Columnwise neighborhood search: A novel set partitioning matheuristic and its application to the VeRoLog Solver Challenge 2019

Caroline J. Jagtenberg | Oliver J. Maclaren | Andrew J. Mason | Andrea Raith | Kevin Shen | Michael Sundvick

Department of Engineering Science, University of Auckland, Auckland, New Zealand

Correspondence
Caroline J. Jagtenberg, Department of Engineering Science, University of Auckland, 70 Symonds Street, Auckland 1010, New Zealand.
Email: c.jagtenberg@auckland.ac.nz

Funding information
This research was supported by the Nederlandse Organisatie voor Wetenschappelijk Onderzoek, Grant/Award Number: Rubicon grant [project number 019.172EN.016]

Abstract
This article reports on an approach for the VeRoLog Solver Challenge 2019: the fourth solver challenge facilitated by VeRoLog, the EURO Working Group on Vehicle Routing and Logistics Optimization. The authors were awarded third place in this challenge. The routing challenge involved solving two interlinked vehicle routing problems for equipment: one for distribution (using trucks) and one for installation (using technicians). We describe our solution method, based on a matheuristics approach in which the overall problem is heuristically decomposed into components that can then be solved by formulating them as set partitioning problems. To solve these set partitioning problems we introduce a novel method we call “columnwise neighborhood search,” which allows us to explore a large neighborhood of the current solution in an exact manner. By iteratively applying mixed-integer programming methods, we obtain good quality solutions to our subproblems. We then use a simple local search “fusion” heuristic to further improve the solution to the overall problem. Besides introducing and discussing this solution method, we highlight the problem instances for which our approach was particularly successful in order to obtain general insights about our methodology.

Keywords
columnwise neighborhood search, local search, matheuristics, routing problem, set partitioning

1 | INTRODUCTION

The VeRoLog Solver Challenge 2019 was facilitated by VeRoLog, the EURO Working Group on Vehicle Routing and Logistics Optimization and organized in cooperation with the Dutch company ORTEC. The challenge took place over the course of approximately 4 months, and teams’ performances were measured on both public and unpublished problem instances. This was the fourth solver challenge organized by VeRoLog.

The problem central to the challenge was first introduced in [15]. In essence, it consists of two interlinked vehicle routing problems (VRPs). The challenge rules and setup are found in [23]; for the full documentation of the challenge, see [22].

The authors of this paper constitute the team “COKA coders,” and were awarded third prize in the challenge finals. In this paper, we report on our contribution.
We adopted a matheuristics approach, that is, a hybridization of heuristic methods with exact, mathematical programming tools, in particular mixed-integer programming (MIP)-based methods. We used a heuristic decomposition of the full problem into parts that were then formulated as set partitioning problems and solved using a novel matheuristic we call columnwise neighborhood search. Our solution strategy is described in more detail below.

Our approach was different from that of all other participating teams in the challenge. In particular, our approach is the only one that involves integer programming. The first prize in this challenge was awarded to Benjamin Graf (team UOS) with an adaptive large variable neighborhood search. The second prize was won by Martin Geiger (team mjg) using a heuristic search algorithm.

We next summarize existing literature on VRPs, focusing in particular on matheuristic approaches utilizing MIP methods and set partitioning/set covering-based formulations.

### 1.1 Previous work

Classical VRPs appear in many applications and have been intensively studied since their introduction in 1959 by Dantzig and Ramser [7]. Although significant effort has been made to extend exact methods to handle larger problems, as well as variations such as VRPs with time windows (VRPTWs), only relatively small problems may be solved exactly and hence heuristic/meta-heuristic approaches are frequently employed. A comprehensive review of the standard literature, including a number of variations of VRPs, is given by Toth and Vigo [21]. Complex VRP problems are often termed “rich”; a survey of the various “dimensions of richness” of VRPs is given by Drexl [9], while a more systematic taxonomy and definition of rich VRPs is given by Lahyani et al. [19]. As Puchinger and Raidl [20] note, probably the most successful examples of exact and (meta-)heuristic approaches to general combinatorial optimization problems are integer programming and (variations on/extensions of) local search, respectively. We take this latter category to include meta-heuristic extensions designed to escape local optima, for example, repeated local search strategies.

Here we focus primarily on so-called “matheuristic” approaches to rich VRPs. Rather than resort entirely to (meta-)heuristic methods, matheuristic approaches aim to blend traditional exact methods with heuristic ones, for example heuristically decomposing the overall problem into subproblems that may be solved by exact methods and recombined to give a heuristic solution to the original problem. As discussed by Fischetti and Fischetti [10], matheuristics do not constitute a rigid paradigm, but rather a conceptual framework that aims to utilize existing methods and results from exact mathematical programming, while extending these to deal with more realistic problems. Nevertheless, some attempts have been made to classify common classes of matheuristic approaches, and we give a brief overview of these classifications next, both in a general context and in the context of VRPs. Furthermore, many hybrids of exact and heuristic methods are based on some combination of two important strands of each area noted above: integer programming and local search.

Ball [2] classifies matheuristics in general according to

- Decomposition techniques.
- Modifications of exact methods to produce approximate solutions/Relaxation-based methods.
- Improvement heuristics.

As discussed by Ball [2], another important classification is the separation into “construction” or “improvement” heuristics, that is, whether the method starts “from scratch” or takes an existing solution as input. Puchinger and Raidl [20] use a simpler classification of methods for combining exact and (meta-)heuristics methods, dividing them into “collaborative” and “integrative” categories. In the former, “the algorithms exchange information, but are not part of each other,” while in the latter “one technique is a subordinate embedded component of another technique,” that is, there is a clear “master” and “subproblem” division.

Motivated by the classification given by Ball [2], but developed in the specific context of VRPs and their extensions, Archetti and Speranza [1] classify matheuristics according to

- Decomposition approaches.
- Heuristic branch-and-price/column generation-based approaches.
- Improvement heuristics.

Again in the context of VRPs and extensions, Doerner and Schmid [8] classify matheuristics according to

- Decomposition techniques.
- Set-covering/partitioning formulations.
- Local branching.

“Decomposition” is used in a broad sense by the above authors, referring to for example, inherent or “natural” decompositions into qualitatively distinct subproblems, and/or to practical decomposition strategies that are typically based on
approximating a hard problem using the solutions of two or more easier subproblems. This is also how we use the term here. An example of the former type of decomposition would be the Passenger-Then-Vehicle-Then-Crew decompositions often adopted for airline scheduling/rostering problems, while an example of the latter would be the cluster-first, route-second decompositions adopted in early approaches to VRPs [2].

Archetti and Speranza [1]’s “heuristic branch-and-price/column generation-based approaches” category is essentially a more advanced form of the “set partitioning/set covering” category used by Doerner and Schmid [8], due to the fact that the former class of methods usually relies on some version of a set partitioning/set covering formulation (as noted in [1]). An interesting early set covering heuristics for VRP problem is the petal algorithm [11], which creates many alternative routes and then chooses the best subset using set partitioning. According to [21, p. 95], such an approach involving set partitioning tends to work well for problems with many relatively short routes.

“Local branching,” as used above by Doerner and Schmid [8], refers to an improvement heuristic that can be applied when solving hard mixed-integer programs (MIPs). It thus provides a general approach for introducing local search to integer programming. Combining local search with exact methods offers the opportunity to explore much larger neighborhoods than usually considered in heuristic approaches, and to find optimal solutions within these neighborhoods.

The approach we present in this paper includes a MIP-based improvement heuristic that we term “columnwise neighborhood search”. Unlike local branching, this algorithm is tailored for set-partitioning problems, and thus integrates two of the algorithm classes identified above. Columnwise neighborhood search allows us to exploit modern MIP solvers such as Gurobi [16] to efficiently explore very large neighborhoods, making it easy to apply local search to help solve large set partitioning problems. Columnwise neighborhood search has been used by the authors to solve a number of problems in different areas. Crowder and Mason [6] implemented columnwise neighborhood search for a census districting problem, while Waugh and Mason [24] applied the same approach to a vehicle scheduling problem. More recently Cleland et al. [5] have adapted the approach for use with column generation to solve staff scheduling problems.

Our approach also utilizes decomposition approaches, including the following:

- Global decomposition of the original problem into qualitatively distinct subproblem types (i.e., technician and truck routing problems) that can each be formulated as MIPs and are solved sequentially (technician-first-truck-second).
- Further decomposition of the technician subproblem into distinct routing and scheduling phases, with the first phase generating routes that are then assigned to days in the second phase.
- Local decomposition/relaxation of the resulting subproblems using set covering/partitioning formulations of the MIPs and heuristically-generated columns.

We also use a number of construction and improvement heuristics at various stages to help piece together these component strategies and address the problem of scheduling activities over a time horizon. The technician problem requires scheduling technicians to days, and this requirement is only included indirectly in the set partitioning problem via heuristic constraints restricting the number of days worked by each technician. A simple repair heuristic is then used to ensure feasible solutions to the technician subproblem. Finally, a simple local search heuristic is used as a “fusion” method to combine and modify the separate truck and technician solutions. This targets, in particular, the reduction of idle costs (as described in the next section) which, when excessively large, essentially represent costs associated with solving the truck and technician subproblems separately.

