Integrating Column Generation and Large Neighborhood Search for Bus Driver Scheduling with Complex Break Constraints

LUCAS KLETZANDER*, Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, Databases and Artificial Intelligence Group, TU Wien, Austria
TOMMASO MANNELLI MAZZOLI, Databases and Artificial Intelligence Group, TU Wien, Austria
NYSRET MUSLIU, Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, Databases and Artificial Intelligence Group, TU Wien, Austria
PASCAL VAN HENTENRYCK, Milton Stewart School of Industrial and Systems Engineering, Georgia Institute of Technology, USA

Background: The Bus Driver Scheduling Problem (BDSP) is a combinatorial optimization problem with the goal to design shifts to cover prearranged bus tours. The objective takes into account the operational cost as well as the satisfaction of drivers. This problem is heavily constrained due to strict legal rules and collective agreements. **Objectives:** The objective of this article is to provide state-of-the-art exact and hybrid solution methods that can provide high-quality solutions for instances of different sizes. **Methods:** This work presents a comprehensive study of both an exact method, Branch and Price (B&P), as well as a Large Neighborhood Search (LNS) framework which uses B&P or Column Generation (CG) for the repair phase to solve the BDSP. It further proposes and evaluates a novel deeper integration of B&P and LNS, storing the generated columns from the LNS subproblems and reusing them for other subproblems, or to find better global solutions. **Results:** The article presents a detailed analysis of several components of the solution methods and their impact, including general improvements for the B&P subproblem, which is a high-dimensional Resource Constrained Shortest Path Problem (RCSPP), and the components of the LNS. The evaluation shows that our approach provides new state-of-the-art results for instances of all sizes, including exact solutions for small instances, and low gaps to a known lower bound for mid-sized instances. **Conclusions:** We observe that B&P provides the best results for small instances, while the tight integration of LNS and CG can provide high-quality solutions for larger instances, further improving over LNS which just uses CG as a black box. The proposed methods are general and can also be applied to other rule sets and related optimization problems.

JAIR Associate Editor: Ken Brown

JAIR Reference Format:
Lucas Kletzander, Tommaso Mannelli Mazzoli, Nysret Musliu, and Pascal Van Hentenryck. 2026. Integrating Column Generation and Large Neighborhood Search for Bus Driver Scheduling with Complex Break Constraints. *Journal of Artificial Intelligence Research* 85, Article 44 (April 2026), 37 pages. DOI: 10.1613/jair.1.19027

*Corresponding Author.

Authors' Contact Information: Lucas Kletzander, lucas.kletzander@tuwien.ac.at, ORCID: 0000-0002-2100-7733, Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, Databases and Artificial Intelligence Group, TU Wien, Vienna, Austria; Tommaso Mannelli Mazzoli, tommaso.mazzoli@tuwien.ac.at, ORCID: 0000-0002-5861-3155, Databases and Artificial Intelligence Group, TU Wien, Vienna, Austria; Nysret Musliu, nysret.musliu@tuwien.ac.at, ORCID: 0000-0002-3992-8637, Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, Databases and Artificial Intelligence Group, TU Wien, Vienna, Austria; Pascal Van Hentenryck, pascal.vanhentenryck@isye.gatech.edu, ORCID: 0000-0001-7085-9994, Milton Stewart School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia, USA.

DOI: 10.1613/jair.1.19027

1 Introduction

Scheduling employees for public transport is an area that needs to respect a variety of complex constraints, time schedules, spatial requirements, and conflicting objectives, giving rise to difficult optimization problems. Drivers bear great responsibility for their passengers, leading to a range of legal requirements, collective agreements and company policies that demand particular break assignments and shift characteristics. Companies require cost-efficient schedules, while drivers and labor unions request employee-friendly schedules to reduce stress and increase compatibility of the required shift work with the private life of the drivers.

This paper is a significantly extended version of our previous conference papers presented at AAAI (Kletzander, Musliu, and Van Hentenryck 2021) and ICAPS (Mazzoli et al. 2024). It presents exact and hybrid methods for a complex Bus Driver Scheduling Problem (BDSP). The constraints of the BDSP depend on the country's legal regulation, and in our case, they follow the Austrian *collective agreement for employees in private omnibus providers* (WKO 2019) using the rules for regional lines. In particular, the collective agreement has stringent rules requiring drivers to take frequent breaks, with the eventual option of splitting them into multiple parts, and the objectives in this article combine cost optimization with several employee satisfaction criteria, such as reducing long unpaid breaks or frequent vehicle changes. While the rules are specific to the application, the concepts of the solution approach are generally applicable to other rule sets and problems with similar structure.

As an exact approach, we propose Branch and Price (B&P). This method works on a branching tree, and performs Column Generation (CG) at each node, where the problem is split into Set Partitioning as the master problem and the Resource Constrained Shortest Path Problem (RCSPP) as the subproblem. However, due to the complex set of constraints, the labels used in solving the subproblem have a high number of dimensions, leading to a slow determination of non-dominated labels when solving the subproblem. Therefore, several options to improve the generation of new columns in the presence of a high number of dimensions are investigated, ultimately leading to the use of k-d trees and bounding boxes in a two-stage dominance algorithm. This approach was originally published in a conference paper at AAAI (Kletzander, Musliu, and Van Hentenryck 2021), and is extended with more detailed and formal descriptions of the novel improvements to solve the high-dimensional sub-problem as well as new experiments.

Although B&P gives very good results for small- to medium-sized instances, exact methods are computationally too expensive for large instances, and heuristic methods cannot obtain optimal solutions. Therefore, the study of new methods to tackle larger instances is of particular interest. In this paper, we present a novel approach based on the Large Neighborhood Search (LNS) framework, which has been successfully used for solving other challenging real-life problems. However, applying this method to the BDSP requires innovative ideas for the destroy and repair operators, as well as a detailed investigation into their impact and the choice of parameters. In particular, B&P (or the CG part of B&P) is used as the repair operator, as it provides high-quality solutions for small sub-problems very quickly. This approach was originally published in a conference paper at ICAPS (Mazzoli et al. 2024), and is extended with more details on the design of LNS and an extended set of new experiments.

However, it is possible to see B&P and LNS not only as separate components, but to enforce a tighter integration between the two. Therefore, this article also includes a substantial novel extension to the previous conference papers investigating how to boost performance even further by allowing exchange between the different sub-problems of LNS, in particular, by storing columns from sub-problems for use in later, different sub-problems, and by using the combined set of columns that are aggregated from different sub-problems to find better solutions even more quickly. This leads to significant improvements in the results, establishing this hybrid as the new state-of-the-art method for this version of the BDSP.

The main contributions of this paper are:

- A Branch and Price (B&P) approach provides an exact solution method that can solve small instances optimally, and provide high-quality solutions for medium size.

- Several improvements in solving the Resource Constrained Shortest Path Problem (RCSPP) in general are proposed and evaluated, which allow solving high-dimensional RCSPPs much faster.
- A novel Large Neighborhood Search (LNS) method is proposed , including innovative destroy operators, using the CG part of our B&P as a repair.
- A detailed analysis of the effect of the different destroy and repair components of LNS is provided.
- We propose a novel tight integration of LNS and CG, where columns from each sub-problem are aggregated and reused to significantly improve the search.
- A detailed evaluation is provided on the complex Austrian BDSP using a public benchmark instance set, where our approach clearly constitutes the new state of the art, outperforming previous results across all sizes of instances. While this evaluation is specific to this version of the problem, the ideas used in B&P and LNS and their integration are generally applicable for complex personnel scheduling problems.

The remainder of this article is structured as follows. Section 2 provides related work to the problem and methods, Section 3 introduces the formal definition of the problem, Section 4 contains the details about Branch and Price, Section 5 explains the details about Large Neighborhood Search, Section 6 provides the tight integration of the two approaches, Section 7 contains the experiments for the different approaches and their detailed analysis, and finally Section 8 provides conclusions and an outlook to future work.

2 Related Work

There are many versions of employee scheduling problems and several surveys (Ernst et al. 2004; Van den Bergh et al. 2013) provide a good overview of this line of work. Driver scheduling, as a part of crew scheduling, resides between vehicle scheduling and driver rostering in the process of operating bus transport systems (Ibarra-Rojas et al. 2015).

Research on bus driver scheduling problems has a long history (Wren and Rousseau 1995) and uses a variety of solution methods. Exact methods mostly use column generation with a set covering or set partitioning master problem and a resource constrained shortest path subproblem (Desrochers and Soumis 1989; Lin and Hsu 2016; Portugal et al. 2009; Smith and Wren 1988). Exhaustive search has also been investigated (S. Chen et al. 2013).

Heuristic methods like greedy (De Leone et al. 2011; Martello and Toth 1986; Tóth and Krész 2013), or tabu search (Lourenço et al. 2001; Shen and R. S. K. Kwan 2001), genetic algorithms (Li and R. S. Kwan 2003; Lourenço et al. 2001), or iterated assignment problems (Constantino et al. 2017) are used in different variations.

