A Flow-Based Solver for Large-Scale Combinatorial Optimization

Alexander Souza
Algomia GmbH and University of Zurich
alexander@algomia.com

Tristan Koning
Algomia GmbH and University of Zurich
tristan@algomia.com

Tiago Ferreiro Matos
University of Zurich
tiago.ferreiromatos@uzh.ch

Lucien Kern
University of Zurich
lucienkimi.kern@uzh.ch

Kai Schärer
University of Zurich
kai.schaerer@uzh.ch

Abstract

We report on our design, implementation, and computational results of a new flow-based solver for combinatorial optimization. The technology is capable of solving very large instances having up to billions of graph-edges within days and accuracy of $1-2\%$, consistently. The results have been validated on real-world instances occurring in the Swiss railway industry.

1 Introduction

Combinatorial optimization has proved time and again, to be an extremely useful method for deriving *better decisions*, whenever scarce and constrained resources have to be allocated, see, e.g. [1]. Typical areas of industrial use-cases include transportation, logistics, healthcare, finance, technology, defense, and more. A common problem is *combinatorial explosion* at growing problem-sizes, which limits the scalability of algorithms. Given the usefulness of the combinatorial optimization approach, the need for large-scale solvers becomes evident.

The present paper reports our current engineering progress in designing, implementing, and operating a flow-based solver for large-scale and real-world combinatorial optimization. The solver has been tested extensively at the Swiss Federal Railways (and other customers), see Section 4. The challenge there was to optimize the crew shifts of the train-drivers across the entire country. This is a massive scheduling problem; and our instances feature graphs with billions of edges. We could solve these problems with high quality, i.e., optimality gap usually around 1-2$\%$, consistently and within few days of computation time, see Table 1. Our largest instance has 2.3 billion edges and was solved to an accuracy of 4.74% within 104 hours. This was a substantial breakthrough, since the methods applied previously did not scale accordingly.

Our main design goals and decisions, see Section 3, have been:

1. We require that the solver is applicable to combinatorial optimization problems, with reasonable modeling efforts. Our answer to this is to formulate such problems in terms of SET CONSTRAINED FLOW, which extends network flows with additional edge-set constraints.
2. Many real-world use-cases have *locality* and *globality* properties, that shall be supported in the modeling: There is a clear separation between groups of constraints that link resources at a narrow perspective, e.g., precedence requirements for tasks, and groups of constraints that have to hold at a high level view, e.g., task coverage requirements. In our approach,

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: ScaleOPT.

local constraints are mostly captured in a network, while the global constraints are treated with edge-set constraints.

3. At the same time, we want to provide strong guardrails in modeling, by only allowing a small number of types of constraints.

4. The solver has to scale "out of the box". We chose to implement a long-known, proven method for solving large-scale linear programs: column generation, see, e.g., [2, 3]. The approach iterates between solving master- and pricing-problems. We solve the pricing-problems in parallel, and will implement GPU support soon.

5. We seek to have guarantees on the quality of the solutions found by the solver. Comparing the objective-value of the solution of the solver with the linear programming lower bound, determined by column generation, achieves this goal in a well-known manner.

6. Expert users shall be able to configure the solver, in order to fine-tune it towards their use-cases. We argue that our modular architecture, see below, yields this objective. Our implementations of the modules that have been tested and yield proven value.

7. The software shall be accessible easily, even if sophisticated features, like GPU-acceleration (once supported), are in place. It was straightforward to make the solver available through a REST API, which hides the technical deployment details from a user.

2 Related Work

This brief survey reviews leading commercial and open-source solvers, emphasizing their suitability for large-scale optimization and advances in GPU-acceleration, see [4, 5] for an in-depth study.

Commercial Solvers. IBM ILOG CPLEX excels in solving linear and mixed-integer programming problems, offering reliable performance and industry-heavy documentation. It supports large models and integrates with high-level modeling languages, but does not natively leverage GPU-acceleration [6]. Gurobi is recognized for fast, scalable solutions on substantial LP and MILP problems, often setting performance standards. Gurobi utilizes CPU parallelism but currently lacks native GPU-acceleration in its commercial releases [7]. FICO Xpress offers robust capabilities for enterprise-scale LP and MIP, with broad language interfaces and cloud integration. Although not specialized for GPU, its parallel algorithms enable the solution of large and complex models [8]. Hexaly (Local Solver), engineered for large scheduling problems, harnesses advanced metaheuristics and has outperformed conventional solvers in job-shop benchmarks. It primarily capitalizes on CPU heuristics rather than explicit GPU-acceleration [9].

