm 2** Structured learning heuristic to find negative reduced cost solutions of (4)

**input:** an instance $\mathcal{D}$ of the pricing subproblem (4), an integer $K$;
solve (28) for $\mathcal{D}$ with Algorithm ADP;
$\mathcal{S} \leftarrow \{P\} \cap \mathcal{P}$ where $P$ is the optimal solution computed at Step (2);
$\mathcal{L} \leftarrow \left\{P \in \mathcal{S} \cap \mathcal{P} \colon c(P) - \sum_{v \in P} \nu_v < 0\right\}$;
$k \leftarrow 1$
**while** $k < K$ **and** $\mathcal{L} = \emptyset$ **do**
  $k \leftarrow \min(5k, K)$;
  $\mathcal{S} \leftarrow$ solution of K$^*$ on the $k$-shortest path problem on $\mathcal{D}$ with $r_a = \hat{c}(a, \mathcal{D})$;
  $\mathcal{L} \leftarrow \left\{P \in \mathcal{S} \cap \mathcal{P} \colon c(P) - \sum_{v \in P} \nu_v < 0\right\}$;
**end while**
**return** $\mathcal{L}$;

**Heuristic to get an integer solution** Since we do not solve the linear relaxation to optimality, we do not have a lower bound on the best integer solution that can be reached from the current node. Hence, we cannot use a branch-and-bound algorithm to find an optimal solution. Instead, we use a heuristic branching only to find a feasible integer solution. We branch on the arc variables $t_a$ of the flow formulation (30). Since our objective is to find a feasible solution, we run a depth-first search branching algorithm where we branch on the most integer variable $t_a$ and stop as soon as we find an integer solution. A node $u$ of the branch-and-bound tree is a pair $(d_u, \mathcal{C}_u)$ where $d_u$ is the depth of $u$, and $\mathcal{C}_u$ a set of additional branching constraints of the form

$$\sum_{P \ni a} z_P = b_a \quad \text{with } a \text{ in } A \text{ and } b_a \text{ in } \{0, 1\}. \tag{31}$$

The algorithm maintains a list $\mathsf{L}$ of active nodes until it finds an integer solution. Algorithm 3 states our heuristic branching algorithm. At step 9, the depths $d + 1$ and $d + \frac{1}{2}$ enables to prioritize the first node over the second one. Practically, when initializing Algorithm 1 with $\mathcal{P}'$, we do not add all the columns $P$ for which we can deduce that $z_P = 0$ from the constraints in $\mathcal{C}_u$.

It only remains to explain how to handle the additional constraints of the form (31) in the pricing subproblem of the column generation algorithm. In Approximation (28), we keep the same parameter $\boldsymbol{\theta}$, and only modify the feature $\phi^{\mathrm{d}}\big(a, \mathcal{D}(\mathcal{M}, \boldsymbol{\nu})\big)$ corresponding to the dual. Let $a = (u, v)$ be an arc in $A$. Note that there is at most one constraint of the form (31) in $\mathcal{C}_u$. If

**Algorithm 3** Heuristic branching algorithm

**input:** an instance of (3), a set of columns $\mathcal{P}'$
$\mathsf{L} \leftarrow \{(0, \emptyset)\}$;
**while** $\mathsf{L}$ is not empty **do**
  extract from $\mathsf{L}$ a node $u = (d_u, \mathcal{C}_u)$ with maximum depth $d_u$;
  $\boldsymbol{z} \leftarrow$ solution returned by Algorithm 1 on (3) and $\mathcal{P}'$ with additional constraints $\mathcal{C}_u$;
  add to $\mathcal{P}'$ the columns generated by Algorithm 1;
  **if** $\boldsymbol{z}$ is integer **return** $\boldsymbol{z}$; **stop**;
  choose an arc $a$ such that $\sum_{P \ni a} z_P$ is maximal and there is no constraint (31) for $a$ in $\mathcal{C}_u$;
  add $\left(d+1, \mathcal{C}_u \cup \{\sum_{P \ni a} z_P = 1\}\right)$ and $\left(d+\frac{1}{2}, \mathcal{C}_u \cup \{\sum_{P \ni a} z_P = 0\}\right)$ to $\mathsf{L}$;
**end while**
**return** "no integer solution";

there is one such constraint, we denote by $\nu_a$ the corresponding dual variable, and replace $\nu_u$ by $\nu_u + \nu_a$ in the feature $\phi(a, \mathcal{D}(\mathcal{M}, \boldsymbol{\nu}))$. Otherwise, we keep $\nu_u$.