However, most work so far focuses mainly on cost, rarely minimizing idle time and vehicle changes (Constantino et al. 2017; Ibarra-Rojas et al. 2015). Break constraints are mostly simple, often including just one meal break. Complex break scheduling within shifts has been considered by authors in different contexts (Beer, Gaertner, et al. 2008; Beer, Gärtner, et al. 2010; Widl and Musliu 2014). There is not much work on multi-objective bus driver scheduling (Lourenço et al. 2001), but multi-objective approaches are used in other bus operation problems (Respício et al. 2013). Recent work includes automated weight setting (Kletzander and Musliu 2023a) and decision support for human planners (Frohner et al. 2024).

This article investigates a complex real-life Bus Driver Scheduling Problem that was first introduced by (Kletzander and Musliu 2020). This work explains the need for a combined objective function that goes beyond cost. The published instances were solved with simulated annealing and a hill climbing heuristic. Good results have also been provided using tabu search (Kletzander, Mazzoli, et al. 2022) and hyper-heuristics (Kletzander, Mazzoli, et al. 2022; Kletzander and Musliu 2023b), while the state-of-the-art heuristic solutions beside the methods in this article are achived using Construct, Solve, Merge, and Adapt (Rosati et al. 2023).

Branch and Price is a decomposition technique for large mixed integer programs (Barnhart et al. 1998). At each node in a branching tree, a master problem works on a set of columns, while a sub-problem is responsible for generating new columns with negative reduced cost, i.e., the potential to improve the solution to the master

problem. This work uses set partitioning (Balas and Padberg 1976) as the master problem and the RCSPP (Irnich and Desaulniers 2005) as the subproblem. Resources are modeled via resource extension functions (REF) (Irnich 2008). Different techniques for dealing with pareto-front calculation can mostly be found as maxima-finding algorithms in a geometrical context (Bentley 1980; W.-M. Chen et al. 2012). Large neighborhood search (Shaw 1998) is an iterative solution method, where in each iteration a part of the solution is destroyed and rebuilt by dedicated destroy and repair operators.

3 Problem Description

The investigated Bus Driver Scheduling Problem (BDSP) deals with the assignment of bus drivers to vehicles that already have a predetermined route for one day of operation. The problem specification was introduced by (Kletzander and Musliu 2020). The notation is summarized in Table 2 at the end of the section.

3.1 Problem Input

The input of the BDSP consists of three pieces of data:

- **Positions and Distance Matrix:** A finite set $P \subseteq \mathbb{N}$ of *positions*. A time distance matrix $D = (d_{pq}) \in \mathbb{R}_{\geq 0}^{(P \times P)}$ where d_{pq} represents the time needed for a driver to go from position p to q when not actively driving a bus. If no transfer is possible, then we set d_{pq} to a big constant M. If $p \neq q$, then d_{pq} is called *passive ride time*. The value d_{pp} represents the time it takes to switch tour at the same position, but is not considered passive ride time.
- **Start and End Work**: Shifts can start and end at any position. For each position $p \in P$, the two values $startWork_p$ and $endWork_p$ represent, respectively, the time required to start or end a shift at that position. This is usually used at the depot as time to prepare the parked bus for use, or shut down and check the bus at the end of the day. Travel to the first position or from the last position is not part of the shift, and therefore not considered in this formulation.
- **Bus Legs**: A set L of *bus legs*, where each leg $\ell \in L$ is a 5-tuple:

  $$\ell = (tour_\ell, startPos_\ell, endPos_\ell, start_\ell, end_\ell),$$

  representing the trip of a vehicle from a start position to an end position over a specified time interval:
  - $tour_\ell \in \mathbb{N}$ is the ID of the vehicle
  - $startPos_\ell, endPos_\ell \in P$ are respectively the starting and the ending positions of the leg
  - $start_\ell \in \mathbb{R}$ is the time at which the vehicle departs from position $startPos_\ell$
  - $end_\ell \in \mathbb{R}$ is the time at which the vehicle arrives at position $endPos_\ell$

  Legs with the same tour t do not overlap, which means that the intervals $(start_\ell, end_\ell)$ for ℓ with $tour_\ell = t$ are disjoint.

Table 1. A Bus Tour Example

ℓ	$tour_\ell$	$start_\ell$	end_ℓ	$startPos_\ell$	$endPos_\ell$
1	1	400	495	0	1
2	1	510	555	1	2
3	1	560	602	2	1
4	1	608	640	1	0

Note that L is totally ordered by *start*, using *tour* as tie-breaker. Let $\preceq$ be the relation on L defined as follows: For any $\ell_1, \ell_2 \in L$, write $\ell_1 \preceq \ell_2$ if

- $start_{\ell_1} < start_{\ell_2}$, or
- $start_{\ell_1} = start_{\ell_2}$ and $tour_{\ell_1} \leq tour_{\ell_2}$

Lemma 1. *Then $\preceq$ is an order relation, and $(L, \preceq)$ is a totally ordered set.*

Proof. It is immediate to check that this is an order relation (it is essentially the lexicographic order), and that any $\ell_1, \ell_2 \in L$ are in relation, so $(L, \preceq)$ is a totally ordered set. □

Henceforth, all time quantities are expressed in minutes unless otherwise noted. Table 1 shows a short example of one particular bus tour. The vehicle starts at time 400 (6:40 AM) at position 0, does multiple bus legs between positions 1 and 2 with waiting times in between and finally returns to position 0. Within the same vehicle's tour, bus legs do not overlap in time. Moreover, consecutive bus legs are consecutive in space, satisfying $endPos_i = startPos_{i+1}$. A tour change occurs when a driver has an assignment of two consecutive bus legs i and j with $tour_i \neq tour_j$.

3.2 Solution

A solution S to the BDSP is an assignment of drivers to bus legs. Formally, it is represented as a partition of the set L into disjoint subsets, expressed as $S = \{s_1, s_2, \ldots, s_n\}$. Each block s_i is called a **shift**. In practical terms, a *shift* corresponds to the work scheduled to be performed by a driver in one day (Wren 2004).

For a given shift s, we denote $L_s \subseteq L$ as the subset of bus legs assigned to s. While s and L_s are mathematically equivalent, we conceptually distinguish them: L_s represents only a set of bus legs, whereas s also contains additional details such as breaks, passive ride time, etc. This additional information is uniquely determined by the set of bus legs L_s. Thus, the reader may treat s and L_s interchangeably, if it is convenient.

Note that a priori, the number of shifts of a solution $|S|$ is not given. Nevertheless, we may set it as large as necessary to get a feasible solution. An immediate upper bound is $|S| \leq |L|$. This represents the situation in which each shift is composed of only one leg. Note that, since the set L is totally ordered (Lemma 1), the notion of *consecutive* bus legs in a shift is well-defined. Moreover, a solution is also totally ordered by the order induced by the bus legs.

A solution S is feasible if each of its shifts $s \in S$ satisfies the following criteria:

- Changing tour or position between consecutive bus legs $i, j \in s$ requires

$$start_j \geq end_i + d_{endPos_i, startPos_j}.$$

  This implies that no overlapping bus legs can be assigned to s.
- The shift s respects all hard constraints regarding work regulations as specified in the next section.

3.3 Constraints

This section describes the constraints derived from the Austrian collective agreement (WKO 2019). Figure 1 depicts an example of a shift with three bus legs. We now list all the hard constraints of a shift s.

3.3.1 Total Time. Let f be the first leg and ℓ the last leg of the shift s.

$$T_s = end_\ell + endWork_\ell - (start_f - startWork_f) \qquad \forall s \in S \tag{1}$$

$$T_s \leq T_{\max} = 840 \qquad \forall s \in S \tag{2}$$

Equation (1) defines the *total time* of the shift s: It is the span from the start of work until the end of work. Equation (2) sets the upper bound of T_s: No driver can work more than fourteen hours.

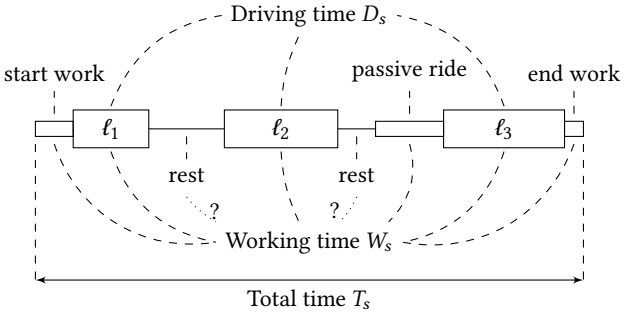


Fig. 1. Example shift (Kletzander and Musliu 2020)

3.3.2 Driving Time Regulations. First, define the *length* of a bus leg $i \in L$ as follows:

$$length_i = end_i - start_i \tag{3}$$

$$D_s = \sum_{i \in L_s} length_i = \sum_{i \in L_s} (end_i - start_i) \qquad \forall s \in S \tag{4}$$

$$D_s \leq D_{\max} = 540 \qquad \forall s \in S \tag{5}$$

Equation (4) defines the *driving time* D_s of a shift s. Equation (5) sets the upper bound of D_s to nine hours. The length of a *driving break* between two consecutive bus legs i and j is defined as

$$length_{ij} = start_j - end_i \tag{6}$$

The driving break can be split into multiple parts, all of which must be completed *before* the cumulative driving time without a break reaches four hours:

- One driving break of at least 30 min;
- Two driving breaks of at least 20 min each;
- Three driving breaks of at least 15 min each.