Open-Source and Specialized Solvers. SCIP stands as a top open-source solver for MIP and MINLP, praised for extensibility and competitive performance on large-scale academic models. Focused on CPU computation, it does not incorporate GPU support by default, but is highly customizable [10]. HiGHs specializes in large-scale, sparse LP, MIP, and QP [11]. Importantly, HiGHs distinguishes itself with GPU support via CUDA, enabling efficient parallel computations for large linear problems [12]. However, it has been reported, that the quality of the solutions found with GPU-acceleration falls behind [13]. Its simplicity and MIT license increase accessibility for large-scale industrial and academic projects. Google OR-Tools offers a versatile toolkit for various combinatorial problems, integrating HiGHs and other engines to tackle expansive optimization models. It is widely used in routing and scheduling applications [14]. DISCO (Diffusion Solver for Large-Scale Combinatorial Optimization) leverages neural and diffusion model techniques to rapidly solve large combinatorial instances. DISCO is especially efficient in both inference speed and solution quality for benchmarks like TSP, notably outperforming classic methods in large-scale settings with divide-and-conquer and GPU-acceleration [15]. Knitro introduces new algorithms and performance improvements tailored for non-linear and linear optimization. Features such as primal-dual gradients and augmented Lagrangian methods, faster MILP/MINLP presolve, and hardware-aware enhancements enable quick solutions for previously hard models [16].

3 Solver

Our algorithm solves a combinatorial optimization problem called SET CONSTRAINED FLOW directly, by employing the column generation method in a modular software architecture. For an introduction to the field, see, e.g. [1]. In this section, we summarize our approach.

3.1 Problem Formulation

The problem SET CONSTRAINED FLOW is defined as follows: We are given a directed graph $G = (V, E)$ on n vertices and m edges, and a family $\mathcal{S} = \{S_1, \ldots, S_k\}$ of k *set constraints*, where the element $S_i = (E_i, \ell_i, u_i)$ defines some set $E_i \subseteq E$ and two numbers $\ell_i \leq u_i$. Each vertex $v \in V$ is given by a pair $v = (i_v, b_v)$, where $i_v \in \mathbb{N}$ is a unique *identifier*, and $b_v \in \mathbb{R}$, the *balance* of v. A vertex v is called a *source* if $b_v > 0$ and a *sink* if $b_v < 0$. Each edge $e \in E$ is given by a tuple $e = (i_e, s_e, t_e, c_e, w_e, d_e)$, where $i_e \in \mathbb{N}$ is a unique *identifier*, $s_e \in V$ is the *source*, $t_e \in V$ is the *target*, $c_e \in \mathbb{R}$ is the *cost*, w_e is the *weight*, and finally $d_e \in \{\mathbb{B}, \mathbb{Z}, \mathbb{R}\}$ is the *domain* of e. For each edge e, there is a variable $x_e \in d_e$, i.e., in the domain given by the edge. Define the vector $x = (x_e : e \in E)$ and its *cost* by $\text{cost}(x) = \sum_{e \in E} c_e \cdot x_e$. The optimization problem is:

$$\text{minimize} \qquad \text{cost}(x) = \sum_{e \in E} c_e \cdot x_e, \tag{1}$$

$$\text{subject to} \qquad \sum_{e \in E^+(v)} x_e - \sum_{e \in E^-(v)} x_e = b_v \qquad v \in V, \tag{2}$$

$$\sum_{e \in E_i} w_e \cdot x_e \leq u_i \qquad S_i \in \mathcal{S}, \tag{3}$$

$$\sum_{e \in E_i} w_e \cdot x_e \geq \ell_i \qquad S_i \in \mathcal{S}, \tag{4}$$

$$x_e \in d_e \qquad e \in E. \tag{5}$$

Here, (1) is the *objective function*, (2) are the *balance constraints*, (3) are the *upper constraints*, and (4) are the *lower constraints*. $E^-(v)$ and $E^+(v)$ denote the *incoming* and *outgoing* edges of vertex v.

SET CONSTRAINED FLOW is said to be in *canonical form*, if there is a *singleton constraint* $S_i \in \mathcal{S}$ with $E_i = \{e\}$, $l_i = 0$, and $w_e > 0$ for each $e \in E$. In that case, we identify i and e, and the edge e has a *capacity* $k_e = u_e / w_e$. A set constraint, which is not a singleton constraint is called *complicating*. Observe that MINIMUM COST FLOW and hence also SHORTEST PATH are direct special cases of SET CONSTRAINED FLOW in canonical form.