The rest of this paper is structured as follows. In Section 2 we define the problem, followed by our solution as submitted for the solver challenge finals in Section 3. We report the results in Section 4 and finish with a discussion in Section 5.

2 | PROBLEM DESCRIPTION

The problem of the VeRoLog Solver Challenge 2019 was published in [15] and is available online in [23] (which includes extra information, such as the format of the instance data and how the final ranking of the teams is determined). We briefly recap this below.

The challenge problem combines distribution and subsequent installation of vending machines. Several types of machines must first be delivered within customer-dependent delivery windows, and then installed by a technician. Installation must take place as soon as possible after delivery, that is, there is a cost associated with delays between delivery and installation. The planning horizon consists of a number of consecutive days, and there is one depot location where all the machines are located at day 0.

There is no restriction on the number of trucks that can be used for machine delivery. Each truck has a volume capacity, however, and each vending machine type has a given volume that it occupies in a truck. A daily route for each truck starts and ends at the depot, but trucks can return to the depot several times during a day. The total distance that a truck can travel per day is limited to a given maximum.
Installation of a machine takes place after it is delivered by a truck. This installation is carried out by technicians with different skill sets, and these skill sets determine which kinds of machines a technician can install. A penalty is charged for every full day a machine is idle, that is, delivered to the customer but not yet installed. Technicians can work at most five consecutive days, after which they are required to take 2 days off. Their routes start and end at their own particular home location. There is a maximum daily travel distance as well as a maximum number of requests a technician can carry out per day.

The objective is to find a feasible solution at minimal total cost. The total cost is defined as a weighted sum of:

- distance traveled by truck
- a cost for using a truck for a day
- a cost for the truck fleet size (i.e., maximum number of trucks used on any day)
- distance traveled by a technician
- using a technician for a day
- using a technician at all during the planning horizon
- every full day a machine is idle, with different weights for each machine type

The weights for each of these cost components are specified in each problem instance. This means that different instances emphasize different aspects of the problem, which adds to the challenge of constructing one solution algorithm that performs well across all instances.

Problem instances were distributed by the challenge organizers at three different times. The first set distributed was called “ORTEC early,” and was disclosed early in the competition. At a later stage, another set called “ORTEC late” was provided. Finally, a set “ORTEC hidden” was used to evaluate the performance of the different teams in the finals. This set was not disclosed until the challenge concluded. Each of these sets consist of 25 problem instances with varying cost structure. Throughout this paper, whenever we mention a problem instance, it is always one from the above-mentioned sets.

To formally define this problem, we next formulate it as a mixed integer program (MIP).

### 2.1 Full MIP formulation

In this section we show how to formulate the challenge problem as a MIP. Because of its potentially large size, we did not implement this full model for the competition as we could not guarantee that a solution could be found within the prescribed run time. However, this formulation is helpful in motivating a number of subproblems that we formulated and solved as part of a heuristic decomposition of the problem.

Let $M$ be a set of machine requests that need to be serviced. A machine request $m \in M$ specifies a number of machines (all of the same type) that have to be delivered to a customer site between an integer release date $\hat{r}_m$ and an integer due date $\hat{d}_m$. That is, request $m \in M$ must be serviced by a truck on some day $d \in \{\hat{r}_m, \hat{r}_m + 1, \ldots, \hat{d}_m\}$, and then by a technician on one of days $d + 1, d + 2, \ldots, l$, where $l$ denotes the last day of the planning horizon.

Let $R$ be a set of “daily truck routes,” each of which is feasible for a single truck to execute in 1 day. A particular daily truck route $r \in R$ specifies an (unordered) feasible set of machine requests, given by $r = \{m_1, m_2, \ldots, m_n\} \subseteq M$. We assign a daily truck route $r \in R$ a release date $\hat{r}_r = \max_{m \in r}\hat{r}_m$ and a due date $\hat{d}_r = \min_{m \in r}\hat{d}_m$. We only construct daily truck routes that are feasible in the sense that all truck rules are satisfied, and $\hat{r}_r \leq \hat{d}_r$, $\forall r \in R$. As all trucks are identical and operate from the same depot, the distance $d_r$ of a truck route $r$ can be found by solving a single VRP. This VRP takes into account the location of the depot, the machine delivery requests in that route, the daily and per-kilometer costs of operating a truck and the truck capacity. Note that, due to the capacity constraint, a daily truck route $r$ will often contain multiple stops at the depot to collect machines, with each stop being followed by the delivery of the collected machines. The cost $c_r$ of using route $r \in R$ is then given by some fixed daily cost (termed “TRUCK_DAY_COST” in the competition description) for using a truck, plus a distance-cost given by $\text{TRUCK\_DISTANCE\_COST} \ast d_r$, that is, $c_r = \text{TRUCK\_DAY\_COST} + \text{TRUCK\_DISTANCE\_COST} \ast d_r$.

There is a further “fleet” cost that is given by $\text{TRUCK\_COST} \ast n_{\text{trucks}}$, where $n_{\text{trucks}}$ is the maximum number of trucks used on any day over the full planning horizon.

As detailed above, after a machine is delivered to a site, it needs to be serviced by a technician. For ease of notation, we refer to these technicians as people, of which we have a set $P$. Unlike the trucks, the people are not identical, and so each person $p \in P$ has their own unique home location from which they start and finish on each day they work. They also have their own skill set that determines which machine requests they can service, and their own maximum daily travel distance. If a person $p \in P$ is used at all in the solution, then they are paid both a one-off fixed amount $c_{\text{person}} = \text{TECHNICIAN\_COST}$ plus a variable amount $\text{TECHNICIAN\_DAY\_COST}$ for each day they work, plus a per kilometer cost $\text{TECHNICIAN\_DISTANCE\_COST}$ for the distance they travel. These costs are the same for all people. A day’s work for a person involves visiting a set of machine requests by following a tour defined by the solution to a traveling salesman problem (TSP). Let $T$ denote the set of all possible daily tours that can be worked by the people, where a tour $t \in T$, $t = \{m_1, m_2, \ldots, m_d\} \subseteq M$ specifies a feasible set of machine
requests for a person to visit on 1 day. We note that some tours will be infeasible for some people because the tour is too long, has too many requests, or includes at least one machine of a type for which the technician does not possess the requisite skill. We let \( T^p \subseteq T \) denote the subset of tours that are feasible for person \( p \in P \), and let \( h^p_r \) denote the cost of person \( p \) completing tour \( t \). This cost, which includes both the daily and per-kilometer costs, is person specific as it depends on a TSP solution that includes the machine locations and the person’s home location. As we did with truck routes, we assign a tour \( t \in T \) a release date \( \hat{r}_t = 1 + \max_{m \in \mathcal{M}} \hat{r}_m \). (There is an implied due date in the sense that all machines need to be serviced by the last day, \( l \), in the planning horizon.)

A schedule \( s \) is defined by a set of days \( d \in D \) that can be worked by any single technician. Multiple technicians can work the same schedule. Such a schedule specifies the potential on-days, implying the guaranteed off-days. Note that we only need to consider “efficient” schedules, that is, schedules that would be illegal if any day off were to be worked. We denote the set of all feasible efficient schedules by \( S \).

To define the MIP, we introduce the following decision variables and constants (some of which have been introduced above, but are repeated here for clarity):

1. Decision variable \( x_{r,d} \) = 1 if daily truck route \( r \in R \) is performed on day \( d \in D \), and 0 otherwise. Note that we enforce \( x_{r,d} = 0 \) (and thus effectively remove \( x_{r,d} \)) if a route \( r \) cannot be worked on day \( d \), that is, if \( d < \hat{r}_r \) or \( d > \hat{d}_r \).
2. Decision variable \( n_{\text{trucks}} \) is the truck fleet size, that is, the maximum number of trucks used on any single day over the full planning horizon.
3. Decision variable \( y^p_{r,d} \) = 1 if person \( p \in P \) performs tour \( t \in T^p \) on day \( d \in D \), and 0 otherwise. Note that we enforce \( y^p_{r,d} = 0 \) for all \( (t,d) : d < \hat{r}_r \) for all \( p \in P \), and for all tours \( t \notin T^p \) that are illegal for technician \( p \).
4. Decision variable \( c^p_s \) = 1 if we select schedule \( s \in S \) for person \( p \in P \), and 0 otherwise. Note that a person may remain unused if we do not select any schedule at all for them.
5. Decision variable \( w_m \) is the number of idle days for machine request \( m \in M \), being the number of full integer days that elapse between its delivery and installation.
6. Constant \( a_m = 1 \) if route \( r \in R \) includes machine \( m \in M \), and 0 otherwise.
7. Constant \( c_{\text{truck}} \) is the cost (including the TRUCK\_DAY\_COST and TRUCK\_DISTANCE\_COST components) of a truck route \( r \in R \).
8. Constant \( c_{\text{person}} \) = TECHNICIAN\_COST is the one-off cost of using a truck at all during the planning horizon; this is used to calculate the fleet size cost.
9. Constant \( c_{\text{person}} \) = TECHNICIAN\_COST is the cost of having a technician working at all during the planning horizon; this is used to calculate the workforce size cost, where we pay \( c_{\text{person}} \) for each schedule that is worked.
10. Constant \( b_{t,m} = 1 \) if person tour \( t \in T \) includes machine request \( m \in M \), and 0 otherwise.
11. Constant \( h^p_r \) denotes the cost of person \( p \) completing some tour \( t \in T^p \subseteq T \) that is feasible for them. This cost includes the TECHNICIAN\_DAY\_COST and TECHNICIAN\_DISTANCE\_COST components.
12. Constant \( e_{s,d} = 1 \) if and only if schedule \( s \in S \) includes day \( d \in D \) as a day that may be worked.
13. Constant \( c_m \) is the cost paid for each idle day for machine request \( m \in M \).