Once the required break time has been accumulated, a new driving block of at most four hours begins.

3.3.3 Shift Splits. We say that a shift s contains a *shift split* if there exists a pair of consecutive bus legs $i, j \in L$ such that the break between them satisfies

$$start_j - end_i - r_{endPos_i, startPos_j} \geq 180$$

Here, $ride_{p,q} = d_{p,q}$ denotes the time required for a passive ride between position p and q if $p \neq q$, whereas $ride_{p,p} = 0$. Hence, shift splits refer to breaks longer than three hours. These breaks are unpaid and therefore generally poorly regarded by bus drivers. This plays a role in the objective function.

Denote by $split_s$ the number of shift splits in shift s, and by $splitTime_s$ the total duration of these shift splits. A shift split counts as a driving break.

3.3.4 Rest break. Let $i, j \in L$ be two consecutive bus legs in a shift. The break between these two legs can be represented as the time interval $[end_i, start_j]$, and its length as $length_{ij}$ (as already defined in Equation (6)). We denote with $rest_{ij}$ the length of a *rest break*:

$$rest_{ij} = \begin{cases} length_{ij} - ride_{endPos_i, startPos_j} & \text{if } 15 \leq length_{ij} - r_{endPos_i, startPos_j} \leq 180 \\ 0 & \text{otherwise} \end{cases}$$

Rest breaks may be split into smaller parts. One part must be at least 30 min, and any additional at least 15 min. The first part must occur within the first 6 h of working time. If there is a section of a rest break of at least 15 min which is not located within the first two or last two hours of the shift, this section is considered *unpaid* (up to a maximum), as shown in Figure 2. We denote by $unpaid_s$ the sum of the length of potentially unpaid rest breaks.

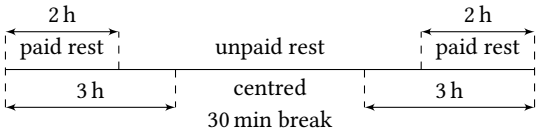


Fig. 2. Rest break positioning (Kletzander and Musliu 2020)

The maximum amount of unpaid rest is limited, as shown in Figure 2:

- If a consecutive part of a rest break of at least 30 minutes is located such that it does not intersect the first 3 h of the shift or the last 3 h of the shift, at most 1.5 h of unpaid rest is allowed and therefore we set $upmax_s = 90$;
- Otherwise, at most one hour of unpaid rest is allowed, and therefore we set $upmax_s = 60$.

Rest breaks beyond this limit are paid.

3.3.5 *Working Time.*

$$W_s = T_s - splitTime_s - \min\{unpaid_s, upmax_s\} \qquad \forall s \in S \tag{7}$$

$$W_s \leq W_{\max} = 600 \qquad \forall s \in S \tag{8}$$

Equation (7) defines the *working time* of a shift s. Equation (8) sets the upper bound of W_s as 10 h.
A minimum rest break is required according to the following options:

- $W_s < 6$ h: no rest break required;
- $6\text{ h} \leq W_s \leq 9$ h: at least 30-minute break;
- $W_s > 9$ h: at least 45-minute break.

3.4 Objective Function

Let S be a solution. The objective function to minimize is defined as follows (Kletzander and Musliu 2020):

$$z(S) = \sum_{s \in S} \left(2\, W'_s + T_s + ride_s + 30\, change_s + 180\, split_s\right) \tag{9}$$

where, for every shift $s \in S$:

- $W'_s = \max\{W_s, 390\}$, where W_s is the working time defined by Equation (7). This objective ensures that drivers are paid for at least $W_{\min} = 6.5$ hours (390 minutes).
- T_s is the total time of the shift as defined in Equation (1).
- $ride_s$ is the sum of passive ride times between consecutive legs.
- $change_s$ is the number of *tour changes*, i.e., the number of occurrences of consecutive bus legs $i, j \in s$ with $tour_i \neq tour_j$.
- $split_s$ is the number of shift splits, as defined in Section 3.3.3.

The weights were determined by previous work (Kletzander and Musliu 2020) based on preferences agreed by different stakeholders at Austrian bus companies and employee scheduling experts.

Table 2. Summary of notation used in Section 3

Symbol	Type	Description
Sets		
P	Set	Positions (bus stops and depots)
L	Set	Bus legs
S	Set	Solution: a partition of L into shifts
L_s	Set	Subset of bus legs assigned to shift s
Input Parameters (Positions)		
d_{pq}	Parameter	Travel time from position p to q (passive ride if $p \neq q$)
$ride_{pq}$	Parameter	Passive ride time: $ride_{pq} = d_{pq}$ if $p \neq q$, else 0
$startWork_p$	Parameter	Working time required to start a shift at position p
$endWork_p$	Parameter	Working time required to end a shift at position p
Input Parameters (Bus Legs)		
$tour_\ell$	Parameter	Vehicle ID for leg ℓ
$startPos_\ell$	Parameter	Starting position of leg ℓ
$endPos_\ell$	Parameter	Ending position of leg ℓ
$start_\ell$	Parameter	Departure time of leg ℓ
end_ℓ	Parameter	Arrival time of leg ℓ
$length_\ell$	Parameter	Duration of leg ℓ: $length_\ell = end_\ell - start_\ell$
Derived Quantities (Breaks)		
$length_{ij}$	Derived	Driving break length between consecutive legs i and j
$rest_{ij}$	Derived	Rest break length between consecutive legs i and j
Shift-Level Variables		
D_s	Variable	Driving time of shift s
W_s	Variable	Working time of shift s
$splitTime_s$	Variable	Total duration of shift splits in shift s
$unpaid_s$	Variable	Sum of unpaid rest break durations in shift s
$upmax_s$	Variable	Maximum allowed unpaid rest for shift s
Objectives		
W'_s	Variable	Paid working time: $W'_s = \max\{W_s, 390\}$
T_s	Variable	Total time of shift s (start-to-end span)
$split_s$	Variable	Number of shift splits in shift s
$ride_s$	Variable	Sum of passive ride times in shift s
$change_s$	Variable	Number of tour changes in shift s
Constants		
$D_{\max}$	Constant	Maximum driving time: 540 min (9 h)
$T_{\max}$	Constant	Maximum total time: 840 min (14 h)
$W_{\max}$	Constant	Maximum working time: 600 min (10 h)
$W_{\min}$	Constant	Minimum working time: 390 min (6.5 h)

4 Branch and Price

An exact solution to the BDSP can be computed by Branch and Price (B&P) (Barnhart et al. 1998). We show the general procedure in Algorithm 1. The outer loop (lines 4 to 29) iterates over the nodes in the branching tree. The inner loop (lines 9 to 16) performs Column Generation (CG) for each node. Set Partitioning is used as the master

Algorithm 1: Branch and Price

Input: Problem instance
Output: Best solution (optimal if no timeout occurs)

```
1  columns ← initialize columns;
2  best ← ∞;
3  nodes ← {root};
4  while nodes is not empty and no timeout do
5      node, columns ← choose(nodes, columns);
6      if node_lb ≥ best then
7          continue;
8      end
9      while no timeout do
10         rmp, duals ← solve relaxed master problem(columns);
11         new_columns ← solve subproblem(duals);
12         if new_columns is empty then
13             break;
14         end
15         columns ← columns ∪ new_columns;
16     end
17     if timeout then
18         if best = ∞ then
19             best ← solve integer master problem(columns);
20         end
21         return best;
22     end
23     if rmp is integer then
24         best ← min(rmp, best);
25     else
26         best ← min(solve integer master problem(columns), best);
27         nodes ← nodes ∪ branch(rmp);
28     end
29 end
30 return best;
```

problem (line 10) and the Resource Constrained Shortest Path Problem (RCSPP) as the subproblem (line 11). The duals of the relaxed master problem are used to find new shifts that have the potential to improve the solution of the master problem. This is repeated until no columns with potential for improvement (having negative reduced cost) are found (lines 12 to 14). Branching occurs when the resulting solution is not integer (line 27). The process continues until all branches either result in integer solutions (lines 23 to 24) or are cut off by the current objective bounds (lines 6 to 8).