# 6 Numerical results

## 6.1 Instances generator

We generate instances of our example problems as follows. Tasks correspond to trips in a city on a given day. Time and distances are expressed in minutes. The city corresponds to a square measuring 50 minutes on each side. The origin and the destination of a trip are identified by their coordinates. The beginning time of each trip is randomly drawn between 6 am and 8 pm. The duration of the trips and the travel time between trips are computed as follows. The city is partitioned into districts, which are squares of width 10 minutes. The duration $t^{\mathrm{e}}_v - t^{\mathrm{b}}_v$ of a task $v$ is $\alpha_v l$, where $l$ is the distance between the origin and the destination of the task, and $\alpha_v$ is randomly drawn between 1.2 and 1.6. The duration $t^{\mathrm{tr}}_a$ of a trip between two tasks is the distance $l$ between the destination of the first task and the origin of the second task. This enables to build the digraph $D$ of both problems. Vehicle cost $c^{\mathrm{veh}}$ is chosen equal to 1000 for both problems.

Delay for the stochastic VSP are generated as follows. For each district $d$ and time $h$ of the day we have a random variable $\zeta^{\mathrm{dis}}_{d,h}$ modeling the congestion of $d$ at time $h$. $\zeta^{\mathrm{dis}}_{d,0} = \varepsilon_{d,0}$, and $\zeta^{\mathrm{dis}}_{d,h+1} = \zeta^{\mathrm{dis}}_{d,h}/2 + \varepsilon_{d,h}$, where the $\varepsilon_{d,h}$ are independent and distributed according to a lognormal distribution. We also have a random variable $\zeta^{\mathrm{i}}_h$ modeling congestion on freeways between districts, whose distribution is defined similarly. The parameters of the lognormal distributions $\zeta_{d,h}$ are chosen to model a morning peak and an evening peak of congestion, as well as a city center that it more congested than peripheral areas. Since the time discretization we use is finer than hours, we introduced the notation $h(t)$ for the hour of a time instant $t$. Consider a task $v$ corresponding to a trip between district $d_1$ and district $d_2$. $\xi^{\mathrm{b}}_v = t^{\mathrm{b}}_v + \varepsilon_v$ where $\varepsilon_v$ is distributed according to a lognormal distribution. We have

$$\xi^{\mathrm{e}}_v = \xi^{\mathrm{b}}_v + t^{\mathrm{e}}_v - t^{\mathrm{b}}_v + \zeta^{\mathrm{dis}}_{d_1,h(\xi_1)} + \zeta^{\mathrm{i}}_{h(\xi_2)} + \zeta^{\mathrm{dis}}_{d_1,h(\xi_3)},$$

where $\xi_1 = \xi^{\mathrm{b}}_v$, $\xi_2 = \xi_1 + \zeta^{\mathrm{dis}}_{d_1,h(\xi_1)}$, and $\xi_3 = \xi_2 + t^{\mathrm{e}}_v - t^{\mathrm{b}}_v + \zeta^{\mathrm{i}}_{h(\xi_2)}$. The random travel time $\xi^{\mathrm{tr}}_a$ on arc $(u, v)$ is computed similarly, the only differences being that $\xi_1$ is replaced by $\xi^{\mathrm{e}}_u$ and $t^{\mathrm{e}}_v - t^{\mathrm{b}}_v$ by $t^{\mathrm{tr}}_a$. This choice of distributions ensures that the random variables $\xi^{\mathrm{b}}$, $\xi^{\mathrm{e}}$, and $\xi^{\mathrm{tr}}$ are correlated. The cost of each minute of delay is $c^{\mathrm{del}} = 2$.

Instances of the chance constrained CVSP are generated as follows. The task set $V$ is partitioned into 10 job clusters $V_i$. The volume $\rho_v$ of goods of a task $v$ is composed of a part $\rho_i^{\text{cl}}$ common to all the tasks of the cluster its cluster $i$ and a part $\rho_v^{\text{ta}}$ that is specific to this task.

$$\rho_v = \max\left(0, \min\left(50, \rho_v^{\text{ta}} + \rho_i^{\text{cl}}\right)\right)$$

where $\rho_v^{\text{ta}}$ and $\rho_i^{\text{cl}}$ are distributed according to normal distributions whose means and standard deviations are randomly chosen in the interval $[5, 25]$. Clusters correspond to clients whose levels of activity are correlated. Vehicles capacity is 150, and $\alpha$ is chosen equal to 5% in (9).

For both problems, we generate an instance by choosing the number of tasks $|V|$, the number of scenarios $|\Omega|$, and the seed of the random number generator.

### 6.2 Experimental setting

For our numerical experiments, on each problem, we proceed as follows. First, we generate two sets of instances with the generator described in Section 6.1, one which will be our training set and the other one our test set. The training set contains only instances of moderate size, since we need to be able to solve the pricing subproblem (4) to optimality on these instances.

Second, we solve the linear relaxation of (3) of all instances of moderate size using our exact column generation solver. At each iteration of the column generation, we store the duals $\boldsymbol{\nu}$ and the optimal solution $P$ of the pricing subproblem. This gives us our *pricing subproblem* (4) *training set* (resp. *pricing subproblem* (4) *test set*), which is composed instances $\mathcal{D}(\mathcal{M}, \boldsymbol{\nu})$ for all pairs $(\mathcal{M}, \nu)$, where $\mathcal{M}$ is an instance of the master problem (3) in the training set (resp. test set), and $\boldsymbol{\nu}$ is a dual vector encountered along the column generation to solve $\mathcal{M}$.