The exact solution to this problem can then be found by solving the following MIP.

\[
\begin{align*}
\text{min} & \quad c_{\text{truck}} n_{\text{trucks}} \quad \text{truck fleet size cost} \\
& + \sum_{r \in R} c_r \sum_{d \in D} x_{r,d} \quad \text{truck daily and distance costs} \\
& + c_{\text{person}} \sum_{p \in P} \sum_{s \in S} c^p_s \quad \text{person workforce size cost} \\
& + \sum_{p \in P} \sum_{t \in T^p} c^p_r y^p_{r,d} \quad \text{person daily and distance costs} \\
& + \sum_{m \in M} c_m w_m \quad \text{total idle machine days cost} \\
\text{s.t.} & \quad \text{Calculate fleet size } n_{\text{trucks}} : \\
& \sum_{r \in R} x_{r,d} \leq n_{\text{trucks}} \quad \forall d \in D
\end{align*}
\]
Every machine request \( m \) is in one selected truck route:
\[
\sum_{r \in R} \sum_{d \in D} a_{r,m} x_{r,d} = 1 \quad \forall m \in M
\] (7)

Up to one schedule for each person:
\[
\sum_{s \in S} z^p_s \leq 1 \quad \forall p \in P
\] (8)

No person works a tour on day \( d \) unless this day is a scheduled work day:
\[
\sum_{s \in S} y^p_{t,d} = \sum_{s \in S} e_{s,d} z^p_s = 1 \quad \forall d \in D, p \in P
\] (9)

Every machine request \( m \) is installed by one person:
\[
\sum_{p \in P} \sum_{t \in T} \sum_{d \in D} b_{t,m} y^p_{t,d} = 1 \quad \forall m \in M
\] (10)

A person cannot install a machine request before it is delivered by a truck:
\[
\sum_{p \in P} \sum_{t \in T} \sum_{d \in D} b_{t,m} y^p_{t,d} \leq \sum_{d'>d} \sum_{r \in R} a_{r,m} x_{r,d'} \quad \forall m \in M, d \in D
\] (11)

Calculate idle days between truck and person visits for machine request \( m \):
\[
w_m = \sum_{p \in P} \sum_{t \in T} \sum_{d \in D} d b_{t,m} y^p_{t,d} - \sum_{d \in D} \sum_{r \in R} a_{r,m} x_{r,d} \quad \forall m \in M
\] (12)

Exclude truck routes/person tours outside release/due dates or illegal for a person:
\[
x_{r,d} = 0 \quad \forall r \in R, d \in D : d < \hat{r}_r \text{ or } d > \hat{\hat{d}}_r
\] (13)
\[
y^p_{t,d} = 0 \quad \forall p \in P, t \in T, d \in D : d < \hat{t}_t \text{ or } t \notin T^p
\] (14)

Variable restrictions:
\[
x_{r,d} \in \{0, 1\} \quad \forall r \in R, d \in D
\] (15)
\[
n_{\text{trucks}} \in \mathbb{Z}^+
\] (16)
\[
z^p \in \{0, 1\} \quad \forall p \in P, s \in S
\] (17)
\[
y^p_{t,d} \in \{0, 1\} \quad \forall p \in P, t \in T^p, d \in D
\] (18)
\[
z^p_s \in \{0, 1\} \quad \forall s \in S, p \in P
\] (19)
\[
w_m \in \mathbb{Z}^+ \quad \forall m \in M
\] (20)

3 | SOLUTION

In this section we describe our solution as implemented for the Solver Challenge. We emphasize again that we did not implement the full MIP as described in Section 2.1, as we were not confident that this would be able to handle instances of the magnitude required for the competition within the available run time. Instead we adopted a matheuristic approach, which we next describe.
and link to the MIP from Section 2.1. The implementation of our matheuristic was done in Python. Moreover, we used Gurobi as a MIP solver, which we restricted to a single core (as per competition rules [23]).

The rest of this section is organized as follows. We first describe how we decomposed the problem at hand into subproblems. Then, we introduce our “columnwise neighborhood search” heuristic, and describe how we used this to find solutions to the two key subproblems. Finally, we describe our fusion method which uses the solutions to the subproblems to compose a solution to the overall problem.

### 3.1 Decomposition

As our full MIP formulation shows (see Section 2), the problem consists of two subproblems associated with the two key resources, trucks and people. The first subproblem addresses the construction and selection of delivery routes for trucks (the \( x_{r,d} \) and associated variables). The second addresses the creation and selection of installation tours for people (the \( y_{p,d} \) and associated variables). These two subproblems have their own objective function terms (Equations (1) and (2) for trucks, and (3) and (4) for people) and their own associated constraints (Equations (6), (7), (13) for trucks, and Equations (8), (9), (10), (14) for people). These subproblems are connected through the machine delivery and installation dates, as given in linking constraints (11) and (12) and associated idle costs (Equation (5)). Our approach assumes that this coupling between these two subproblems is relatively weak, and so we can still form good solutions by solving the truck and people problems independently. Of these two subproblems, the technician problem is the more complicated problem, due to its nonhomogeneous fleet and additional scheduling rules. We therefore chose to decompose the technician problem again, and split it into a routing subproblem and a scheduling subproblem. This avoided the need for linking constraints between tours, and the consequential increase in model size (and potential run time) that this would create. For trucks, a further decomposition was not necessary.

An intuitive order of solving these decomposed problems is perhaps chronologically: first the delivery of a request has to be planned, followed by its subsequent installation. We, however, decided to start by planning all the installations, that is: we solve the technician problem first. By first finding good tours for technicians, followed by assigning those tours to days, we then solve the truck problem as constrained by the solution for technicians.

The benefit of our approach is that it allows us plan the technicians, which using our approach constitutes the harder subproblem, without taking into account the idle costs. The idle costs can then be modeled as part of the truck problem. The remainder of this section is organized as follows. In Section 3.2 we formally define the columnwise neighborhood search method that we will be using use to solve the various set partitioning problems we use. This is followed by a description of how we applied columnwise neighborhood search to solve the technician subproblems (Section 3.3) and the truck subproblem (Section 3.4). Finally, we describe how we combined, or “fused” the solutions to the subproblems to construct a solution to the overall problem (Section 3.5).

### 3.2 Columnwise neighborhood search

A wide variety of problems, including the subproblems we consider here, can be formulated as set partitioning problems in which we need to partition n objects into subsets; in our case we partition machine requests to form a day’s activities for a truck or a technician. One approach to solving these set partitioning problems is to enumerate a set \( P \) of (ideally all) feasible subsets (termed “parts”), and then create an integer program which uses binary variables \( x = (x_p, p \in P) \), \( x_p \in \{0, 1\} \forall p \in P \) to select the least cost feasible set of parts, where \( c(p) \) is the cost of using part \( p \in P \). Note that we use the term “feasible” here in two ways, both of which depend on the particular problem being modeled. First, we say that a subset if feasible (and thus defines a part \( p \in P \)) if the set of machine requests it contains satisfies the rules governing what a truck (or technician) can do in a day. Second, we need to determine if a solution \( x \) is feasible for the problem. Feasibility of \( x \) can usually be expressed in the form \( A^p x = b \) for some integer vector \( b \). In our case, we require every machine request to appear in exactly one part and so we have a pure set partitioning problem with \( b = 1 \). The matrix \( A^p = (a_1, a_2, \ldots, a_p) \) is a matrix of 0’s and 1’s formed from \( P \) in which the \( j \)th column \( a_j = (a^1_j, a^2_j, \ldots, a^n_j) \) represents the \( j \)th part \( p \) in \( P \), where \( a^k_j = 1 \) if part \( p \) contains object \( k \), and \( a^k_j = 0 \) otherwise. This gives rise to the following generalized set partitioning integer program \( GSPP(P) \):

\[
\text{GSPP}(P) : \min \ c(x) = \sum_{p \in P} c(p)x_p \\
\text{s.t.} \quad A^p x = b \\
x \in \{0, 1\}^{\mid P\mid}
\]
Often the number of possible feasible subsets is too large to be enumerated, and so finding an optimal solution using enumeration becomes intractable. One approach is to use branch and price [3] in which a column generator is used to create new improving columns by solving a column pricing subproblem. Implementation of a column generation approach is nontrivial, and typically requires the development of a customized branch and bound algorithm that allows the column generator to be called while solving the linear programs at each node in the branch and bound tree. A simpler approach is to enumerate not all, but instead just a promising subset of the columns, and then, given this promising subset \( P \), solve GSPP(\( P \)) to obtain a good (but not necessarily optimal) solution to the problem. Because integer programs can often be very slow to solve for large problem instances, practitioners instead often choose to use heuristic local search in which small changes are made to some existing solution in an attempt to find better local minima. Examples of local search methods include simulated annealing [18] and tabu search [13]. In local search, the possible changes that can be made to a solution are defined by a “neighborhood rule,” and might include, for example, moving an object from one part to another, or swapping an object between two parts. The set of all solutions \( y \) that can be formed from some solution \( x \) using the neighborhood rule is termed the neighborhood of \( x \), \( N(x) \). We say that \( x \) is a local minimum if \( c(x) \leq c(y) \) \( \forall y \in N(x) \). In general, a local search will be effective if these local minima are of good quality. Larger neighborhoods typically give better solutions, and so researchers have developed Very Large Scale (VLS) neighborhood approaches, an example of which is “Cyclic Exchange” [14] which defines a neighborhood for a set partitioning problem as a sequence of simultaneous object moves around an ordered sequence of parts.