In Section 5, we present reductions that solve SET COVER and KNAPSACK and give computational evaluations for selected benchmark instances.

3.2 Column Generation

Column generation is a well-known method for solving large-scale linear programs, having a substantial body of literature, see, e.g., [2, 3]. It can be seen as an implementation of the SIMPLEX algorithm, in which the columns of the original constraint matrix A are not given explicitly, but where "promising" columns are generated on the fly. In the particular approach of *Dantzig-Wolfe decomposition* [2, 3], the original matrix A is split into submatrices C and $P_1, \ldots, P_r$, where the P_j are disjoint block matrices, and C is the remainder of A. After the decomposition is determined, the problem is reformulated: Any solution of a *pricing problem* induced by P_j represents a column of the *master problem*. In each iteration, the (restricted) master problem is solved with the currently available columns, while each pricing problem P_j is usually optimized with a specific combinatorial algorithm. The coefficients of the objective functions of the pricing problems are the reduced costs derived from dual solutions of the master. We omit the details here.

Our basic approach for SET CONSTRAINED FLOW (in canonical form), is to construct the Dantzig-Wolfe decomposition as follows: The overall graph G is suitably subdivided into subgraphs, where each subgraph H induces a pricing problem P. Each edge e of H yields a coefficient r_e in the objective function of P. Further, we include the balance constraints and the non-negativity constraints.

Therefore, P becomes a SHORTEST PATH problem:

$$\text{minimize} \qquad \text{cost}(x) = \sum_{e \in E_H} r_e \cdot x_e, \tag{6}$$

$$\text{subject to} \qquad \sum_{e \in E_H^+(v)} x_e - \sum_{e \in E_H^-(v)} x_e = b_v \qquad v \in V_H, \tag{7}$$

$$x_e \geq 0 \qquad e \in E_H. \tag{8}$$

Let $\mathcal{P} = \{p_1, p_2, \ldots p_N\}$ denote the paths that have been generated and let these be indexed by $J = \{1, 2, \ldots, N\}$. Let y_j denote a variable associated with the path p_j. Let c_j be its cost, which equals the total cost of the edges in the path. Then the restricted master problem becomes:

$$\text{minimize} \qquad \text{cost}(y) = \sum_{j \in J} c_j \cdot y_j, \tag{9}$$

$$\text{subject to} \qquad \sum_{e \in E_i} \sum_{j \in J : e \in p_j} w_e \cdot y_j \leq u_i \qquad S_i \in \mathcal{S}, \tag{10}$$

$$\sum_{e \in E_i} \sum_{j \in J : e \in p_j} w_e \cdot y_j \geq \ell_i \qquad S_i \in \mathcal{S}, \tag{11}$$

$$y_j \geq 0 \qquad j \in J. \tag{12}$$

Regarding the implementation of the above schema, observe that the number of non-empty singleton constraints is bounded by the number of distinct edges in all generated paths. This property allows us to deal with large graphs, as we will only generate the necessary singleton constraints. Furthermore, many use-cases exhibit the property that G is an acyclic directed graph. In that case, the SHORTEST PATH problem can be solved in linear time. By independence of the pricing problems, they can be solved in parallel (and possibly with GPU-acceleration).

3.3 Modular Software Architecture

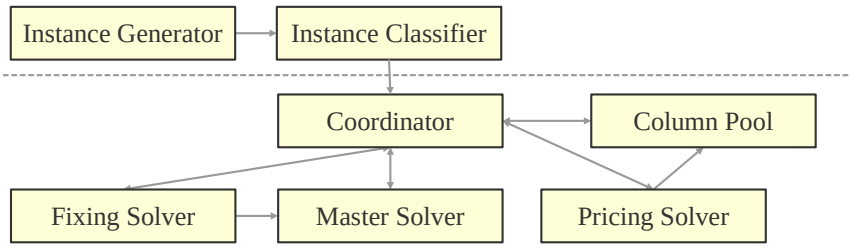


Figure 1: Architecture of the solver

Figure 1 displays the overall architecture of the solver. Its main purpose is that expert users are able to fine-tune the solver towards the specialties of their use-cases.

The initial module is the *instance generator*, which is a framework that helps to transform the input of some given use-case into an instance of SET CONSTRAINED FLOW. The user simply has to define the inter-dependencies between the objects of the use-case. The actual graph generation is taken care of by the module, by making use of parallelization to speed up the process. Finally, the user needs to specify the set constraints to complete the definition of the instance. Given that, the *instance classifier* detects properties that will be exploited in the subsequent modules and hence affects the overall configuration of the optimizer.