Third, we use the pricing subproblem training set to learn the parameter $\theta$ using the BFGS algorithm of Section (4.2).

Fourth, we generate our results on the pricing subproblem (4) test set. We solve each instance with the usual acyclic dynamic programing algorithm.

Fifth, we run the exact column generation algorithm on moderate size instances of our test set, and our MILP solver on the flow MILP (30) and Algorithm 3 on all instances of the test set. This time, we do not store duals and optimal paths in the exact column generation algorithm to ensure a fair comparison. In the heuristic column generation algorithm, we use $K = 50,000$ for both the stochastic VSP and the chance constrained CVSP.

**Master problem instances.** Table 1 summarizes the instances and numerical experiments carried out on the master problem (3). The first line of the table indicates which columns of the table correspond to the training and to the test set. We split the test set into groups of moderate, medium, large, and very large instances. The training set contains only a group of moderate size instances. The second line of the table indicates the group of instances. A *family of instances* is a collection of instances with identical $(|V|, \Omega)$. Each group of instances contains several family of instances. One line in the part "Instance family" of the table corresponds to a family of instances. Column $|V|$ indicates the number of tasks in the instance, column $|\Omega|$ the number of scenarios, and column # the number of instances with $|V|$ tasks and $|\Omega|$ scenarios in the family. The last four lines of the table indicates which algorithms are run on the instances of the family. A tick ✓indicates that the algorithm can solve the instances of both problems on that family, while an S indicates than only the instance of the stochastic VSP can be solved.

**Experimental settings.** All the numerical experiments are run on a linux computer with 16 Gb of RAM and four cores at 2.60 GHz. Algorithms are implemented in `C++`, and LPs are solved using `CPLEX 12.5`. We use the BFGS algorithm of the `CppOptimizationLibrary` [25].

| | **Training set** | | | **Test set** | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Moderate | | | Moderate | | | Medium | | | Large | | | Very Large | | |
| | $\|V\|$ | $\|\Omega\|$ | # | $\|V\|$ | $\|\Omega\|$ | # | $\|V\|$ | $\|\Omega\|$ | # | $\|V\|$ | $\|\Omega\|$ | # | $\|V\|$ | $\|\Omega\|$ | # |
| **Instance** | 10 | 50 | 2 | 10 | 50 | 8 | 100 | 50 | 10 | 500 | 50 | 1 | 1000 | 50 | 1 |
| **family** | | 100 | 2 | | 100 | 8 | | 100 | 10 | | 100 | 1 | | 100 | 1 |
| | | 200 | 2 | | 200 | 8 | | 200 | 10 | | 200 | 1 | | 200 | 1 |
| | 50 | 50 | 2 | 50 | 50 | 8 | 200 | 50 | 10 | 750 | 50 | 1 | 2000 | 50 | 1 |
| | | 100 | 2 | | 100 | 8 | | 100 | 10 | | 100 | 1 | | 100 | 1 |
| | | 200 | 2 | | 200 | 8 | | 200 | 10 | | 200 | 1 | | 200 | 1 |
| | | | | | | | | | | | | | 5000 | 200 | 1 |
| Exact CG | | ✓ | | | ✓ | | | | | | | | | | |
| Flow MIP (30) | | | | | S | | | S | | | S | | | S | |
| Heuristic CG | | | | | ✓ | | | ✓ | | | S | | | | |

Table 1: Summary of the numeric experiments performed on the stochastic VSP and on the chance constrained CVSP – Flow MIP (30) is run only on the stochastic VSP.

## 6.3 Stochastic VSP

### 6.3.1 Structured learning for path problems

Table 2 describes our training and test sets instances as well as the prediction results we obtain on these instances.

**Pricing subproblem instances.** As we already mentioned, our training and test sets for the pricing subproblems are composed of the pricing subproblem instances encountered along the column generation with exact pricing we launched on our instances of the master problem of moderate size. The two first columns of Table 2 provide the characteristics of the family of master problem instances, where $|V|$ is the number of tasks and $|\Omega|$ the number of scenarios. For the training set and the test set, the column "Pr. #" provides the number of pricing subproblem instances. Remark that there are roughly 4 times more instances in the test set than in the training set. This is simply because, as we indicated in Table 1, for each pair $(|V|, \Omega)$, there are 2 instances in the master problem training set and 8 instances in the master problem test set. The column "CG iter." provides the average number of iterations of the column generation. We have a larger number of instance with 50 tasks simply because the column generation needs more iterations to converge on instance with 50 tasks that on instance with 10 tasks. This makes a total of 440 instances in the training set, among which 75 instances with 10 tasks and 365 instances with 50 tasks.