We now introduce columnwise neighborhood search, which allows us to employ local search concepts to explore very large neighborhoods by exploiting the ability of modern integer programming solvers to solve large set partitioning problems. Columnwise neighborhood search is an example of a matheuristic in that it seeks to combine the benefits of integer programming with local search.

Let us assume that we have created some initial set of promising parts \( P \), and solved GSPP(\( P \)) to find some current solution. Standard local search would use some neighborhood rule to define neighboring solutions. Under columnwise neighborhood search, we do not define neighbors for the current solution, but instead use a “columnwise neighborhood rule” to define neighbors for each individual part (i.e., each column) in the solution. This rule lets us create a set of neighboring columns in an attempt to find better local minima. Examples of local search methods include simulated annealing [18] and tabu search [13]. In local search, the possible changes that can be made to a solution are defined by a “neighborhood rule,” and might include, for example, moving an object from one part to another, or swapping an object between two parts. The set of all solutions \( y \) that can be formed from some solution \( x \) using the neighborhood rule is termed the neighborhood of \( x \), \( N(x) \). We say that \( x \) is a local minimum if \( c(x) \leq c(y) \) \( \forall y \in N(x) \). In general, a local search will be effective if these local minima are of good quality. Larger neighborhoods typically give better solutions, and so researchers have developed Very Large Scale (VLS) neighborhood approaches, an example of which is “Cyclic Exchange” [14] which defines a neighborhood for a set partitioning problem as a sequence of simultaneous object moves around an ordered sequence of parts.

We now introduce columnwise neighborhood search, which allows us to employ local search concepts to explore very large neighborhoods by exploiting the ability of modern integer programming solvers to solve large set partitioning problems. Columnwise neighborhood search is an example of a matheuristic in that it seeks to combine the benefits of integer programming with local search.

Let us assume that we have created some initial set of promising parts \( P \), and solved GSPP(\( P \)) to find some current solution. Standard local search would use some neighborhood rule to define neighboring solutions. Under columnwise neighborhood search, we do not define neighbors for the current solution, but instead use a “columnwise neighborhood rule” to define neighbors for each individual part (i.e., each column) in the solution. This rule lets us create a set of neighboring columns \( N(p) \) for each part \( p \in P : x_p \neq 0 \) in the current solution \( x \) of GSPP(\( P \)). For example, a neighborhood \( N(p) \) for part \( p \) might be the set of all new parts formed by adding an object to \( p \) and/or removing an object from \( p \). Note that this rule does not need to consider feasibility of the full problem, but instead can just focus on ensuring each neighboring column we create is feasible for the problem. In our case, this means we must ensure each neighboring column is a feasible truck/technician route. To generate our next solution, we augment the current solution’s parts with the neighboring parts to form a new set:

\[
P' = \{ p \in P : x_p \neq 0 \} \cup \bigcup_{p \in P : x_p = 0} N(p)
\]

and then we solve GSPP(\( P' \)). The solution \( x' \) to this problem—and the associated parts \( \{ p \in P' : x_p \neq 0 \} \)—then define the best neighboring solution for \( x \) under the columnwise neighborhood rule \( N(\cdot) \). Note that the neighborhood being searched here is massive as we are effectively considering the simultaneous application of all possible variants of the columnwise neighborhood rule applied to all of the current parts. Even simple columnwise neighborhood rules such as “remove an existing object, and add a new object” result in neighborhoods that include and extend very large scale (VLS) neighborhood approaches such as cyclic exchange.

Like other local search procedures, columnwise neighborhood search is typically implemented as an iterative process that runs until the last iteration made no improvements. Let us assume that, at step \( k \), a partitioning problem defined over some set of parts \( P^k \) has been solved to generate a current solution \( x^k \). The “pure” columnwise neighborhood search as described above would discard all parts in \( P^k \) except those parts \( \{ t \in P^k : x^k_t \neq 0 \} \) chosen by the current solution \( x^k \). In practice, we typically do not discard the unused parts, but modify the above process so that the neighboring columns are added to the existing set of columns. That is, we form an augmented set of parts

\[
P^{k+1} = P^k \cup \bigcup_{p \in P^k : x^k_p \neq 0} N(p)
\]

(24)

Solving GSPP(\( P^{k+1} \)) using this augmented set of parts then gives the next solution \( x^{k+1} \). We repeat this process until no further improvement is found, that is, until \( x^k = x^{k-1} \), at which point \( x^k \) is a local minimum under the very large neighborhood induced by the columnwise neighborhood rule \( N(\cdot) \).

Columnwise neighborhood search is not restricted to integer programs, but can also be applied to linear programs. Assume that \( x^k \) is a fractional solution. As before, we can simply use (24) to add new neighboring columns to our \( A \) matrix, where these new columns are the neighbors of each column with a nonzero value in \( x^k \).

In applying columnwise neighborhood search to this competition problem, we switch between solving the original integer program and solving its linear program relaxation using the iterative process outlined in Figure 1. In this process, we first apply columnwise neighborhood search to the integer program to find a local optimum. We then apply columnwise neighborhood search to the linear programming relaxation with the hope that this will add promising new columns to the set of columns, and
thus let us escape the integer local optimum. Once we stop generating new columns using the linear program, we switch back to solving the integer program, which hopefully uses the new columns to find a better integer solution. This process repeats until no improvements have been made in either the integer program or linear program solutions, at which point we have a locally optimal solution.

Algorithm 1. High level overview of our solution method

```plaintext
while Time < AllowedTechnicianInitializationTime do
    TechnicianTours = Create initial technician tours using GRASP
end while

while Time < AllowedTechnicianSolveTime do
    SelectedTours = Solve MIP to select subset of TechnicianTours
    if SelectedTours has not changed since last iteration then
        break
    else if SelectedTours for each technician can be feasibly scheduled to satisfy days on/off rules then
        TechnicianTours += Neighbors of SelectedTours
    else
        Repair SelectedTours
        break
    end if
end while

TruckRoutes = Create initial truck routes, being each request in its own route
SelectedRoutes = DoTruckColNbhdSearch (TruckRoutes, SolverMode = SolveMIP)

while Time < AllowedRunTime do
    DoTruckColNbhdSearch (TruckRoutes, SolverMode = SolveLP)
    SelectedRoutes = DoTruckColNbhdSearch (TruckRoutes, SolverMode = SolveMIP)
    if SelectedRoutes has not changed since last iteration then break
end while

Assign each route in SelectedRoutes to the earliest feasible day for its machine requests
while Time < AllowedRunTime do
    Run Fusion Method on SelectedRoutes and SelectedTours
end while

sub DoTruckColNbhdSearch (TruckRoutes, SolverMode)
while Time < AllowedRunTime do
    SelectedRoutes = Solve MIP or Solve LP (as per SolverMode) to select subset of TruckRoutes
end while
```

FIGURE 1 The essence of columnwise neighborhood search
We next continue by describing how we embedded columnwise neighborhood search in our solution method. First, to get an initial idea of the structure of our method, see Algorithm 1. This is a high level overview; details on the individual elements mentioned in Algorithm 1 are found below.

### 3.3 Technician solution

To prepare our columnwise neighborhood search, we first construct an initial set of feasible tours \( T^p \) for each person \( p \in P \). These, together, will form the set \( P^1 \), that is, the columns of the constraint matrix in the first iteration of our columnwise neighborhood search method. We chose to let \( P^1 \) consist of the tours used in several greedy-constructed feasible solutions to the technician subproblem. This ensures that we can always find a feasible solution \( x^1 \) that covers all requests. We point out that it is also possible to supplement \( P^1 \) with the tours from partial solutions to the technician VRP, but this is not what we implemented.

To find these initial solutions for the technician subproblem, we used a greedy randomized adaptive search procedure (GRASP). Using a GRASP procedure has three advantages: solutions are (a) fast to obtain, (b) of reasonable quality, and (c) plentiful. The details on the procedures that we implemented to find these initial solutions can be found in Appendix A.

Now that we have constructed the set \( P^1 \), we need to define a column’s cost. As in Section 2.1, we define the costs \( c^p_t \) to be the sum of the daily cost for using a person and the distance-based cost of tour \( t \) for person \( p \). We emphasize that this ignores the workforce size cost \( c_{\text{person}} \), a choice we made since we expected this aspect to be of minor importance.