4.1 Master Problem

The goal of the master problem is to select a subset of the shift set $\mathcal{S}$, such that each bus leg is covered by exactly one shift while minimizing total cost. This corresponds to the set partitioning problem:

$$\text{minimize} \sum_{s \in \mathbf{S}} cost_s \cdot x_s \tag{10}$$

$$\text{subject to} \sum_{s \in \mathbf{S}} cover_{s\ell} \cdot x_s = 1 \qquad \forall \ell \in L \tag{11}$$

$$x_s \in \{0, 1\} \qquad \forall s \in \mathcal{S} \tag{12}$$

Here x_s is the variable for the selection of shift s. The objective in Equation (10) minimizes the total cost, Equation (11) states that each bus leg needs to be covered exactly once (using $cover_{s\ell} \in \{0, 1\}$ as a parameter to indicate whether shift s covers leg ℓ), and Equation (12) states the integrality constraint. This constraint is relaxed to $0 \leq x_s \leq 1$ for the relaxed master problem which is repeatedly solved at each node of the branching tree (line 10 in Algorithm 1). Instead of the full set of possible shifts $\mathcal{S}$, a subset *columns* is maintained by the algorithm. Once no more new shifts can be found by the subproblem, the result of the relaxed master problem provides a local lower bound for the solution of the integer problem.

In some cases, the resulting solution might already be integer (line 23). Usually, however, the result will be fractional. Then, the integer version of the master problem is solved with the current set of columns (line 26). While there are no guarantees regarding the quality of the integer solution in this case, in practice, solutions are often very close to the lower bound already. In any case, the result from the integer master problem is a feasible solution that provides a global upper bound for the problem. If the solution is not integer, branching will be done, resuming calculation on one of the open branches.

4.2 Subproblem

The goal of the subproblem (line 11 in Algorithm 1) is to find the column (shift) with the lowest reduced cost. For the BDSP this corresponds to the Resource Constrained Shortest Path Problem (RCSPP) on an acyclic graph. In this problem, a graph $G = (\mathbf{N}, \mathbf{A})$ is given where $\mathbf{N}$ is the set of n nodes, corresponding to all bus legs, a source node h (head), and a target node t, and $\mathbf{A}$ is the set of arcs, which connect bus legs that can be scheduled consecutively in the same shift. As the bus legs are naturally ordered by time, the RCSPP is acyclic.

Each node and arc is associated with a cost and an r-dimensional resource vector representing the resource consumption when using the node or arc. A shift is defined as a path from h to t such that the path satisfies all feasibility criteria associated with the resources. Each node corresponding to a bus leg is associated with a dual from the master problem. The reduced cost of a path is therefore the cost of a path minus the sum of the duals along the path. The RCSPP aims at finding the least-cost feasible path from h to t.

There is an arc from node h to each node i except $i = t$ (indicating that a shift can start with any bus leg), and there is an arc from each node i except $i = h$ to node t (indicating that a shift can end with any bus leg). Nodes corresponding to bus legs i and j are only connected by an arc ij if chaining the bus legs is feasible according to their times and the distance matrix d. Building the graph is done once per instance in $O(n^2)$. Figure 3 shows such a graph for a small toy instance. The remainder of this section goes into the details of this graph, its costs, and its constraints. Due to the complex regulations, solving this sub-problem is very challenging and requires several novel optimizations. While the rules regarding individual resources are problem-specific, the optimizations are general for solving high-dimensional RCSPPs.

The problem is solved using a label setting algorithm (Irnich and Desaulniers 2005). For each node i it maintains a set of labels, which are partial paths from h to i which are non-dominated according to both cost and all

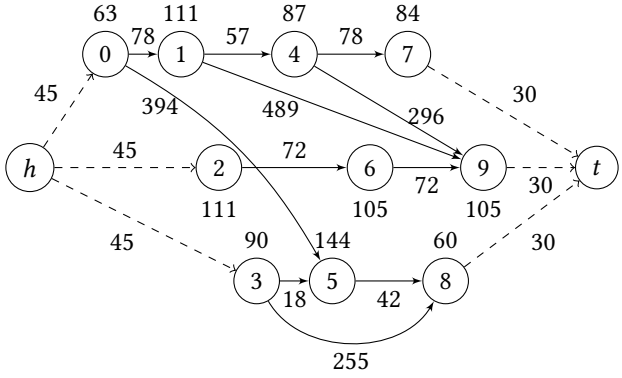


Fig. 3. RCSPP graph for a Toy Instance. Bus legs, represented by nodes, are numbered 0 to 9 according to their order by start time. Each row represents one vehicle: The first vehicle covers legs 0, 1, 4, 7. The second covers legs 2, 6, 9. The third covers 3, 5, 8. Horizontal arcs connecting consecutive bus legs show that it is always possible to stay on the same vehicle. Additional arcs between bus legs show possibilities to change from one vehicle to another, or to skip legs on a vehicle during a break. Nodes h and t are connected to all bus legs (shifts can start and end at any point during a vehicle schedule), for visual clarity, we show only some of these arcs. We show costs for both nodes and arcs as well, resources are not visualized.

resources (i.e., an $r + 1$-dimensional Pareto front). Due to the acyclic nature of the graph, each node can be processed in temporal order, starting with an initial label representing an empty path at h where each resource usage is 0. For each node i and each non-dominated label x on that node, all arcs ij are processed and a label y is placed at the end node j of the arc unless some constraints are violated. Therefore, each label x represents a path from h to its node i, capturing the nodes in the path, the cost, and the resource usage. All labels collected at t with negative reduced cost correspond to columns that can be added to master problem.

4.2.1 Costs. The costs for nodes and arcs are based on the objective in Equation (9). The cost c_i for each node i corresponding to a bus leg i is defined as $c_i = 3 \cdot length_i$ as each bus leg contributes its length to both W_s (weight 2) and T_s (weight 1). By definition, $c_h = c_t = 0$.

Several helpful properties of arcs from node i to node j are defined and used for defining the arc costs:

$$length_{ij} = \begin{cases} start_j - end_i & \text{if } i \neq h \wedge j \neq t \\ startWork_{startPos_j} & \text{if } i = h \\ endWork_{endPos_i} & \text{if } j = t \end{cases} \tag{13}$$

$$ride_{ij} = \begin{cases} d_{endPos_i, startPos_j} & \text{if } i \neq h \wedge j \neq t \wedge endPos_i \neq startPos_j \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

$$change_{ij} = \begin{cases} 1 & \text{if } i \neq h \wedge j \neq t \wedge tour_i \neq tour_j \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

$$split_{ij} = \begin{cases} 1 & \text{if } i \neq h \wedge j \neq t \wedge length_{ij} - ride_{ij} \geq 3 \cdot 60 \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

$$remain_{ij} = \begin{cases} length_{ij} - ride_{ij} & \text{if } split_{ij} = 0 \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

$$rest_{ij} = \begin{cases} remain_{ij} & \text{if } i \neq h \wedge j \neq t \wedge remain_{ij} \geq 15 \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

Equation (13) defines the length of an arc, taking into account start and end arcs. Equation (14) states the passive ride time, Equation (15) whether a tour change occurs, Equation (16) whether the arc corresponds to a shift split, Equation (17) captures the remaining arc length after removing the passive ride time and the shift split time, and Equation (18) captures a potential rest break. These definitions extend those in Section 3 to the source and sink nodes.

The cost c_{ij} for arc ij (i.e., bus leg i to j) is defined as

$$c_{ij} = 2 \cdot remain_{ij} + length_{ij} + 3 \cdot ride_{ij} + 30 \cdot change_{ij} + 180 \cdot split_{ij} \tag{19}$$

Note that unpaid rest cannot be determined at this point, therefore all rest is treated as paid for this computation. Unpaid rest is separately treated when solving the subproblem. The term $ride_{ij}$ contributes to both working time W_s and the additional passive ride penalty.

4.2.2 Resources. In total, eleven resources need to be tracked to satisfy the BDS constraints. The first resources are classical additive resources with a maximum allowed usage.

$$d_y = d_x + length_j \tag{20}$$

$$s_y = s_x + length_{ij} + length_j \tag{21}$$

Equation (20) tracks total driving time, Equation (21) the span.

$$rd_y = \begin{cases} true & \text{if } length_{ij} \geq 30 \vee (length_{ij} \geq 20 \wedge b20_x \geq 1) \vee (length_{ij} \geq 15 \wedge b15_x \geq 2) \\ false & \text{otherwise} \end{cases} \tag{22}$$

$$dc_y = \begin{cases} 0 & \text{if } rd_y \\ dc_x + length_j & \text{otherwise} \end{cases} \tag{23}$$

$$b15_y = \begin{cases} 0 & \text{if } rd_y \\ b15_x + 1 & \text{if } \neg rd_y \wedge length_{ij} \geq 15 \\ b15_x & \text{otherwise} \end{cases} \tag{24}$$

$$b20_y = \begin{cases} 0 & \text{if } rd_y \\ b20_x + 1 & \text{if } \neg rd_y \wedge length_{ij} \geq 20 \\ b20_x & \text{otherwise} \end{cases} \tag{25}$$

Resources for monitoring drive breaks need to be reset at each full drive break. Equation (22) defines a helping flag to indicate whether the current drive block is finished. Equation (23) uses this flag to reset or increase the current driving time. Equation (24) tracks the number of 15-minute breaks in the current driving block and Equation (25) the number of 20-minute breaks. The number of 30-minute breaks does not need to be counted as each of them resets the driving block.