It took a total of 1 hour and 52 minutes of CPU time to generate the training set, and 8 hour and 53 minutes to generate of test set, which gives a total of 10 hours and 46 minutes.

**Learning results.** Table 3 provides the result we obtained when the structured learning problem (19) using the BFGS algorithm of [25] and our algorithms of Section 4.2 to compute the objective function and its gradient. These results are given for different regularization parameters $\lambda$. The first column provides the regularization parameter $\lambda$, the second the number of iterations of the BFGS algorithm, the third the total CPU time of the BFGS algorithm in seconds, and the fourth the value of the regularized log-likelihood $\mathcal{L}(\theta^*)$ on the optimal solution.

We first underline the fact that it takes only a few seconds to find an optimal $\theta$, a time negligible with respect to the time needed to generate the learning dataset. Unless otherwise

| Mast. pb inst. fam. | | Training set | | | | | | Test set | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Pr. | CG | Gap | | | | Pr. | CG | Gap | | | |
| $\|V\|$ | $\|\Omega\|$ | # | iter | tot | 50% | 90% | 100% | # | iter | tot | 50% | 90% | 100% |
| 10 | 50 | 21 | 10.5 | 7% | 2% | 7% | 30% | 85 | 10.6 | 528% | 6% | 434% | 3782% |
| | 100 | 20 | 10.0 | 9% | 3% | 9% | 35% | 82 | 10.2 | 31% | 7% | 34% | 144% |
| | 200 | 22 | 11.0 | 9% | 5% | 6% | 46% | 80 | 10.0 | 21% | 6% | 32% | 57% |
| 50 | 50 | 128 | 64.0 | 24% | 13% | 31% | 55% | 493 | 61.6 | 73% | 16% | 43% | 513% |
| | 100 | 112 | 56.0 | 217% | 16% | 73% | 2014% | 507 | 63.4 | 711% | 14% | 51% | 7214% |
| | 200 | 113 | 56.5 | 31% | 17% | 29% | 123% | 492 | 61.5 | 145% | 17% | 53% | 1256% |

Table 2: Prediction results on the training set and test set on the pricing subproblem (4) regularization parameter $\lambda = 0.001$.

| Regularization $\lambda$ | BFGS iterations | BFGS CPU time (s) | $\mathcal{L}(\theta^*)$ |
|---|---|---|---|
| 0 | 47 | 4.9 | 6.11361 |
| 1e-4 | 46 | 5.1 | 6.12167 |
| 1e-3 | 44 | 4.2 | 6.16037 |
| 1e-2 | 40 | 3.7 | 6.31977 |
| 1e-1 | 28 | 2.8 | 6.83834 |
| 1 | 25 | 2.5 | 7.5175 |

Table 3: Performance of the BFGS algorithm on the structured learning problem.

stated, we always use the $\theta$ obtained with $\lambda = 0.001$, because it is the one that leads to the best results.

**Prediction results.** Given an instance $\mathcal{D}$ of the pricing subproblem, we measure the prediction power of our structured prediction problem using the gap

$$\frac{c(\hat{P};\mathcal{D}) - c(P^*;\mathcal{D})}{|c(P^*;\mathcal{D})|}$$

between the cost of the solution $\hat{P}$ returned by the structured prediction problem and the cost of the optimal solution $P^*$ of the pricing subproblem. Since the denominator of the last iteration is generally equal to 0, we compute averages of the gap on all iterations but the last. A very important aspect when using our structured predictor within a column generation is its ability to find a negative cost path when such a path exists. Observe that this is the case if the gap is smaller than 100%

Table 2 provides the value of this gap on for the different families of instances. The column "Gap tot" provides the average gap on all the instances of the family. Since the accuracy of this prediction may vary across a column generation, the three next columns provide the average gap corresponding to a subset of the instances: "Gap 50%" corresponds to iterations in the 50% of iterations at the beginning of the column generation, "Gap 90%"to iterations between 50% and 90% of the total, and "Gap 100%" to iterations in the last 10%. We can see that our approximate pricing algorithm performs very well during the first half of the column generation, slightly less well at the beginning of the second half, and poorly during the last iterations. This fact is made clearer on Figure 2, which gives, on the instances of the test set, the average reduced cost $c(\hat{P};\mathcal{D})$ and $c(P^*;\mathcal{D})$ of the path generated by the approximate