We now are ready to formulate our generalized set partitioning problem. Note that since we decomposed the technician routing and scheduling, the MIP for this subproblem ignores the scheduling of tours to days. In that sense it solves a relaxation of the full MIP formulation in Section 2.1, and this significantly reduces the number of variables. For example, it reduces the variables \( y_{t}^{p} \) (whether tour \( t \) is done by person \( p \) on day \( d \) ), to \( y_{t}^{p} \) (whether tour \( t \) is done by person \( p \) ). Moreover, the model no longer has to keep track of the scheduling variables \( z_{t}^{p} \). We define our MIP for the technician (or person) subproblem as follows.

\[
\text{Person SPP :} \quad \min \sum_{p \in P} \sum_{t \in T^p} c^p_t y^p_t 
\]

s.t.
\[
\sum_{p \in P} \sum_{t \in T} b_{t,m} y^p_t = 1 \quad \forall m \in M, \tag{26}
\]
\[
\sum_{t \in T^p} y^p_t \leq \lfloor \theta l \rfloor \quad \forall p \in P, \tag{27}
\]
\[
y^p_t \in \{0, 1\} \quad \forall p \in P, \forall t \in T. \tag{28}
\]

Although this MIP does not schedule the tours, it will select a subset of tours that is likely to fit in a person’s schedule later. This is modeled by introducing constraints (27) that ensure that the number of days worked by each person is not too high. These restrictions are defined as a fraction \( \theta \) of the length of the time horizon \( l \) (in days). The floor function in (27) was added since it would not affect the outcome of the MIP, but should make it easier to solve. The question remains how to choose a value for this parameter \( \theta \), which we discuss in Appendix B. Solving the MIP above with the columns \( P^1 \) corresponds to solving GSPP(\( P^1 \)), and leads to our first solution \( x^1 \). The next step is to define a columnwise neighborhood rule \( N(\cdot) \).

In our implementation, we defined \( N(t) \) to be all feasible tours that can be constructed from tour \( t \) by adding or removing one request. We also experimented with a “swap” neighborhood (remove one request from the tour while simultaneously adding another), but found that they did not really help in addition to the neighborhoods described above.

We continue by iteratively solving GSPP(\( P^1 \)) and using \( N(\cdot) \) to add more columns to construct \( P^k+1 \). This procedure terminates when the columnwise neighborhood search has converged, or when the time limit is reached. When running according to competition rules, the time limit was always reached first.

The method described above was implemented for the solver challenge, albeit with one small change. Note that Equation (26) ensures that the selected routes cover each request exactly once. After experimenting with this formulation, we eventually decided to allow a request to be covered by more than one tour. If this occurred in the MIP solution, we greedily removed the over-covered requests later. Since the distance metric respects the triangle inequality, a removal of a request from a tour can only decrease the tour’s cost. We note that with this change, our implementation of the technician solution is actually a generalized set covering problem.
Now that we have found good tours for technicians, what remains to be done is assigning tours to days. This was done by a simple heuristic that is described next. For each tour, we determine the earliest day that it can be done. We then sort tours starting with the largest “earliest day” and schedule them as late as possible while adhering to a 4 days on, 1 day off person scheduling rule (see also Appendix B). We break ties by looking at a second sorting criterion: the number of requests in the tour—we prefer to schedule the larger tours later. We designed this heuristic in the hope of scheduling the installation of requests after their delivery window, in order to give maximum flexibility to the truck problem. Note that this is not always possible; in that case the truck problem is further restricted by the now fixed installation day.

This concludes our approach for the technician subproblem. We allow 50% of the available computation time for the method above. Of that time, 10% is allocated to constructing the initial set of columns \( \tilde{P} \). After the technician sub-problem is solved, the remaining 50% of the time is used to solve the truck problem and connect the subproblems, as described in the next two subsections. (Note that we experimented with numbers other than 50% but did not find significant improvements.)

### 3.4 Truck solution

We next solve the truck subproblem given the previously found technician solution. First, we need to deduce new due dates for truck deliveries: it is possible an installation was planned inside the original truck time window for that request. In that case we decrease the request’s due date \( \hat{d}_m \) accordingly, that is, we set \( \hat{d}_m \) to be 1 day earlier than the planned installation day.

We are now ready to model a set partitioning problem for trucks, and prepare another columnwise neighborhood search. While the details differ from the solution for technicians described above, the main ideas are the same.

First, let us look at the initial set of routes, \( P^1 \). As in the technician subproblem, we attempted to construct \( P^1 \) by gathering the routes of several greedily constructed solutions. We experimented with several options, including a parallel version of the savings algorithm [4]. Eventually, we decided to let \( P^1 \) be a small and easy to construct set: each request in its own route. Our experiments showed that although this initial set is rather limited, it did not significantly reduce the quality of the solutions found when columnwise neighborhood converges. In fact, the quality was very similar to what we could obtain with more complex definitions of \( P^1 \).

As with the route cost defined in Section 2.1, the costs associated with each route \( r \in R \) are denoted \( c_r \) and consist of the truck’s daily cost and the route’s distance cost. However, we now add a third component to \( c_r \), which is the idle cost for all requests in this route (computed under the assumption that \( r \) is scheduled as late as possible, that is, on day \( \hat{d}_r \)). In this definition of \( c_r \), it should be noted that the cost related to fleet size \( c_{\text{truck}} \) is not reflected here, but instead this is considered in the final fusion heuristic. We are now ready to formulate our generalized set partitioning problem.

Given a set of possible truck routes \( R \) to deliver machine requests \( M \), we model a set partitioning problem named “Truck SPP” that selects a subset of \( R \) that covers each machine request exactly once at minimum cost. The set \( R \) consists of feasible routes that can be completed by a truck during 1 day, that is, they satisfy capacity and distance constraints. In this model, the decision variables for each route \( r \in R \) are:

\[
x_r = \begin{cases} 
1 & \text{if route } r \text{ is chosen} \\
0 & \text{otherwise.}
\end{cases}
\]

The set partitioning problem for trucks is then:

**Truck SPP**:

\[
\min \sum_{r \in R} c_r x_r \\
\text{s.t. } \sum_{r \in R} a_{r,m} x_r = 1 \quad \forall m \in M, \\
x_r \in \{0, 1\} \quad \forall r \in R,
\]

where parameters \( a_{r,m} \) are defined as in Section 2.1. In contrast to Section 2.1, this MIP does not decide on which day the routes are execute (it will be on day \( \hat{d}_r \), an assumption also used when calculating the costs \( c_r \)).

In our implementation a feasible solution of the Truck SPP always exists, since \( P^2 \) always includes \( P^1 \): the routes that consist of only one machine request. Note that solving the Truck SPP identifies a subset of routes that covers all machine requests at minimum cost, but it does not schedule these on particular days. Since we ensured that all routes in \( R \) are feasible to perform by one truck in 1 day, and the number of trucks is not limited, this implies that any solution of the Truck SPP can always be feasibly assigned to days.

After solving the Truck SPP using columns \( P^1 \) and obtaining \( x^1 \), we use a neighborhood rule \( N(\cdot) \) to generate new columns. In our implementation, we defined neighbors to be either existing routes with one request added to them, or combining two routes into one while stopping at the depot in the middle. The latter turned out to be a useful addition since the truck capacity is often restrictive for the problem at hand.
TABLE 1 Performance of all teams in the finals

| Final rank | Mean rank |
|------------|-----------|
| 1          | 1.24      |
| 2          | 2         |
| 3          | 4.56      |
| 4          | 4.8       |
| 5          | 5.32      |
| 6          | 5.88      |
| 7          | 5.96      |
| 8          | 6.24      |

Note: The authors of this paper are the members of the team with final rank 3 and mean rank of 4.56.

It turns out that for all instances provided, this series of integer programs converges while there is still time left. At that point we solve the LP relaxation of the Truck SPP, as described in Section 3.2. In our numerical experiments, adding these LP relaxations resulted in up to 12% improvement compared to using IPs only (although there were also many problem instances for which LP relaxations lead to no further improvement).

Our columnwise neighborhood search quickly solves each subproblem (truck or technician) MIP until a local optimum for that subproblem is reached, that is, until adding columns does not improve the solution further. During each iteration, that is, for each fixed set of columns, the corresponding column-restricted MIP is solved exactly using Gurobi.

The methods above only optimize the decomposed problem, that is, they separately optimize the two subproblems. The remainder of the time is spent on a heuristic that constructs and improves the solution of the overall problem, which we describe next.

3.5 Fusion method

Our algorithm finishes with a “fusion method” that brings the solutions to the subproblems together and makes local changes to improve overall cost. It starts by combining the technician and truck solutions found thus far. As the truck problem was solved conditional on the technicians’ solution, this always results in a feasible initial solution to the overall problem.