Constraints regarding rest breaks need to consider different sums of rest breaks and their positioning.

$$r_y = \min(r_x + rest_{ij},\ 45) \tag{26}$$

$$b30_y = b30_x \vee rest_{ij} \geq 30 \tag{27}$$

$$rest'_{ij} = rest_{ij} - \max(2 \cdot 60 - s_x,\ 0) - \max(end_i + rest_{ij} - (e_j - 2 \cdot 60),\ 0) \tag{28}$$

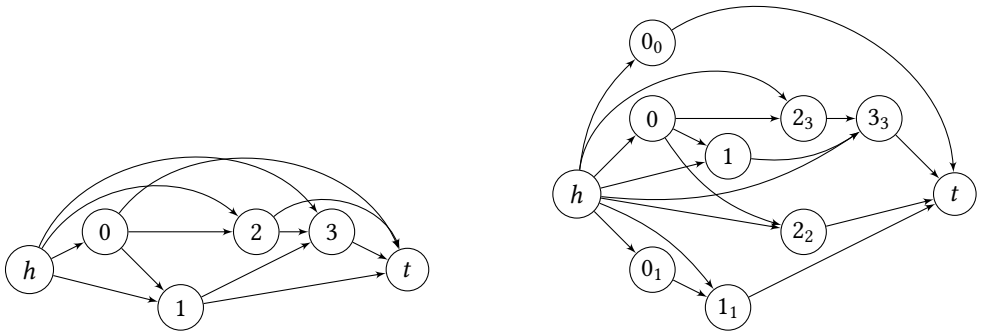
$$u_y = \min\left(u_x + \begin{cases} rest'_{ij} & \text{if } rest'_{ij} \geq 15 \\ 0 & \text{otherwise} \end{cases},\ 90\right) \tag{29}$$

$$w_y = w_x + u_x - u_y + remain_{ij} + ride_{ij} + length_j \tag{30}$$

$$rest''_{ij} = rest_{ij} - \max(3 \cdot 60 - s_x,\ 0) - \max(end_i + rest_{ij} - (e_j - 3 \cdot 60),\ 0) \tag{31}$$

$$bc30_y = bc30_x \vee rest''_{ij} \geq 30 \tag{32}$$

Equation (26) tracks the amount of required rest time, capping it at 45 as higher values do not matter. Equation (27) deals with the occurrence of a 30-minute rest break.



(a) Original graph (b) Extended graph

Fig. 4. Extension of RCSPP graph to deal with distance to shift end. E.g., going from node 0 to 2 requires to know whether node 2 is the last node before t (node 2_2 in the extended graph), or node 3 is the last node before t (node 2_3, only connected to 3_3 but not t in the extended graph).

Equation (28) captures the part of the rest break that can be unpaid depending on the position. However, this requires knowing the end time of the shift end_y which violates the general assumption of the algorithm that nodes can be processed in temporary order. Therefore, for each node j, with known end time end_j if j is the last bus leg in the shift, new nodes are added to the graph: $\mathbf{N}_j$ is the set of nodes reachable within 3 hours from end_j when traversing the network backwards from j (including j itself). For each node $i \in \mathbf{N}_j$ a new node i_j is created. For each pair of nodes i and k both in $\mathbf{N}_j$, an arc $i_j k_j$ is added if ik is in the original graph. For $i \notin \mathbf{N}_j$ and $k \in \mathbf{N}_j$, an arc ik_j is added if ik is in the original graph. The arc $j_j t$ replaces the arc jt. Each new node k is associated with the shift end time $e_k = end_j$. All original nodes have shift end time $e_k = \infty$. This solves the problem, but increases the graph size by a factor of 6 for small instances up to more than 30 for large instances. Figure 4 shows this extension on a very small graph, assuming that only node $n-1$ is in the 3 hour window ending at the end time of node n. The extended graph contains sub-networks for each final node.

The potential total amount of unpaid rest is captured in Equation (29). Equation (30) tracks working time, assuming that all of u_y is unpaid so far, which constitutes a lower bound for the actual value. Equation (32) tracks the existence of a centered 30-minute break using $rest''_{ij}$ defined like in Equation (28) except with $3 \cdot 60$ instead of $2 \cdot 60$.

Finally, the cost needs to be computed.

$$cost'_y = cost'_x + 2 \cdot (u_x - u_y) + c_{ij} + c_j - dual_j \tag{33}$$

$$cost_y = cost'_y + 2 \cdot \max(W_{min} - w_i,\ 0) \tag{34}$$

Equation (33) takes into account the unpaid rest and the duals for the bus legs. Equation (34) further considers the minimum working time.

4.2.3 Constraints. A shift is a path from h to t that does not violate any resource constraints. The following resource constraints need to be satisfied for every label x: $d_x \leq D_{max}$, $s_x \leq T_{max}$, $dc_x \leq 4 \cdot 60$, $w_x \leq W_{max}$, and $r_x \geq 15$ if $w_x \geq 6 \cdot 60$. Additionally, at node t, all labels x need to satisfy: $r_x \geq 45$ if $w_x > 9 \cdot 60$ and $b30_x$ if $w_x \geq 6 \cdot 60$.

4.2.4 Dominance. To track only paths with the potential to become minimum-cost paths, only pareto-optimal labels in both cost and resource usage are kept at each node, i.e., labels which are not dominated. Regarding more complex constraints, it is important to observe that, if label x dominates label y at a node i, extensions of x to future nodes must still dominate extensions of y. Finally, the least-cost label at node t represents the minimum-cost path.

Therefore, a label x dominates label y if all following conditions are true: $d_x \leq d_y$, $s_x \leq s_y$, $dc_x \leq dc_y$, $b20_x \geq b20_y$, $b15_x \geq b15_y$, $w_x + u_x \leq w_y$, $r_x \geq r_y$, $b30_x \vee \neg b30_y$, $bc30_x \vee \neg bc30_y$, $c_x + 2 \cdot u_x \leq c_y$.

In general, lower values dominate for resources with upper bounds and higher values dominate for resources with lower bounds. The most complex case arises from the fact that the maximum value of unpaid break might be 0, 60, or 90 depending on the existence and position of a 30-minute rest break. This results in neither higher nor lower values of u_i being dominant. Rather, the uncertain amount of unpaid rest weakens the domination power of cost and working time, as those might vary in the amount of unpaid rest until the end of the path. However, in Section 4.3.1 a more useful way to deal with this problem is described.

4.3 Handling the Subproblem Dimensionality

As described in the previous section, due to the complex constraints for valid shifts, the subproblem complexity is very high. In particular, eleven resources are tracked in the labels. However, the efficiency of the label-setting algorithm depends on its ability to keep a small number of non-dominated labels. The more dimensions the labels have, the easier it is for labels to be non-dominated, drastically increasing the number of labels for processing. Therefore, the increase in the number of processed labels turns out to be the bottleneck of scaling the solution method to larger instances. Several novel improvements to reduce this bottleneck are introduced. These are generally applicable to other problems with similar characteristics.

4.3.1 Subproblem Partitioning. Instead of generating all possible shifts from one graph, the subproblem is split into three similar, but disjoint RCSPP problems, removing the uncertainty for unpaid breaks depending on whether the maximum amount of rest break is 0, 60, or 90. This depends on the existence and position of a 30-minute rest break. Note that this is a full partitioning of the original sub-problem, i.e., each possible shift can be created by exactly one of the three subproblems, preserving optimality of the overall algorithm.

- *No 30-minute rest break*: All arcs corresponding to rest breaks of at least 30 minutes can be removed, making the graph very sparse and therefore fast to process. Unpaid rest is guaranteed to be 0, all rest break resources are ignored.
- *Uncentered 30-minute rest break*: $b30_y$ must be true at t. The maximum of u_y is changed to 60, but the current amount of u_y is guaranteed to be unpaid. Therefore, $w_x \leq w_y$ and $w_x + u_x \leq w_y + u_y$ can be used instead of $w_x + u_x \leq w_y$ as the domination criterion (same change for c_x and c_y), reducing the number of undominated labels.
- *Centered 30-minute rest break:* $b30_y$ and $b30c_y$ must be true at t. The maximum of u_y is set to 90, but the improved dominance criteria from the previous graph still apply as again all of u_y is guaranteed to be unpaid.