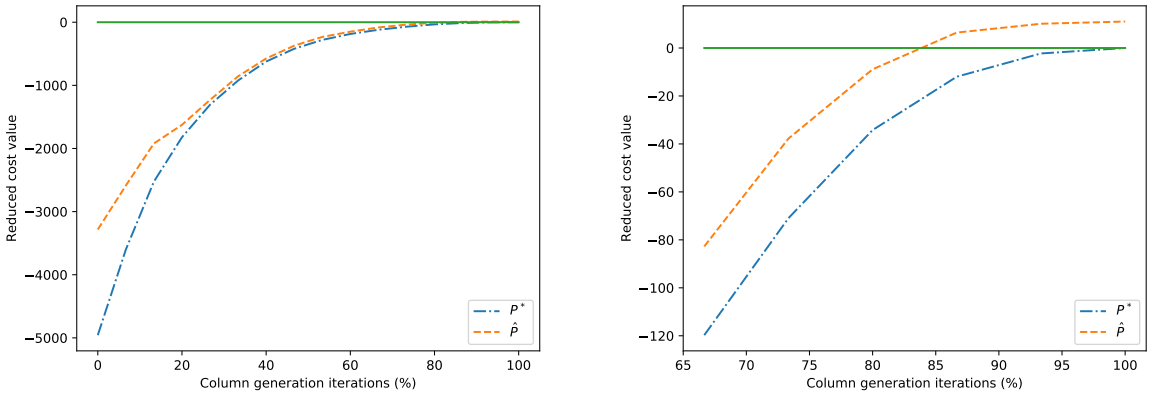


Figure 2: Average reduced cost of the solutions returned by the pricing algorithm as a function of the rank of the iteration among the total number of iterations in the column generation on instances of the test set with 50 tasks. Left hand side: all iterations. Right hand side: last iterations.

and exact algorithms as a function of the rank $\frac{\text{iteration number}}{\text{total number of iterations}}$ of the iteration in the column generation. The right hand side is a zoom on the last iterations. We can see that the approximate pricing algorithm is able to find good quality paths for 80% of the iterations, and is no more able to find negative reduced columns for the last iterations. However, since the last iterations do not have much impact on the objective value of the column generation, and since the approximate pricing algorithm is several order of magnitudes faster than the exact one, using the approximate pricing algorithm looks like a good idea.

### 6.3.2 Column generation matheuristics to solve the master problem

Table 4 provides the performance of our algorithms on the master problem (3). The first column provides the algorithm. "CG ex" corresponds to the exact column generation for the linear relaxation. It thus returns a lower bound on the cost of the optimal solution. "CG heur" corresponds Algorithm 3, which uses Algorithm 2 as a subroutine to solve the pricing subproblem. "Flow" corresponds to the direction resolution of the flow MILP (30). And finally, "No Prop." is a natural heuristic obtained by supposing that the delay does not propagate. This problem is easily solved using a flow approach similar to 5.1. Its value provides a poor lower bound on the value of an optimal solution, and its solution is feasible and hence provides an upper bound on the value of the problem we are interested in. The columns $|V|$ and $|\Omega|$ provide the parameters of the family of master problems instances. Then for each family of problem instances in the training set or in the test set, we provide four statistics averaged on the family of instances. Column "CPU (s)" provides the average CPU time needed to solve an instance. Column "speed-up" provides the average speed-up given by the algorithm considered with respect to the exact column generation. Column "value / $|V|$" provides the average cost per task $\frac{\sum_{P \in \mathcal{P}_{od}} z_P}{|V|}$ in the solution returned. And finally column "gap (%)" provides the average gap between the value of the solution returned and the lower bound on the cost of an optimal solution given by the value of the linear relaxation of (3) computed by the exact column generation.

We underline that our algorithms provide much better solutions than the no propagation heuristic. Our two heuristics have their respective advantages. The flow approach enables to compute extremely fast solutions with an optimality gap non-greater than 5%, while Algorithm 3