Next, we run a best improvement local search: this improves the objective value of the current solution by iterating over all technicians, attempting to swap any two tours for a single technician between all pairs of days. This includes the case where one of the tours is empty, that is, a day off. Obviously, tours can be swapped only if allowed by the scheduling rules. Swapping the tours may also require a change in the days the corresponding truck routes are planned (if some machines need to be available earlier). Moreover, if this swap is executed it may be possible to simultaneously move some truck routes later, which can be beneficial because delivering machines later reduces idle costs. Both of these options, planning truck routes earlier and later, are taken into account when we determine the expected cost change for a potential technician tour swap.

After computing the overall cost improvement for all possible tour swaps we greedily select the swap that yields the biggest improvement across all technicians and pairs of days. We continue until no further improvement can be found or the time limit is reached.

We emphasize that this method has its limitations. First of all, it is unable to swap routes between different technicians. Moreover, it lacks the option to break up existing routes and tours. This was not a tactical decision, but rather a result of limited implementation time in the challenge. Such an elaboration of our fusion method could potentially be a valuable addition to our algorithm.

We continue by describing how the method above performed on the problem instances provided in the competition.

4 RESULTS

Our solution approach led to the third prize in the solver challenge. The finals consisted of eight teams, whose performance was ranked according to the rules described in [23] and is summarized in Table 1 (as published in [22]).

For an overview of the performance of our method per instance, see Table 2. This table shows a range of performance metrics, such as the number of iterations reached within the prescribed solve time and the number of columns in the last iteration. Moreover, we included the percentage of time spent using Gurobi to solve MIPs: this turns out to be 45% on average.
TABLE 2  Algorithm metrics for all hidden instances

| Instance (hidden) | Above best (%) | Total cost | Technician | Truck |
|-------------------|----------------|------------|------------|-------|
|                   |                |            | Iterations | Columns | Iterations | Columns | Time  | Gurobi (%) |
| 1                 | 2              | 68 041 665 | 7          | 59 313  | 30         | 9159    | 52    |
| 2                 | 12             | 992 008 070| 8          | 202 890  | 7          | 22 963  | 69    |
| 3                 | 1              | 1 362 532 465 | 7       | 287 945  | 44         | 29 088  | 42    |
| 4                 | 146            | 11 810 095 | 6          | 236 322  | 49         | 37 489  | 31    |
| 5                 | 16             | 2 815 292 824 | 3        | 271 799  | 30         | 65 746  | 36    |
| 6                 | 6              | 38 105 890 | 0          | 2163     | 7          | 236 011 | 60    |
| 7                 | 8              | 109 007 550 | 23         | 91 453   | 17         | 3561    | 33    |
| 8                 | 1              | 735 151 440 | 13         | 189 173  | 33         | 16 369  | 41    |
| 9                 | 14             | 1 919 207 206 | 10       | 307 915  | 53         | 34 236  | 54    |
| 10                | 42             | 42 455 785 | 7          | 411 435  | 31         | 30 984  | 40    |
| 11                | 33             | 7 523 850 610 | 0       | 2370     | 3          | 303 177 | 48    |
| 12                | 2              | 3 000 293 925 | 0        | 4112     | 7          | 195 768 | 51    |
| 13                | 1              | 5 23 204 809 | 18         | 100 297  | 29         | 5780    | 41    |
| 14                | 1              | 1 385 328 790 | 7       | 76 678   | 39         | 9190    | 55    |
| 15                | 8              | 174 680 300 | 12         | 321 433  | 38         | 18 330  | 22    |
| 16                | 29             | 104 109 870 | 0          | 2127     | 7          | 143 998 | 56    |
| 17                | 0              | 27 405 305 955 | 2       | 172 568  | 27         | 131 519 | 61    |
| 18                | 24             | 60 955 300  | 4          | 624 703  | 4          | 123 294 | 53    |
| 19                | 0              | 4 386 452 670 | 23        | 101 886  | 25         | 2895    | 27    |
| 20                | 41             | 143 750 270 | 16         | 202 867  | 40         | 16 486  | 33    |
| 21                | 35             | 62 835 495 | 0          | 1861     | 30         | 58 151  | 76    |
| 22                | 38             | 9 322 625  | 5          | 397 311  | 51         | 32 227  | 24    |
| 23                | 0              | 22 362 004 090 | 5       | 466 341  | 69         | 76 567  | 49    |
| 24                | 2              | 31 441 397 305 | 2       | 209 303  | 55         | 88 742  | 47    |
| 25                | 10             | 611 535 420 | 26         | 89 248   | 22         | 2076    | 28    |

Note: The three leftmost columns show competition performance metrics, provided by the challenge organizers. These results are aggregated over multiple runs with different seeds. The remaining columns show representative results that we recomputed after the challenge, as it was not necessary to calculate/output these metrics during the challenge. These metrics were obtained using a single representative algorithm run, using the solve time and hardware restrictions as per competition rules.

Note that there are a few instances that show 0 iterations for the technician subproblem. In all of these cases, the columns selected in the first technician MIP could not be feasibly assigned to days. In this case our method uses a simple greedy heuristic to repair the solutions so that a feasible schedule could be found, as discussed in Appendix A. Although it is possible to continue with column-wise neighborhood search upon finding a feasible technician solution, this is not what we implemented. Instead, after a successful repair we immediately move to the truck subproblem, hence the observation of 0 technician iterations.

4.1 Comparison to other participants

After the winners were announced, the organizers published the results of all finalists on the hidden instances [12]. This allowed us to compare our results to the other teams in more detail. We discovered that our method performs very well on some instances, but not all. In fact, given that we ended up in third place, it is somewhat remarkable how variable our performance was across instances. We next present our worst and best performances, and consider potential sources of variability in our performance.

4.2 Best and worst instances

Our best performance was on instance “ORTEC Hidden 23,” hereafter simply referred to as Hidden 23. For this instance, our total cost was only 0.08% above that of the best team for that instance. This contrasts strongly with our worst performance, occurring for instance Hidden 04. Not only were we eighth (last) in the ranking, the differences were also very large: a factor 2.45 from the best solution (i.e., a cost difference of 145% between our solution and the best) for that instance.

As we see in Figure 2, our best solution (left bar chart), corresponding to instance Hidden 23, consists of 99.9% of truck distance cost. In contrast, our worst solution (right bar chart), corresponding to instance Hidden 04, has many more components...
that significantly contribute to its total cost. In particular, truck distance is no longer the dominant cost, and idle costs and truck
day costs (costs associated with using a truck for a day) are much more important than in our solution to Hidden 23.

This led us to believe that the key determinants of how well our method performs include (a) the fraction of our overall
cost that is made up of truck distance cost, and (b) the performance of our fusion method for combining solutions and reducing
interaction effects. This is also consistent with how our method is designed: while it first solves the technician routing problem
MIP, it then uses a heuristic technician scheduling method that attempts to account for downstream effects on the truck problem
by scheduling installations as late as possible. This provides maximum flexibility for solving the subsequent truck problem MIP.

Based just on the above comparison of best and worst cases, it appears that this scheduling heuristic indeed allows sufficient
flexibility for the truck problem to be solved well: when truck distance is important, our method performs well. We expect,
however, that it may be doing worse at optimizing the other factors, for example, by scheduling the technicians too poorly
initially, and/or not being able to compensate for this during the final fusing of solutions. The role of these other factors is
expected to be most apparent for those instances in which truck cost is not the dominant factor.

Next we consider exploratory results using all hidden instances to (a) confirm the impact of truck distance cost on perfor-
mance as seen in the best and worst cases, and (b) consider the effects of factors beside truck distance cost on our performance
across instances.

4.3 Variability in performance

4.3.1 Performance by truck distance cost

We measured our performance on all hidden instances, relative to the performance of the best team for each instance. This is
shown in Figure 3, which plots our relative overall performance against the percentage of our overall cost made up by truck
distance cost.

We see that our solutions fall into three categories in terms of the percentage of the total cost made up by truck distance cost.
These categories correspond, roughly, to the ranges: 0-10%, 40-50% and 90-100% of total cost due to truck distance cost. Aside
from the one outlier in the 0-10% group for which we do particularly poorly, there is no obvious difference in the performance
between the 0-10% and 40-50% truck distance cost categories, especially as compared to the 90-100% category. Hence the
0-10%, 40-50% categories were merged into a category of “<50%” in what follows. The 90-100% category was labeled as
“≥50%” for simplicity. These may be interpreted more generally as categories of “truck distance cost does not dominate” and
“truck distance cost dominates.”

4.3.2 Performance by fusion heuristic effect

Figure 4 shows our overall performance on the hidden instances, relative to the best team, plotted against the percentage reduc-
tion achieved by our fusion heuristic. We have further labeled the data points by “<50%” and “≥50%” groupings (i.e., “truck
distance cost does not dominate” and “truck distance cost dominates” categories, as discussed above).

As can be seen in the figure, our relative performance actually gets worse—our percentage cost above the best increases—as
the fusion heuristic has more effect. One interpretation of this is that when improvement is possible using our simple fusion
heuristic, there exist even better strategies for improving the overall solution that we are not using.
Figure 5 is a close up view of Figure 4 above, restricted to those instances for which the fusion heuristic has less than a 1% effect on the overall solution quality. Our performance over this range is relatively constant within truck distance cost groups, and has large differences between truck distance cost groups. We again see that we perform well when truck distance costs are the dominant factor, and perform relatively badly when truck distance cost is not the dominant factor.