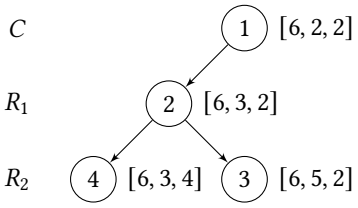
The pricing subproblem is used to add multiple columns (up to 1000 per graph) at once. Indeed, solving the relaxed master problem even with thousands of columns is much faster than solving the pricing subproblem. Therefore, accepting some unnecessary columns while reducing the number of times the subproblem is solved pays off. The different graphs are ordered by their complexity, measured by the number labels they expand. The pricing subproblem avoids searching more complex graphs, as long as those with less complexity still produce enough shifts with negative reduced costs (up to $col_{\max}$, with $col_{\max} = 10$ by default, $col_{\max} = 100$ if the objective in the master problem did not change, $col_{\max} = 1000$ if the objective did not change repeatedly).

4.3.2 Cost Bound. An upper bound for the reduced cost for each node can be computed by processing the RCSPP graph backwards with the current duals. The upper bound describes the maximum reduced cost at each node, such that there is still a path to arrive at t with reduced cost < 0, disregarding resource constraints. All labels above this bound can be discarded during the solution process.

4.3.3 Exponential Arc Throttling. While it might be hard to find the best column, at least in the beginning of the solving process, there are many columns with negative reduced cost. Two options were considered to reduce the size of the subproblem in early iterations. The first option is to reduce the number of labels at each node. This can be done either by rejecting new labels after a certain threshold or only retaining labels according to certain criteria, e.g., the best 100 by cost.

It is more promising to exploit the fact that good columns are unlikely to include very costly arcs. We propose a new throttling mechanism that imposes a maximum cost per arc and gradually increase this cost limit over time. We define a cost limit $c_\ell = c_{start}$ and remove all arcs ij with cost $c_{ij} > c_\ell$. The initial limit of $c_{start} = 100$ is chosen for this particular problem as it just enough to cover a 30 minute break. This greatly reduces the size of the graph and therefore the number of labels.

Once the number of new columns is below $col_{\max}$ (over all subproblems for different rest breaks combined), the limit c_ℓ is multiplied by 2 for the next time the subproblem is solved. This is repeated until all arcs are included. While a lot of arcs are not present in the early stage, the focus on arcs that are likely to appear in good columns leads to very fast convergence to good solutions, even if there is not enough time to finish the column-generation process. On the other hand, increasing the limit until all arcs are included ensures that the problem can still be solved to optimality if the given runtime allows.



Label	C	R_1	R_2
1	8	2	3
2	7	4	3
3	6	5	2
4	6	3	4

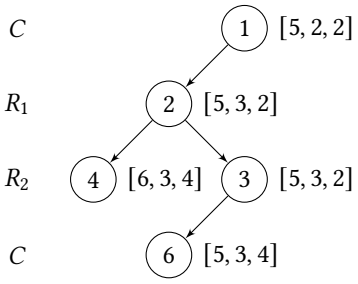
Fig. 5. Example k-d tree

4.3.4 Improved Elimination of Non-Dominated Labels. Even with the previously mentioned improvements, the majority of runtime is spent figuring out which labels are non-dominated (maintain the Pareto front regarding cost and all resources for each node). The naive approach is to compare each new label on a node with each label in the current set of non-dominated labels. However, the runtime complexity of this method is quadratic in the number of non-dominated labels. Therefore, two different methods are explored to speed up the dominance checks. The first method is the multi-dimensional divide-and-conquer (Bentley 1980): It is based on recursively solving the k-dimensional dominance problem with n labels by two k-dimensional problems of size $n/2$ and one $k-1$-dimensional problem of size n.

The second method is a two-stage approach based on k-d trees and bounding boxes (W.-M. Chen et al. 2012). Instead of just storing the labels at node i in a list, they are additionally stored in a binary tree structure. Figure 5 shows a simple example of a k-d tree for node i in the RCSPP graph for a cost component C and two resource components R_1 and R_2. Each label x represents a partial path from node h to node i with cost and resource usage as given in the corresponding table. Each level in the tree is associated with a dimension (cost or resource). Once all dimensions are exhausted, they repeat from the start for deeper levels. With $R = [C, R_1, R_2, \ldots]$, level v is associated with the resource at the 0-based index v mod $|R|$ in R.

For simplicity, the example in Figure 5 assumes R_1 and R_2 are additive resources where lower consumption is better. The example contains four non-dominated labels. Label 1 was added first as the root. Label 2 is better in cost, therefore added as the left child of 1. Label 3 is better in cost compared to 1, and worse in R_1 compared to 2, while label 4 is also better in cost compared to 1, but better in R_1 compared to 2. Each label is associated with a bound u_x that stores the best value for this dimension in the whole sub-tree (e.g., $[6, 2, 2]$ captures the minimum cost 6 and the minimum resource usage of 2 for each of the resources).

Assume a new label 5 with $C_5 = 7$, $R_1^5 = 5$, and $R_2^5 = 4$ is added. It is dominated by u_1, so it is compared to label 1, which is incomparable. Next it is also dominated by u_2, so it is compared to label 2, which dominates it, and label 5 gets discarded.



Label	C	R_1	R_2
1	8	2	3
2	7	4	3
3	6	5	2
4	6	3	4
6	5	3	4

Fig. 6. Example k-d tree after inserting label 6

Assume a new label 6 with $C_6 = 5$, $R_1^6 = 3$, and $R_2^6 = 4$ is added. This label is not dominated by any label in the tree, therefore, it should be added. It is compared with u_1, and is not dominated by this bound. Therefore, it is immediately clear that it is not dominated by any label in this sub-tree (in this case the whole tree), and no further comparisons are needed. This bound comparison greatly reduces the total number of comparisons. Figure 6 shows the tree after insertion of label 6.

Label 6 dominates label 4, however, no removals are done to avoid costly reorganizations of the tree. By insertion order, a label y added after label x can dominate x, but never be dominated by y (otherwise it would never be added to the tree). When the corresponding node is expanded in the RCSPP graph, the whole tree is rebuilt in reverse order. If y dominates x, it will prevent x from being added to the rebuilt tree, leaving only non-dominated labels.

4.4 Branching

Branching (line 27 in Algorithm 1) is performed on the connections between bus legs in a shift that correspond to arcs in the subproblem. The branching considers a connection between bus legs i and j that appears fractionally in the solution to the relaxed master problem, and selects the most fractional (closest to 0.5) to maximize impact. In the left branch, all columns containing i or j, but not both consecutively, are removed. In the RCSPP graph all outgoing arcs from i except for the one connecting to j are removed. In the right branch, all columns with i and j assigned consecutively are removed, as well as the arc from i to j in the RCSPP graph.

Algorithm 2: Adaptive Large Neighborhood Search

Input: k_0 (initial destruction size)
Output: S_{bsf} (best solution found)

$k \leftarrow k_0$;
Construct the initial solution S_0 using a greedy algorithm;
$S_{\text{bsf}} \leftarrow S_0$;
Initialise the weights $\boldsymbol{\rho}$;
while *time* $< t_{\max}$ **do**
  Select destroy operator $\omega \in \Omega$ using $\boldsymbol{\rho}$;
  $S' \leftarrow r(\omega(S_{\text{bsf}}, k))$;
  if $z(S') < z(s_{\text{bsf}})$ **then**
    $S_{\text{bsf}} \leftarrow S'$;
  end
  Update weights $\boldsymbol{\rho}$ and sub-problem size k;
end
return S_{bsf};

4.5 Lagrangean Bound

In the case when the column generation terminates at the root node, small optimality gaps are typically achieved. However, when there is not enough time to complete the column generation at the root node, a Lagrangean lower bound can be computed as $O(RMP) + \kappa \cdot O(PP)$, where $O(RMP)$ is the optimum of the reduced master problem, while κ is an upper bound on the number of shifts and $O(PP)$ is the optimum of the pricing problem. The value κ can be bounded by $\kappa = \lfloor O(RMP)/minCost \rfloor$, where $minCost = 2 \cdot W_{min} + \min_{\ell \in \mathbf{L}} length_\ell$. The minimum reduced cost is only known when no more throttling is in play. Otherwise a lower bound for the minimum reduced cost can be computed where the constraints regarding break positions for unpaid breaks are relaxed. However, these bounds turned out to be rather weak for the BDSP.