The main module of the solver is the *coordinator*: It instructs the behavior of the remaining modules, observes if a terminating solution has been found or if time-limits have been exceeded. Following the column generation approach, the constraints of the instance are split into local- and global constraints, respectively. The *master solver* hosts the global constraints and always solves some given linear program. The corresponding dual solution yields reduced cost coefficients on the edges, which are

Table 1: Real-world CREW SCHEDULING results

Nr.	Name	D, G	T	m	k	Time	LB	Value	Gap [%]
1	Avg depot 1	$1, 2$	556	16190245	370	575s	3835.68	3848	0.33
2	Avg depot 2	$1, 3$	784	7068272	515	205s	7331.53	7497	2.20
3	Large depot 1	$1, 8$	2096	150605068	1334	3.03h	37880.79	38342	1.20
4	Country 1	$34, 69$	12982	1350972506	11871	42.9h	1177.87	1200	1.87
5	Country 2	$34, 66$	15192	853239517	9241	27.1h	1244.69	1256	0.90
6	Country 3	$34, 67$	20041	2315427477	12244	104.0h	1158.01	1213	4.74

reported to the pricing solver by the coordinator. We currently have connectors to CPLEX, Gurobi, HiGHS, and SCIP to implement the solution of the linear programs. The *pricing solver* solves the subproblem, which is given by the graph, the local singleton constraints, and the vertex-balances. In many cases, these problems are simply SHORTEST PATH problems of some variety (and determined by the instance classifier). The pricing solver is responsible for most of the speed and scalability of our approach: Especially on directed acyclic graphs, SHORTEST PATH can be solved in linear time and a large number of paths can be generated in parallel. The paths are reported to the *column pool*, deciding which ones to keep and which ones to discard. The remaining columns are finally added to the linear program to be solved by the master solver. Finally, the *fixing solver* decides which variables or constraints in the linear program are rounded, frozen, or released. This module can implement branch-and-bound schemes (if need be) or fixing heuristics originating from insights into the use-case at hand.

4 Real World Application: Crew Scheduling

This section originates from our work at the Swiss Federal Railways. Our optimizer is used for annual driver- and conductor-planning and for strategic simulations.

The CREW SCHEDULING problem asks to partition a set of tasks into a set of shifts, while respecting temporal consistency, geographic consistency, break requirements, and other labor rules. Drivers are organized into depots, where each depot has one or more driver groups with different skill sets. The objective is to minimize a weighted sum over the shifts. The problem has attracted substantial research, see, e.g., [17, 18, 19, 20, 21, 22]. The dominant approach is column-generation [23], with SET COVER as the master problem [17, 20, 22] and the NP-complete RESOURCE CONSTRAINED SHORTEST PATH as the pricing problem [24, 20, 25, 22].

We applied the SET CONSTRAINED FLOW approach as follows: The graph captures the possible transitions between the tasks. States stored on the vertices represent a task and the relevant history of a shift that contains it. A source vertex marks the start of a planning day at some location, while a target vertex marks its end there. By construction, each path from the source to the target represents a valid shift. Set constraints ensure that at most one unit of flow passes through each vertex. Hence any integral flow decomposes into disjoint paths. For each task i, there is a constraint S_i defined of the edges that lead to the vertices containing the task. The set bound constraints $\ell_i = 1$ and $u_i = 1$ ensure that each task occurs in exactly one vertex with positive flow.

Notice that this construction yields that our pricing problems are now ordinary SHORTEST PATH problems (even on an acyclic directed graph), instead of the usual RESOURCE CONSTRAINED SHORTEST PATH. This advantage is paid by having to deal with large graphs. However, our parallelized solvers for SHORTEST PATH and the availability of computational resources with large memory outweighs this drawback. Thus, our strategy is to generate a lot of promising paths very quickly and filter them before we solve the linear program. In addition, we developed special fixing strategies, that exploit insights into the CREW SCHEDULING problem.

Our benchmark instances cover a variety of real-world scenarios, ranging from single-depot to full-country optimizations. Table 1 summarizes our results. Therein, "D, G" count the number of depots and groups in the instance, respectively; "T" counts the number of tasks. The column m is the number of edges, while column k shows the number of non-singleton constraints after preprocessing. The overall results substantiate that our technology is capable to solve instances with billions of edges in high quality. The experiments were run on a HPE ProLiant ML350 Gen9, with two Intel Xeon E5 CPU having 14 cores, each, and 512 GB of main memory.