Since our fusion heuristic is designed to compensate for the costs associated with an overall decomposition into technicians and trucks, the above analysis indicates that we are not achieving a well-balanced split. Combined with the fact that our algorithm performs well on problems for which the truck distance cost is dominant, this likely indicates that our initial technician solution is quite far from optimal and/or that our fusion heuristic cannot compensate for this.

As discussed above, the source of the inefficiency in technician solutions is most likely in the technician scheduling heuristic and the inability of the fusion heuristic to fix this.

Next we consider the performance of our MIP solution strategies.

4.4 Columnwise neighborhood search: Performance over time

In this section we investigate how the solution quality of our columnwise neighborhood search method improves over time. Since the two parts of our solution, that is, the truck and technician MIP, do not interact with one another, we present the performance on these two subproblems separately.
Figure 5 shows performance relative to best against fusion heuristic effect, grouped by truck distance cost importance. Same as previous figure but restricted to fusion heuristic effect <1%. [Color figure can be viewed at wileyonlinelibrary.com]

Figure 6 shows how the performance of columnwise neighborhood search for technicians progresses over time. We experimented with five instances: ORTEC Early 01-05. We also investigated how this performance would improve if the challenge rules had prescribed a longer run time. To that end, we set the time limit to be five times longer, and plotted the results in the same figure. Recall that 10% of the time spent on the technician subproblem (i.e., 5% of the total run time) is set aside for constructing initial tours. Therefore, a longer run time results in a larger set \( P_1 \), and therefore also a better first solution \( x_1 \), as can be seen for all five problem instances in Figure 6. Although the quality of the first solution improves with larger run times, Figure 6 shows a trade-off in terms of how long it takes to find those initial solutions. For example, focus on instance Early 01: the objective value of the first solution is 5% better when running the algorithm five times longer. However, if one is interested in achieving a solution of such quality quickly, it is better to run the algorithm as per competition run time as the corresponding smaller set of initial tours shows a faster decrease in objective value. The results for competition run times in Figure 6 indicate that our proportion of time set aside for creating initial solutions was well-chosen, as those graphs appear to be near convergence by the end of the run time.

When solution quality is more important than run time, Figure 6 shows that it is a good idea to start with a larger initial set \( P_1 \). For all instances in Figure 6 this richer set of initial tours eventually leads to an improvement in the objective value. For
example, take instance Early 04 and observe that running five times longer leads to a solution to the technician subproblem with a cost that is approximately \( 0.64 \) \( \approx 85 \) percent of the cost one would get using competition run times. Finally, we note that all performances were normalized with respect to the objective value of the first solution \( x_1 \) obtained when using the competition run time.

The performance of columnwise neighborhood search for trucks is depicted in Figure 7, for instances ORTEC Early 01-05. These results were obtained with restricted run times as per competition rules. Note that longer run times will not help to improve the starting solution for trucks, since our implementation always start with the same set \( P_1 \) (each request in its own route). This figure shows that the first solution \( x_1 \) is always quickly obtained: this has to do with our choice of \( P_1 \), which implies there is only one feasible solution \( (x_1 = 1) \). In general, we notice that our method converges quicker for instances with fewer requests, which seems reasonable. However, this is not always the case, and to demonstrate this we also added instance Early 08 to Figure 7. Despite having only 300 requests, we observe our method required a remarkably long time to find the second solution \( x_2 \). We investigated this instance and observed that it has a very high truck day cost (100 000) relative to all other costs. To analyze if this is what caused the long solve time, we tested a version of this instance where we adapted the truck day cost to 1. The results are found in Figure 8. We reiterate that the truck day cost does not affect feasibility; it only affects the objective function as it is included in \( c_r \), the costs associated with each route \( r \in R \). From Figure 8 we conclude that the high truck day cost indeed was what caused the large MIP solve time for instance Early 08.

We next analyze the behavior of columnwise neighborhood search when it switches between IPs and LP relaxations. Figure 9 shows the performance on instance Hidden 23 over time, where we zoom in on the moment when the LP relaxations start occurring. Note that the objective value can go up instead of down over time. This can be explained by observing that the lower values are the result of solving an LP relaxation of the problem, whereas the higher versions are solutions of the corresponding integer program. Moreover, note that LP solutions are found quickly after one another, because those problems are easier to solve. For this particular instance, the LP relaxations contributed to eventually find a slightly better integer solution.

We have just demonstrated how the cost of our best found solution decreases over time. Meanwhile, the problem size of our MIP increases, which we discuss next.

4.5 | MIP size

This section reports on the size of our mixed integer programs. We measure this in terms of the number of columns per iteration, that is, the number of routes or tours that we select from. (Recall that for trucks this is equal to \( |R| \) and for technicians it is equal to \( \sum_{p \in P} T^p \).)
We show these column size measures for a particular instance; Early 02, a medium-sized instance in terms of requests and technicians, with 300 requests and 50 technicians. The results are shown in Table 3.

For the subproblem concerning technicians, recall that the number of columns in the first iteration depends on the run time. Our algorithm contains an initialization phase which amounts to 10% of the run time for the technician subproblem, so 5% of the total run time. When running instance Early 02 for the time allowed by the competition rules, this leads to 40 different initial solutions to the technician problem. These 40 solutions together contain 2900 tours, corresponding to 2900 columns in the constraint matrix in the first iteration in Table 3. For the truck subproblem, recall that the number of initial solutions does not depend on the run time. The 300 requests in instance Early 02 explain the 300 columns in the first iteration in Table 3.

The bottom row in Table 3 is the last iteration when running the algorithm for the time allowed in the competition. Throughout all problem instances, we saw that the MIP for trucks converged within that time, whereas the MIP for technicians reached a time-out.

5 | DISCUSSION

In our solution approach we adopted a heuristic decomposition to create technician and truck problems. Furthermore, we decomposed the technician problem into a routing problem and a scheduling problem. Each of these decompositions requires further heuristic strategies to attempt to reduce the costs associated with solving these subproblems separately. For the technician and truck problem division, we first solved the technician problem before the truck problem. In order to allow for downstream effects on the truck problem, our technician scheduling heuristic was designed to provide as much flexibility as possible to the truck problem by scheduling installations as late as possible.
TABLE 3 Mixed-integer programming size per iteration for a problem instance with 300 requests

| Iteration | Columns |
|-----------|---------|
| Trucks    |         |
| 1         | 300     |
| 2         | 17000   |
| 3         | 20000   |
| 4         | 21000   |
| ⋮         | ⋮       |
| 70        | 23500   |
| Technicians|       |
| 1         | 2900    |
| 2         | 20000   |
| 3         | 36000   |
| 4         | 53000   |
| ⋮         | ⋮       |
| 10        | 159000  |

In analyzing our performance results, it appears that our scheduling heuristic indeed allows sufficient flexibility for the truck problem to be solved well: when truck distance is important, we perform well. Our columnwise neighborhood search procedure appears to perform well on the MIPs, giving a good overall reduction in cost.

We acknowledge that our implementation probably was not able to demonstrate the full potential of the concept of columnwise neighborhood search. For example, note that how we defined our “neighborhood rules” is merely an example; there may be other, more clever ways to construct neighborhoods. For the truck problem, where columnwise neighborhood search quickly converges, one might have benefited from defining larger neighborhoods. Conversely, for the technician problem the columnwise neighborhood search does not converge within the prescribed run time, which means one could have benefited from smaller, more cleverly defined neighborhoods. This could have been done, for example, by preprocessing the data to identify “tricky installations” or by bundling requests that have the same location.

Despite the success of our columnwise neighborhood search procedure in solving the MIP problems, we see that we are potentially sacrificing too much performance on the technician scheduling problem in an effort to provide flexibility to the truck problem. When the truck distance cost is not dominant, we tend to perform relatively poorly. Our decomposition strategy is still potentially appealing, however. We find that we can solve the technician problem first in such a manner that it does not impact much on the downstream truck problem. The issue is whether we can subsequently improve the technician scheduling, and the associated idle and other costs, during our fusion method stage. Currently it appears that our fusion method is too limited.

These results highlight a key issue in applying matheuristics methods more generally: the heuristics used both to decompose the problem into components that can be solved “exactly” (or very well), and those used to recombine these solutions into an overall solution, can have a large impact on performance. If these heuristics are not carefully designed to balance overall performance, we expect to do well when one “well-solved” subproblem dominates, but may have very poor performance when this is not the case. This is reflected in our particularly variable performance across instances as compared to other teams.

Finally, we note that our approach was able to achieve third place in the challenge despite the programming language of our choice being Python. Although a significant fraction of computation time is spent in Gurobi (on average 45%, see Table 2), this still leaves 55% of run time in Python. Python is known to be slow, mainly because it is interpreted at run time, as opposed to the languages that the first and second prize winners used (C++ and Visual Basic). The latter are compiled to native code and are up to a factor 100 faster than Python when considering large matrix sizes (see [17] for a comparison of Python to C). This observation strengthens our belief that columnwise neighborhood search is a powerful method for solving various types of VRPs, in particular those for which a set partitioning approach is suitable.