5 Large Neighborhood Search

The *Large Neighborhood Search* (LNS) algorithm was introduced by Paul Shaw in 1998 (Shaw 1998). The algorithm starts from an already feasible solution. A part of the current solution is destroyed to obtain a sub-problem that is easy to solve optimally or at least close to optimality. Selecting the part to destroy is done by a set Ω of *destroy operators* (or *destroyers*), the operator to apply is chosen randomly proportional to a given weight vector $\boldsymbol{\rho}$. Solving the sub-problem is done by a *repair operator*, often an exact method. We accept the new solution S' if $z(S') < z(S_{\text{bsf}})$, where z represents the objective function value described in Equation (9) and s_{bsf} is the best-so-far solution. Algorithm 2 shows the pseudo-code of the algorithm.

5.1 Destroy Operators

Since our repair mechanism can only produce complete shifts, the aim of the destroyers is to select a subset of shifts $S' \subseteq S$ that is removed from the current solution. The size of the sub-problem $k = |S'|$ is given to the destroy operator. We propose three distinct ways to select S':

Shift uniform (ω_{SU}): Select k shifts uniformly.
Shift weighted (ω_{SW}): $\left\lfloor \frac{k}{2} \right\rfloor$ of the shifts are selected using their cost as weight, the others uniformly. This is motivated by the fact that shifts with high cost have a higher potential to benefit from reoptimization. The

split is done since a combination of high-cost and low-cost shifts can have a better potential to balance the shifts in the sub-problem, e.g., by transferring some legs from the high-cost shift to an underutilized shift.
Tour remover (ω_{TR}): A tour is uniformly selected and all shifts that share at least one leg of this tour are removed. This process is iterated until at least k shifts are removed. This operator is based on the idea of selecting shifts that have something in common and therefore have a higher potential that useful recombinations of their shifts are possible, e.g., optimizing when and where a bus is handed over from one driver to the other. Note that this operator might select more than k shifts because it removes all the shifts who share a tour. However, tours are usually not shared by too many shifts since this incurs extra cost, so $|S'|$ does typically not exceed k by much.

5.2 Repair Operators

Once a set of removed shifts S' is selected, the repair mechanism needs to solve the sub-instance that is created by using all legs ℓ assigned to any shift $s \in S'$ together with the common data for the whole instance. This sub-instance represents a complete instance of BDSP and can therefore be solved with any solution method of choice. Since the Branch and Price (B&P) approach presented in Section 4 is the most powerful for small instances (it can provide an optimal solution for instances with 10 tours within seconds), it is the best fit for solving these sub-instances.

However, as the evaluation in Section 7 shows, for small instances, the results are very close to the optimum when only solving Column Generation (CG) on the root node and then solving the master problem with integrality constraint on the set of columns obtained during CG. These solutions are often much faster, but achieve a gap of around 1% while the following branching process only closes this remaining gap very slowly.

Therefore, we propose to drop the aim of optimally solving the sub-instance with B&P, and instead only use CG on the root node to get very good solutions to the sub-instance very fast. In the evaluation, we compare the repair operators using Column Generation (r_{CG}) and full Branch and Price (r_{BP}).

Once the repair mechanism returns a solution consisting of shifts S^* that contain all bus legs from S', the new solution for the full problem is provided by $(S \backslash S') \cup S^*$.

5.3 Sub-Problem Size

An important parameter for Large Neighborhood Search is the size of the sub-problem. However, the appropriate size depends on the destroy and repair operators. In the case of our system, the destroy operators are easy and fast to apply, but the complexity of Branch and Price increases rapidly with the size. Even when just applying Column Generation, the size of the RCSPP in the sub-problem still leads to considerable increases in runtime.

Therefore, based on preliminary experiments, the smallest sub-problem size in use is $k = 5$. This size can still be solved in a few seconds, so it is fast enough, but it also leads to a high number of improvements, so it is large enough to allow meaningful changes of the solution. In the process of the search this size can be increased if too many iterations without improvement occur. This indicates that a larger size might be needed to escape local optima.

We use a maximum size of $k_{\max} = 20$ since runtime grows rapidly and for larger size too much time would be spent on each individual sub-problem. When running the algorithm, the size starts with an initial value of k_0, and is increased by 1 until reaching $k_{\max} = 20$ whenever the previous improvement was more than $n_{\max}$ iterations ago. As soon as an improvement is found, k is set back to the initial value k_0.

5.4 Adaptivity

Adaptive Large Neighborhood Search (ALNS) is an extension of LNS, where the weights $\boldsymbol{\rho}$ for selecting the operators are adapted dynamically based on their performance (Ropke and Pisinger 2006).

Our method takes into account the score and the time required by destroy operator i. At first, every component of the weight vector $\boldsymbol{\rho}$ is set to $\frac{1}{|\Omega|}$. The destroy operator is selected in a random way with weights $\boldsymbol{\rho}$ using the *roulette wheel principle*:

$$\mathbb{P}\left(i\text{-th operator is selected}\right) = \frac{\rho_i}{\sum_{j=1}^{|\Omega|} \rho_j} \tag{35}$$

The selected destroy operator is then applied to the current solution S, which results in a sub-problem that is passed to the repair operator r. We update the weights considering the number of successes and the total time of its selections. A similar approach was used in a related crew scheduling domain (Carmo Martins and Silva 2019). At iteration n, we update the weight ρ_i^n of the i-th destroy operator using the following equation:

$$\rho_i^{n+1} = \lambda \rho_i^n + (1-\lambda)\,\frac{\sum_{j=0}^{n} \sigma_i^j}{\sum_{j=0}^{n} \tau_i^j} \tag{36}$$

where $\sigma_i^j = 1$ if the i-th operator has improved the best-known-solution at iteration j, else 0. The denominator is a sum of runtimes, so τ_i^j represents the time the i-th operator took for the entire process (destroying + repairing) at iteration j. If operator i was not selected at iteration j, then $\sigma_i^j = \tau_i^j = 0$. The real parameter $\lambda \in [0, 1]$ controls the sensitivity of the weights. A value of λ close to 0 implies that the operator performance during the search has a large influence while a value of 1 keeps the initial weights static.

As long as the denominator is 0, the value of the fraction is set to 0. In this case, $\rho_i^{n+1} = \lambda \rho_i^n$.

6 Integration of Column Generation and Large Neighborhood Search

When applying Large Neighborhood Search (LNS) on an optimization problem, by default the repair operators are used in a black-box fashion: The current sub-problem is fed into the operator, the corresponding solution is used to update the solution to the overall problem, and it does not matter how it was obtained. Each sub-problem is solved from scratch, and all additional information gained while solving the sub-problem is discarded every time.

However, this might actually be inefficient, as information from solving each sub-problem might be useful for future sub-problems, or it might be used beyond individual sub-problems to globally enhance the best solution. In this section, we present two novel integration techniques for combining LNS with a Column Generation (CG) repair operator that are generally applicable for this combination of solution methods.

6.1 Column Storage

The first integration is dedicated to the reuse of information between sub-problems. Recall that each shift (column) s generated by CG contains a subset of the bus legs $L_s \subseteq L$. Each time a subproblem i consisting of legs $L_i \subseteq L$ is solved, a large set of columns $\mathcal{S}_i$ is generated, and a solution to the sub-problem $S_i^* \subseteq \mathcal{S}_i$ is returned.

By default, the columns are regenerated for each sub-problem, however, the same columns might be generated repeatedly by multiple sub-problems. Therefore, for a potential improvement, a column storage $\hat{\mathcal{S}}$ is introduced, and after solving a sub-problem, the update in Equation (37) is performed.

$$\hat{\mathcal{S}} \leftarrow \hat{\mathcal{S}} \cup \texttt{select}(\mathcal{S}_i) \tag{37}$$

A subset of the newly generated columns is added to the column storage $\hat{\mathcal{S}}$, where the selection criteria can be chosen freely, including no storage, or storing all columns.

Now, each time a sub-problem i is solved, the set of columns $\mathcal{S}_i$ can be initialized as seen in Equation (38).

$$\mathcal{S}_i \leftarrow \{s \in \hat{\mathcal{S}} \mid \forall \ell \in L_s :\ \ell \in L_i\} \tag{38}$$

Each column from the storage that only contains legs that are part of the sub-problem are added to the initial set of columns, therefore, these columns do not need to be rediscovered again in the current sub-problem. Using this column reuse strategy is denoted by adding +R to the LNS version.

6.2 Global Background Solver

The second improvement adds the global view of B&P into the local view based on sub-problems in LNS. The whole set of columns $\hat{\mathcal{S}}$ can be used in a global master problem as described in Section 4.1. Since in LNS we do not aim to solve the problem exactly, there is no need to solve the relaxed master problem, instead the integer problem can be solved to get the best possible solution for the current set of stored columns.