| | Mast. pb inst. fam. | | Training set | | | | Test set | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **alg.** | $\|V\|$ | $\|\Omega\|$ | CPU (s) | speed up (×) | value $/\|V\|$ | gap (%) | CPU (s) | speed up (×) | value $/\|V\|$ | gap (%) |
| CG ex | 10 | 50 | 0.44 | – | 380.9 | – | 0.41 | – | 380.0 | – |
| | | 100 | 0.73 | – | 383.1 | – | 0.96 | – | 381.1 | – |
| | | 200 | 1.66 | – | 380.0 | – | 1.49 | – | 381.7 | – |
| | 50 | 50 | 417.94 | – | 282.3 | – | 442.70 | – | 281.2 | – |
| | | 100 | 804.32 | – | 282.3 | – | 946.08 | – | 280.8 | – |
| | | 200 | 1846.72 | – | 282.0 | – | 2279.81 | – | 281.2 | – |
| CG heur | 10 | 50 | 0.08 | 7 | 380.9 | 0.0% | 0.09 | 4.5 | 380.0 | 0.0% |
| | | 100 | 0.17 | 4.2 | 383.1 | 0.0% | 0.17 | 5.8 | 381.1 | 0.0% |
| | | 200 | 0.30 | 5.5 | 380.0 | 0.0% | 0.33 | 4.6 | 381.7 | 0.0% |
| | 50 | 50 | 0.72 | 5.9e+02 | 283.1 | 0.3% | 0.59 | 8.4e+02 | 284.0 | 1.0% |
| | | 100 | 0.61 | 1.3e+03 | 283.3 | 0.3% | 0.98 | 1e+03 | 284.9 | 1.4% |
| | | 200 | 1.19 | 1.6e+03 | 283.0 | 0.3% | 2.21 | 1e+03 | 284.9 | 1.3% |
| Flow | 10 | 50 | 0.01 | 49 | 432.2 | 13.5% | 0.01 | 50 | 428.6 | 12.8% |
| | | 100 | 0.01 | 80 | 434.2 | 13.3% | 0.01 | 1.1e+02 | 434.7 | 14.1% |
| | | 200 | 0.01 | 1.7e+02 | 432.9 | 13.9% | 0.01 | 1.8e+02 | 434.8 | 13.9% |
| | 50 | 50 | 0.02 | 2e+04 | 295.1 | 4.5% | 0.02 | 2.1e+04 | 293.5 | 4.4% |
| | | 100 | 0.02 | 4.9e+04 | 296.9 | 5.1% | 0.02 | 5.6e+04 | 292.7 | 4.2% |
| | | 200 | 0.02 | 1.2e+05 | 295.2 | 4.7% | 0.02 | 1.4e+05 | 294.0 | 4.6% |
| No prop. | 10 | 50 | 0.01 | 46 | 392.4 | 3.0% | 0.01 | 54 | 392.2 | 3.2% |
| | | 100 | 0.01 | 92 | 395.2 | 3.2% | 0.01 | 1.3e+02 | 393.6 | 3.3% |
| | | 200 | 0.01 | 2.4e+02 | 392.3 | 3.3% | 0.01 | 2e+02 | 394.0 | 3.2% |
| | 50 | 50 | 0.02 | 1.9e+04 | 337.3 | 19.5% | 0.02 | 2.2e+04 | 325.1 | 15.6% |
| | | 100 | 0.02 | 3.9e+04 | 335.6 | 18.9% | 0.02 | 4.8e+04 | 328.0 | 16.8% |
| | | 200 | 0.02 | 9.2e+04 | 320.3 | 13.6% | 0.02 | 1.1e+05 | 330.3 | 17.5% |

Table 4: Performance of our heuristics on instances whose relaxation is solved to optimality by our exact column generation.

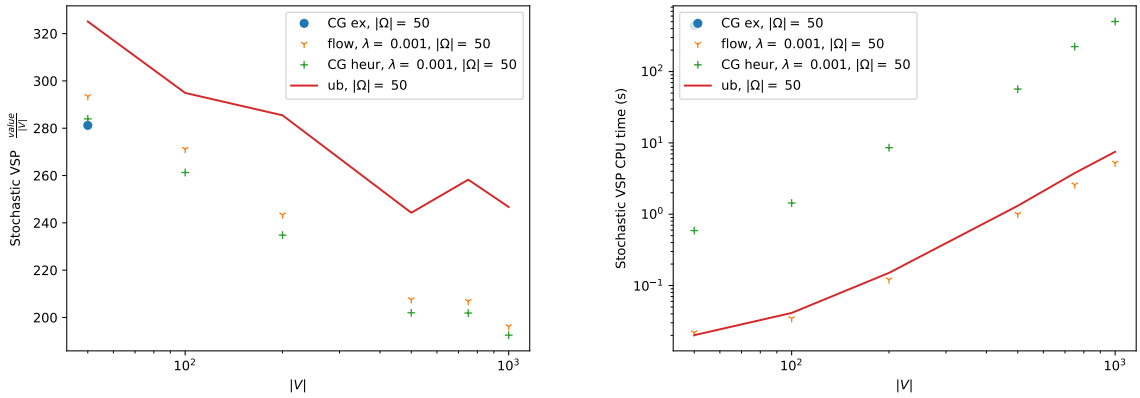


Figure 3: Average CPU time of and cost by task returned by our algorithms on the large instance of our test set with 50 scenarios.

is slightly slower but enables to compute solution with a 1.5% optimality gap. We underline that Algorithm 3 is still 3 to 4 order of magnitude faster than the exact column generation.

Figure 3 provides the average computing time and value by task $\frac{\sum_{P \in \mathcal{P}_{od}} z_P}{|V|}$ on our instances. Larger instances offer more flexibility in the construction of the sequences of tasks. Hence, we expect the average cost by task to decrease with the size of the instance. We say that an algorithm scales if it indeed finds solutions with smaller average cost by task on larger instances. Indeed, finding solutions of large instances that have the same cost by task as the solution of small instances is easy: It suffices to arbitrarily partition the large instance into small instances, and compute the solution on the small instances. We can see that *both of our algorithms scale very well to instances of large size*: For both algorithms, the average cost by task decreases with the size of the instance, and is much smaller on large instances than the average cost by task found by the exact algorithm on the instances it can solve.