Table 2: KNAPSACK benchmarks

Nr.	Name	I	B	m	k	Time	Opt	Value	Gap [%]
1	n_400	400	10^6	401	402	4s	602022	601925	0.016
2	n_600	600	10^6	601	602	7s	603289	603284	0.00082
3	n_800	800	10^8	801	802	15s	99762610	99138454	0.0062
4	n_1200	1200	10^{10}	1201	1202	36s	9999255664	9954720602	0.0044

Table 3: SET COVER benchmarks

Nr.	Name	U	$\mathcal{F}$	m	k	Time	LB	Value	Gap [%]
1	rail516	516	47311	47312	47827	18s	182.0	182	0.00
2	rail2536	2536	1081841	1081842	1084377	4.78h	688.39	693	0.66

5 Modeling Problems and Experimental Results

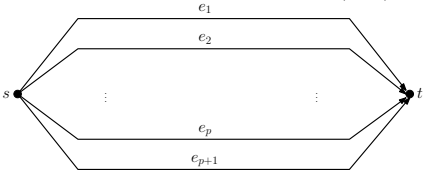
Here we exemplify the modeling of two classical combinatorial optimization problems. Both of them only require a single decision gadget, which is defined as follows: The directed graph $D(c, p)$ is called *decision graph*, see Figure 2, of up to c choices amongst p options. It has two vertices $V = \{s, t\}$ with balances $b_s = c$ and $b_t = -c$. There are $p+1$ directed edges $E = \{e_i = (i, s, t, c_i, w_i, \mathbb{B}) : i = 1, \ldots, p+1\}$. Edge e_{p+1} has a capacity of c, while the remaining edges have a capacity equal to one. The cost structure c_i and weight structure w_i is dictated by the application. Our experiments were run on hardware with one Intel i7 CPU, having 8 cores, and 64 GB main memory.

5.1 Knapsack

The problem KNAPSACK is NP-complete (in its decision version) and one of the classics described in [26]. In the optimization version, we are given an overall *budget* B, and n items $I = \{I_1, I_2, \ldots, I_n\}$, where any item is a tuple $I_i = (i_i, b_i, p_i)$ with *unique identifier* i_i, *budget* b_i, and *profit* p_i. Select a subset $K \subseteq I$, such that $\sum_{k \in K} b_k \leq B$, as to maximize $\sum_{k \in K} p_k$.

The reduction from SET CONSTRAINED FLOW is: Consider the decision graph $D(n, n)$, where each edge e_i corresponds to an item i with weight $w_i = b_i$ and cost $c_i = -p_i$. The edge e_{n+1} has weight $w_{n+1} = 1$ and cost $c_{n+1} = 0$. There is one constraint $S_0 = (0, \{e_1, \ldots, e_n\}, 0, B)$ and n singleton constraints $S_i = (i, \{e_i\}, 0, w_i)$ yielding that each such edge has a capacity equal to one. Notice that any feasible integral flow on the edges $e_1, \ldots, e_n$ corresponds to a selection of items not exceeding the budget B (due to constraint S_0) and whose profit is equal to the negative cost of the flow. We have chosen benchmark instances from the set given in [27]. The optimal values of these instances are known. See Table 2 for our results.

Figure 2: Decision graph $D(c, p)$



5.2 Set Cover

One of the classical NP-complete problems (in its decision version) given in [26] is SET COVER, and defined as follows. Given an n-element set, called *universe* $U = \{1, \ldots, n\}$ and a k-element family of U-subsets $\mathcal{F} = \{F_1, \ldots, F_k\}$ with *cost* $h(F_i) = h_i \geq 0$ for each $F_i \in \mathcal{F}$, find a subfamily $\mathcal{C} \subseteq \mathcal{F}$, called *cover*, such that each element $u \in U$ is contained in some set $C \in \mathcal{C}$. The objective is to minimize the total cost $h(\mathcal{C}) = \sum_{C \in \mathcal{C}} h(C)$.

We construct an instance of SET CONSTRAINED FLOW as follows: Consider the decision graph $D(k, k)$, where each edge e_i corresponds to a set F_i with weight $w_i = 1$ and cost $c_i = h_i$. The edge e_{n+1} has weight $w_{n+1} = 1$ and cost $c_{n+1} = 0$. There are n constraints $S_j = (j, \{e_i : j \in F_i\}, 1, k)$ and n singleton constraints $S_\cdot = (\cdot, \{e_i\}, 0, w_i)$ yielding that each such edge has a capacity equal to one. Notice that any feasible integral flow on the edges $e_1, \ldots, e_k$ corresponds to a selection of sets covering each element of U and whose cost is equal to the cost of the flow. The benchmark instances are taken from [28] and were discussed also in [29]. See Table 3 for our results.