6 | CONCLUSIONS

In this article we have presented our solution approach to the VeRoLog Solver Challenge 2019, organized by VeRoLog, the EURO Working Group on Vehicle Routing and Logistics Optimization. We adopted a matheuristics philosophy, heuristically decomposing the overall problem into parts that could be solved exactly, or near exactly, as MIPs. In order to solve the MIPs,
we introduced a novel method we call “columnwise neighborhood search.” Our methodology was considered successful, as our team was awarded third place in the challenge. Our performance across instances was notably variable compared to other teams, however. Further investigation into this variability indicated that our approach performs particularly well when the problem is dominated by truck distance cost, and less well on other instances. The key limitation in our approach appears to be the heuristics used for piecing component solutions back together into an overall solution, in particular our final “fusion” method. This method was designed to reduce inefficiencies due to the original decomposition, but it appears it was unable to provide any relative benefit. While better heuristics for recombining component solutions thus appear necessary, we conclude that our columnwise neighborhood search procedure showed promising performance in solving set partitioning formulations of VRPs.

ACKNOWLEDGMENTS

We thank the organizers of the VeRoLog Solver Challenge 2019 for a very enjoyable and well-organized challenge. This work was supported in part by the Netherlands Organisation for Scientific Research (NWO) in the form of a Rubicon grant (project number 019.172EN.016).

ORCID

Caroline J. Jagtenberg  https://orcid.org/0000-0002-0742-7707
Oliver J. Maclaren  https://orcid.org/0000-0002-3982-4539
Andrea Raith  https://orcid.org/0000-0002-0417-2972

REFERENCES

[1] C. Archetti and M.G. Speranza, A survey on matheuristics for routing problems, EURO J. Comput. Optim. 2 (2014), 223–246.
[2] M.O. Ball, Heuristics based on mathematical programming, Surv. Oper. Res. Manag. Sci. 16 (2011), 21–38.
[3] C. Barnhart, E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh, and P.H. Vance, Branch-and-price: Column generation for solving huge integer programs, Oper. Res. 46 (1998), 316–329.
[4] G. Clarke and J.W. Wright, Scheduling of vehicles from a central depot to a number of delivery points, Oper. Res. 12 (1964), 568–581.
[5] L.D. Cleland, A.J. Mason, and M.J. O’Sullivan, Using neighbourhood search to solve generalised staff rostering problems, Proceedings of the 51st Conference of the Operations Research Society of New Zealand, Operations Research Society of New Zealand, New Zealand, 2017, http://orsnz.org.nz/Repository/CONF51/ORSNZ17_Cleland.pdf.
[6] M. Crowder and A.J. Mason, Districiting for the New Zealand census:MIP-heuristic approaches, Proceedings of the 46th Conference of the Operations Research Society of New Zealand, Operations Research Society of New Zealand, New Zealand, 2012, pp. 88–96. http://orsnz.org.nz/conf46/content/ORSNZ12_conference_proceedings.pdf.
[7] G.B. Dantzig and J.H. Ramser, The truck dispatching problem, Manag. Sci. 6 (1959), 80–91.
[8] K.F. Doerner and V. Schimid, “Survey: Matheuristics for rich vehicle routing problems,” Hybrid Metaheuristics, Springer, Berlin, Heidelberg, 2010, pp. 206–221.
[9] M. Drexl, Rich vehicle routing in theory and practice, Logistics Res. 5 (2012), 47–63.
[10] M. Fischetti and M. Fischetti, “Matheuristics,” Handbook of Heuristics, R. Marti, P. Panos, and M.G.C. Resende (eds), Springer International Publishing, Cham, 2016, pp. 1–33.
[11] B.A. Foster and D.M. Ryan, An integer programming approach to the vehicle scheduling problem, Oper. Res. Q 27 (1976), 367–384. https://doi.org/10.1007/s41604-019-00011-8.
[12] Full-run-finalists-hidden-instances, 2019. https://verolog2019.ortec.com/raw/full-run-finalists-hidden-instances.csv (Accessed September 10, 2019).
[13] F. Glover and M. Laguna, “Tabu search,” Modern Heuristic Techniques for Combinatorial Problems, C.R. Reeves (ed.), John Wiley and Sons, New York, 1993, pp. 70–150.
[14] T.F. Gonzalez, Districting for the New Zealand census:MIP-heuristic approaches, Chapman & Hall, London, 2018.
[15] J. Gromicho, P. van’t Hof, and D. Vigo, The VeRoLog Solver Challenge, J. Vehicle Rout. Algor. 2 (2019), 109–111. https://doi.org/10.1007/s41604-019-00011-8.
[16] Gurobi. “Gurobi”, 2019. http://gurobi.com.
[17] IBM Community. A speed comparison of C, Julia, Python, Numba, and Cython on LU factorization, 2016. https://www.ibm.com/developerworks/community/blogs/jfp entry/A_Comparison_Of_C_Julia_Python_Numba_Cython_Scipy_and_BLAS_on_LU_Factorization (Accessed February 29, 2020).
[18] S. Kirkpatrick, C. J. Gelatt, and M. Vecchi. “Optimization by simulated annealing,” Technical Report, IBM Research Report RC 9355, 1982.
[19] R. Lahyani, M. Khemakhem, and F. Semet, Rich vehicle routing problems: From a taxonomy to a definition, Eur. J. Oper. Res. 241 (2015), 1–14.
[20] J. Puchinger and G.R. Raidl, “Combining metheuristics and exact algorithms in combinatorial optimization: A survey and classification,” Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach, Springer, Berlin, Heidelberg, 2005, pp. 41–53.
[21] P. Toth and D. Vigo, Vehicle Routing: Problems, Methods, and Applications. MOS-SIAM Series on Optimization, 2nd edn, SIAM, Philadelphia, 2014.
[22] VeRoLog Solver Challenge, 2019. http://verolog2019.ortec.com/ (Accessed July 9, 2019).
[23] VeRoLog Solver Challenge information for participants, 2019. http://verolog2019.ortec.com/pdf/VSC2019_info.pdf (Accessed June 25, 2019).
[24] J. Waugh and A.J. Mason, Columnwise search—A heuristic for the vehicle routing problem applied to waste collection, Auckland, Operations Research Society of New Zealand, 2014. https://secure.orsnz.org.nz/conf48/.
APPENDIX A: CONSTRUCTING INITIAL TECHNICIAN TOURS

In this section we give details of how we constructed the initial set of columns $P^1$ for the technician subproblem. This involved creating multiple feasible solutions to the technician VRP, using greedy and randomized methods that we describe next.

These methods start by sorting the requests. This is done either randomly, by the number of people that can service the request (in ascending order), by the start of their truck delivery window (in ascending order) or by the end of their truck delivery window (in ascending order). We then iterate over this sorted list of requests, assigning them to an available person. This person is chosen either randomly, or by the closest home address, or by the closest home address where distance is measured as a fraction of that person’s maximum daily travel distance, or by the number of requests that that person can service according to their skills: we can choose either the minimum (which is useful for problem instances that are restricted by the number of available people, because it uses the least skilled person whenever possible) or the maximum (which is useful for problem instances where a technician day is rather expensive, because it leads to assigning a lot of requests to the most skilled person). Lastly, we added a randomized method that selects a person with a probability that is proportional to the ratio of their distance from the request and their maximum daily travel distance.

When assigning a request to a person, we first try to merge it with an existing tour for that person. If that is somehow infeasible, we create a new tour. Whenever a tour is changed, we check if it is still feasible to schedule the set of tours for that person. For most instances this approach gave us a multitude of tours which formed the initial set $P^1$.

APPENDIX B: HANDLING OF TECHNICIAN SCHEDULING RULES

In this section we elaborate on how our solution for technician routing takes into account the scheduling (labor) rules. Recall that the labor rule specifies that a person needs 2 days off after having worked for five consecutive days, and only 1 day off after having worked 4 days. This latter pattern has a more favorable ratio of days worked to days off, thereby offering more flexibility to the set covering model, which likely leads to a solution with lower cost. Therefore, it seems reasonable to constrain the number of days a technician works to at most 80% of the length of the time horizon, that is, to choose $\theta = 0.8$ in Equation (27). We experimented with this value and, while this often lead to a technician solution that could feasibly be scheduled to days, using $\theta = 1$ seemed to work just as well. This can be explained by observing that technicians can work for slightly more than 80% of the time, for two reasons: (a) by following a 4 days on, 1 day off pattern and finishing with five consecutive work days at the end of the scheduling horizon and (b) if the scheduling horizon is not a multiple of 5 days. Another explanation might be that, for many of the problem instances, in a near-optimal solution the workload naturally tends to be spread among technicians due to them having different home bases. In our implementation as submitted to the challenge finals, the value $\theta = 1$ was used.

In general, it should be noted that regardless of the value of $\theta$, this approach does not guarantee that a solution of (25) can be scheduled to days. After each iteration of columnwise neighborhood search, we checked whether a feasible schedule can be found for the selected tours. If no feasible schedule could be found, we stopped the columnwise neighborhood search and started repairing the solution. Repair was done by selecting those technicians that are overloaded, and moving their requests to different technicians. We made such moves greedily in terms of the resulting change in objective value. These local moves were done either by keeping the whole tour intact, or by breaking break up existing tours and redistributing the requests among other technicians.