We advocate that our two algorithms correspond to different usecases. The flow approximation provides good solution on instances with one thousand of tasks in a few seconds. Such an algorithm would be extremely useful in an online version of the stochastic VSP, where an instance must be solved extremely fast to adapt the planning after an event – such problems are routinely solved by airlines to adapt in real-time airplane schedules after a disruption. The approximate column generation gives an excellent solution in less than an hour. Such an algorithm is very useful for day-ahead planning.

### 6.4 Chance constrained CVSP

Figure 4 provides the equivalent of Table 3 and Figure 3 for the chance constrained CVSP. We first remark the the value of the log-partition function $\mathcal{L}(\theta^*)$ is larger than for the stochastic VSP. The structured learning algorithm does not fit as well the learning set of the chance constrained CVSP as it fits the learning database of the stochastic VSP. In terms of results on the path prediction problem, the predictor learned performs poorly: The optimal solution of the structured prediction problem is never feasible on our test set. However, there are feasible paths among the $K = 50,000$ best solutions of the usual shortest path problem computed by Algorithm 2, which enables to run our column generation matheuristic 3. The results on Figure 4 show that this matheuristic does not scale: Even on instances with 200 tasks, the average cost by task of the solution returned by our heuristic is larger than the average cost by

| $\lambda$ | iter | CPU(s) | $\mathcal{L}(\theta^*)$ |
|---|---|---|---|
| 0 | 159 | 53.062 | 10.6 |
| 1e-4 | 63 | 19.86 | 10.8 |
| 1e-3 | 36 | 10.945 | 10.9 |
| 1e-2 | 23 | 6.881 | 11.0 |
| 1e-1 | 14 | 4.092 | 11.0 |
| 1 | 11 | 3.447 | 11.3 |

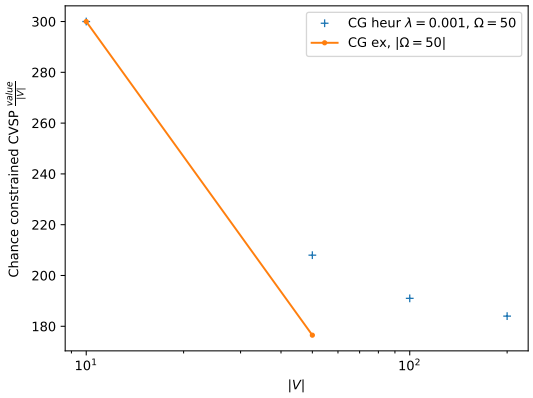


Figure 4: Learning algorithm results, and average cost by task on the test set using the exact column generation and our column generation matheuristic on the chance constrained CVSP.

task on instances with 50 tasks returned by the exact algorithm.

We spent a significant amount of time trying to improve our heuristic, without much success. The lesson we retain from this attempt is that *our paradigm can work only if the hard problem can be well approximated by the easy problem.* One conclusion of our numerical results is that the pricing subproblem of the stochastic VSP can be well approximated by the usual shortest path problem, while this is not the case of the pricing subproblem of the chance constrained CVSP.

# 7 Conclusion

We have proposed the following "ML for OR" paradigm: Use ML to approximate instances of a hard combinatorial optimization problem by instances of a simpler problem. More precisely, suppose that a hard problem of interest is a variant of an easier one. Then our paradigm leads to the following heuristic for the hard problem. Given an instance of the hard problem, use an ML predictor to approximate it by an instance of the easy problem, solve the easy problem, and retrieve a solution of the hard problem from the solution of the instance of the easy problem. And we have introduced a structured learning methodology to learn the ML predictor.

We illustrate our methodology on path problems and path partition problems. To that purpose, we have introduced a structured learning approach to approximate path problems by the usual shortest path problem. And we have proposed a flow based approximation and a column generation matheuristic for path partition problems.

We illustrate our paradigm on the stochastic VSP. Our flow based heuristic is 4 order of magnitudes faster than the original algorithm and gives solution within a 5% optimality gap, while our column generation based matheuristic is 3 order of magnitude faster and provides solutions within a 1.5% optimality gap. Both heuristics scale well on instances with one thousand of tasks, when the initial algorithm could only handle instances with 50 tasks.

Of course, our paradigm can work only if instances of the hard problem can be well approximated by instances of the easy problem. The main reason for the success of our algorithm on the stochastic VSP is that its pricing subproblem is well approximated by the usual shortest path problem. On the chance constrained CVSP, whose pricing subproblem is not well approximated by a usual shortest path problem, our heuristics perform poorly. Applying our paradigm

to a new hard problem therefore requires to find an easy problem that approximates well the hard problem. And our methodology to learn this approximation requires a moderately efficient solver for the hard problems, and an algorithm for the structured learning problem obtained when the easy problem is used as structured prediction problem.