References

[1] B. H. Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer-Verlag, New York, NY, 2012.

[2] Guy. Desaulniers, Jacques. Desrosiers, and Marius M. Solomon. *Column generation / edited by Guy Desaulniers, Jacques Desrosiers, Marius M. Solomon.* GERAD 25th anniversary series ; 5. Springer, New York, 2005.

[3] Eduardo Uchoa, Artur Pessoa, and Lorenza Moreno. Optimizing with Column Generation: Advanced branch-cut-and-price algorithms (Part I). Technical Report L-2024-3, Cadernos do LOGIS-UFF, Universidade Federal Fluminense, Engenharia de Produção, August 2024.

[4] GAMS. An overview of math programming solvers, 2022.

[5] Thorsten Koch, Timo Berthold, Jaap Pedersen, and Charlie Vanaret. Progress in mathematical programming solvers from 2001 to 2020. *EURO Journal on Computational Optimization*, 10:100031, 2022.

[6] IBM. *User's Manual for CPLEX*, version 22.1.2 edition, 2024.

[7] Gurobi. *Gurobi Optimizer Reference Manual*, 2025.

[8] Fair Isaac Corporation. *FICO Xpress Optimization*, 2025.

[9] Hexaly. *Hexaly 14.0 documentation*, 2025.

[10] SCIP Developers. *SCIP Documentation: Overview (User's Manual)*, 2025.

[11] HiGHS. *HiGHS - High Performance Optimization Software for Linear Optimization*, 2025.

[12] HiGHS Developers / ERGO-Code. *HiGHS Developer Guide — GPU Acceleration (cuPDLP-C)*, 2025. HiGHS GPU guide; documents cuPDLP-C support since v1.10.0.

[13] HiGHS. *HiGHS - Newsletter 25.0*, 2025.

[14] Google. *About OR-Tools*, 2024.

[15] Kexiong Yu, Hang Zhao, Yuhang Huang, Renjiao Yi, Kai Xu, and Chenyang Zhu. DISCO: Efficient diffusion solver for large-scale combinatorial optimization problems, 2025.

[16] Artelys. *Artelys Knitro User's Manual*, 2025.

[17] B.M. Smith and A. Wren. A bus crew scheduling system using a set covering formulation. *Transportation Research*, 22A:97–108, 1988.

[18] J.E. Beasley and B. Cao. A tree search algorithm for the crew scheduling problem. *Journal of Operational Research*, 94:517–526, 1996.

[19] R. Borndörfer, M. Grötschel, and A. Löbel. Optimization of transportation systems. Technical Report ZIB-Report 98-09, Zuse-Institut Berlin, 1998.

[20] A.S.K. Kwan. *Train Driver Scheduling*. Phd thesis, University of Leeds, 1999.

[21] G. P. Silva, N. D. F. Gualda, and R. Kwan. Bus scheduling based on an arc generation – network flow approach. In *8th International Conference on Computer-Aided Scheduling of Public Transport - CASPT 2000*, 2000.

[22] S. Jütte. *Large-Scale Crew Scheduling: Models, Methods, and Applications in the Railway Industry*. Springer Verlag, 2012.

[23] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960.

[24] M. Desrochers. *La Fabrication d'horaires de travail pour les conducteurs d'autobus par une méthode de génération de colonnes*. Ph.d thesis, Université de Montréal, Centre de recherche sur les Transports, Canada, 1986.

[25] S. Irnich and G. Desaulniers. Shortest path problems with resource constraints. In G. Desaulniers, J. Desrosiers, and M.M. Solomon, editors, *Column Generation*. Springer, Boston, MA, 2005.

[26] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[27] Jorik Jooken, Pieter Leyman, and Patrick De Causmaecker. A new class of hard problem instances for the 0–1 knapsack problem. *European Journal of Operational Research*, 301(3):841–854, 2022.

[28] John E. Beasley. *OR-Library*, 1990–2018.

[29] Shunji Umetani, Masanao Arakawa, and Mutsunori Yagiura. A heuristic algorithm for the set multicover problem with generalized upper bound constraints. In Giuseppe Nicosia and Panos Pardalos, editors, *Learning and Intelligent Optimization*, pages 75–80, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.