We have shown that, when applied to the stochastic VSP, our paradigm addresses (at least partially) one of the challenges identified by Bengio et al. [5] in they survey on "ML for OR" methods: Our heuristics scale to instances much larger than those used in the training set. Future works may explore if methods using this paradigm can be developed to address the two remaining challenges. That is, what would be the right ML predictor to use within this framework? In particular, we use a feature based predictor, and it would be handy to use feature free methods. And second, how to apply the paradigm when it is hard to find a feasible solution of the hard problem?

**Acknowledgments.** I am grateful to Guillaume Obozinski for his suggestion to look at the structured learning literature.

## References

[1] Shervin Ahmadbeygi, Amy Cohn, and Marcial Lapp. Decreasing airline delay propagation by re-allocating scheduled slack. *IIE transactions*, 42(7):478–489, 2010.

[2] Husain Aljazzar and Stefan Leue. K*: A heuristic search algorithm for finding the k shortest paths. *Artificial Intelligence*, 175(18):2129–2154, 2011.

[3] Radu Baltean-Lugojan, Pierre Bonami, Ruth Misener, and Andrea Tramontani. Selecting cutting planes for quadratic semidefinite outer-approximation via trained neural networks. 2018.

[4] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[5] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv preprint arXiv:1811.06128*, 2018.

[6] Pierre Bonami, Andrea Lodi, and Giulia Zarpellon. Learning a classification of mixed-integer quadratic programming problems. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 595–604. Springer, 2018.

[7] Michelle Dunbar, Gary Froyland, and Cheng-Lung Wu. Robust airline schedule planning: Minimizing propagated delay in an integrated routing and crewing framework. *Transportation Science*, 46(2):204–216, 2012.

[8] Michelle Dunbar, Gary Froyland, and Cheng-Lung Wu. An integrated scenario-based approach for robust aircraft routing, crew pairing and re-timing. *Computers & Operations Research*, 45:68–86, 2014.

[9] Patrick Emami and Sanjay Ranka. Learning permutations with sinkhorn policy gradient. *arXiv preprint arXiv:1805.07010*, 2018.

[10] Michel Gendreau, Gilbert Laporte, and René Séguin. Stochastic vehicle routing. *European Journal of Operational Research*, 88(1):3–12, 1996.

[11] Armand Joulin, Kevin Tang, and Li Fei-Fei. Efficient image and video co-localization with frank-wolfe algorithm. In *European Conference on Computer Vision*, pages 253–268. Springer, 2014.

[12] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

[13] Wouter Kool, Herke van Hoof, and Max Welling. Attention solves your tsp, approximately. *stat*, 1050:22, 2018.

[14] Markus Kruber, Marco E Lübbecke, and Axel Parmentier. Learning when to use a decomposition. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 202–210. Springer, 2017.

[15] Shan Lan, John-Paul Clarke, and Cynthia Barnhart. Planning for robust airline operations: Optimizing aircraft routings and flight departure times to minimize passenger disruptions. *Transportation Science*, 40(1):15–28, 2006.

[16] Eric Larsen, Sébastien Lachapelle, Yoshua Bengio, Emma Frejinger, Simon Lacoste-Julien, and Andrea Lodi. Predicting solution summaries to integer linear programs under imperfect information with machine learning. *arXiv preprint arXiv:1807.11876*, 2018.

[17] Rafid Mahmood, Aaron Babier, Andrea McNiven, Adam Diamant, and Timothy CY Chan. Automated treatment planning in radiation therapy using generative adversarial networks. *arXiv preprint arXiv:1807.06489*, 2018.

[18] Alex Nowak, Soledad Villar, Afonso S Bandeira, and Joan Bruna. A note on learning algorithms for quadratic assignment with graph neural networks. *stat*, 1050:22, 2017.

[19] Sebastian Nowozin, Christoph H Lampert, et al. Structured learning and prediction in computer vision. *Foundations and Trends® in Computer Graphics and Vision*, 6(3–4): 185–365, 2011.

[20] Axel Parmentier. Algorithms for non-linear and stochastic resource constrained shortest path. *Mathematical Methods of Operations Research*, 89(2):281–317, 2019.

[21] Ben Taskar, Vassil Chatalbashev, Daphne Koller, and Carlos Guestrin. Learning structured prediction models: A large margin approach. In *Proceedings of the 22nd international conference on Machine learning*, pages 896–903. ACM, 2005.

[22] T. Vidal, G. Laporte, and P. Matl. A concise guide to existing and emerging vehicle routing problem variants. Technical report, Pontifical Catholic University of Rio de Janeiro, 2019. URL http://arxiv.org/abs/1906.06750.

[23] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

[24] Oliver Weide, David Ryan, and Matthias Ehrgott. An iterative approach to robust and integrated aircraft routing and crew scheduling. *Computers & Operations Research*, 37(5): 833–844, 2010.

[25] Patrick Wieschollek. Cppoptimizationlibrary. https://github.com/PatWie/CppNumericalSolvers, 2016.