However, solving increasingly larger integer master problems takes time and memory. Therefore, we propose to use a second thread for this improvement, using the first thread entirely for LNS, while the background thread repeatedly solves the master problem for $\hat{\mathcal{S}}$. At the end of every repairing phase of LNS, the algorithms checks whether the solution from the second thread is better than the current best:

$$s_{\text{bsf}} \leftarrow \operatorname{argmin}\left(z(s'), z(s_{\text{bsf}}), z(s_{\text{bg}})\right)$$

where s_{bg} is the solution from the second thread, s_{bsf} is the best-so-far and s' is the solution after the repairing phase, as described in Algorithm 2. If s_{bg} is better, it replaces s_{bsf}, and LNS continues from this improved solution. This is only a mild form of parallelization that is easily applicable with current multi-core machines, but can be very beneficial to further improve the joint performance of the methods.

As the background solver repeatedly solves similar problems, and new columns are frequently added, two more considerations are relevant. First, columns for this solver are never removed, but only added, allowing a warm start from the previous result in each solving cycle. Second, the master problem might not be solved to optimality, but stopped according to a given criterion, since incorporating new columns might be more beneficial than spending more time on the current cycle. We propose to use a timeout t_{bg}, but at least solve the root node of the MIP, before ending the current cycle. Adding the background solver is denoted as +B for the LNS version.

This improvement using the background solver can also be used for Branch and Price on its own. By default, the integer master problem is solved there whenever Column Generation on a node is finished (line 26 in Algorithm 1), or in case even the first node runs into timeout (line 19). However, as each integer solution is a global upper bound independent of the current position in the branching tree, this process can be done in the background solver as well on the set of columns $\mathcal{S}$. We call this version BP+B in the evaluation.

6.3 Selection of Columns to Store

A critical parameter for the proposed improvements is the selection of columns to store via the `select` function. The easy way is to store all new columns, we refer to this option as (F) (full set). However, this might use a considerable amount of memory. Further, the time to search for useful columns for the current sub-problem might counter the benefit of not having to rediscover them, and the background master problem might get excessively large.

A very lightweight alternative would be to only select the best subset S_i^* for each sub-problem. We denote this version as (B) (best). The focus on only the best columns will keep the size of $\hat{\mathcal{S}}$ low, but ideally still preserve very good solutions, while the selection of different sub-problems over time should still provide some diversity.

7 Evaluation

All executions were performed on a cluster with 11 nodes using Ubuntu 22.04.2 LTS (GNU/Linux 6.8.0-48-generic x86_64) set up for maximum reproducibility (Fichte et al. 2024). Each node has two Intel Xeon E5-2650 v4 processors (max 2.20 GHz, 12 physical cores, no hyperthreading). Unless otherwise specified, we use a single

thread and up to 25.6 GB of RAM. Unless stated otherwise, all experiments are performed with a fixed budget of 1 h of CPU-time. Deterministic algorithms are executed once, non-deterministic ones are executed 10 times for each instance. Random seeds for different runs are recorded for reproducibility.

The LNS implementation is written in Python, executed with PyPy 7.3.17 for speed using Python 3.10.14. Branch and Price was implemented in Java, using OpenJDK 23.0.1, and Gurobi 12.00 for the master problem. The figures were generated using Matplotlib (Hunter 2007) v3.10.0 and Seaborn (Waskom 2021) v0.13.2.

Software Changes. To get a fair comparison, we reran the experiments from the previous publications (Kletzander, Musliu, and Van Hentenryck 2021; Mazzoli et al. 2024). We updated previously used software versions, but the largest change was switching from CPLEX to Gurobi. This was because at least the Java interface of CPLEX is very memory-hungry, leading to frequent out-of-memory errors for larger instances, and frequent crashes of the CPLEX-Java interface when Java was in the process of garbage collection (freeing unused memory). In contrast, Gurobi works on all instances, even very large ones, without running out of memory.

Evaluation Metric. To have a metric quality that does not scale with the dimension of an instance, we evaluate the quality of a solution using the relative gap (GAP) compared to the best-known solution:

$$\mathrm{GAP}(x) = \frac{z(x) - z\left(x_{\mathrm{bks}}\right)}{z\left(x_{\mathrm{bks}}\right)} \cdot 100, \tag{39}$$

where x is the solution and x_{bks} is the best-known solution among all methods evaluated in this article.

Instances and Initial Solutions. We use the publicly available sets of benchmark instances provided by previous work (Kletzander, Mazzoli, et al. 2022; Kletzander and Musliu 2020)[1]. There are 65 instances in 13 sizes, ranging from around 10 to around 250 tours. Note that the last three sizes have much larger size changes than the earlier ones.

The instances in this paper use $startWork_p = 15$ and $endWork_p = 10$ at the depot ($p = 0$). These values are 0 for other positions. For the given instances, the number of legs is proportional to the number of bus tours with approximately $n_{\mathrm{legs}} \approx 10 \cdot n_{\mathrm{tours}}$.

The initial solutions required by LNS have been generated using a greedy construction method (Kletzander and Musliu 2020), assigning each bus leg to the shift with the lowest additional cost, or to a new shift if this would incur an extra cost of at most 500 compared to the best existing shift assignment. B&P starts from a very simple solution where each bus leg forms its own shift as starting from the greedy solution did not improve the results.

7.1 Branch and Price

This subsection provides the evaluation of important B&P components and their effect, as well as the comparison of using a single thread (BP), parallelization of the MIP (BP_M), and mild parallelization using the background thread (BP+B).

7.1.1 Effects of Subproblem Improvements. We carefully tested all the design choices and parameter settings; this section focuses on the effects of our crucial subproblem improvements. Note that these experiments were done on a faster machine with an i7-6700K with 4x4.0 GHz and with CPLEX as the MIP solver.

We highlight the benefits for splitting the subproblem on the example of instance 40_16. At the root node the subproblem is solved 197 times. Only 76 times the uncentered 30-min break graph is solved and only 14 times the centered 30-min break graph. However, the runtimes for the different graphs are 4 ms, 1.5 s and 1.9 s respectively at the beginning (arc throttling) and 16 ms, 7.7 s and 51.1 s at the end (all arcs). Even though easier subproblems

[1] https://cdlab-artis.dbai.tuwien.ac.at/papers/sa-bds/

only provide parts of the new shifts, their runtime advantage makes it worth only going to the more complex subproblems when necessary.

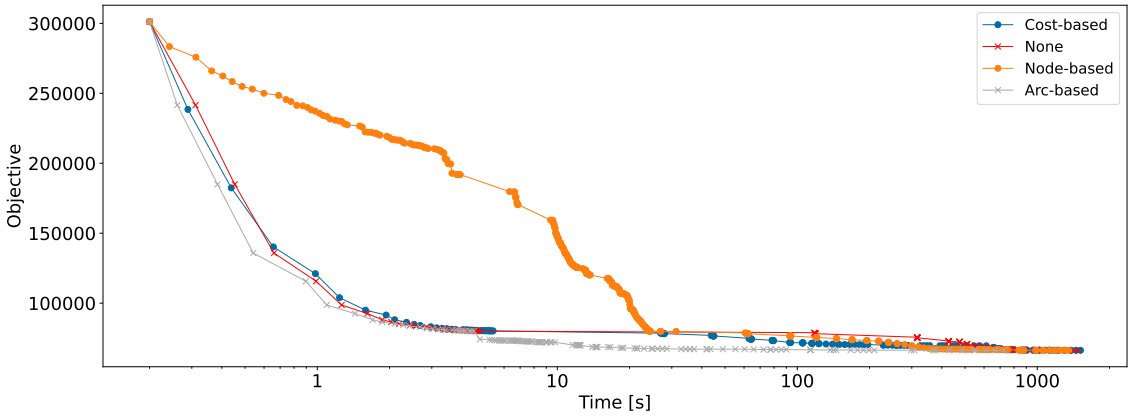


Fig. 7. Throttling approaches for 40_16

Figure 7 shows the comparison of cost-based throttling (100 best regarding cost per node, factor 10 increase), no throttling, node-based throttling (100 first per node, factor 10 increase), and arc-based throttling (arc cost limited to 100, factor 2 increase) on the objective of the reduced master problem over time (logarithmic time axis to highlight the differences early in the search). The results show a significant overhead of cost-based throttling (taking the longest time overall), and weak improvements early but more efficiency later for node-based throttling. Arc-based throttling, however, clearly shows superior performance starting to drop significantly in objective value compared to other approaches in less than 10 seconds. Its overall time is 727 seconds compared to 1354 seconds for node-based throttling, the second best approach.

Regarding the dominance algorithm, one run of the largest subproblem on instance 40_16 takes 290 seconds with the default dominance algorithm, 293 seconds with the multidimensional divide-and-conquer algorithm, and 51 seconds with the two-stage algorithm. Overall, for default dominance the root node terminates in only 25 compared to 33 instances for the two-stage algorithm within the time limit.

7.1.2 Single- and Multi-Threaded Performance. By default BP uses the following schedule in a single thread: It allows up to one hour of B&P, but if Column Generation at the root node is not finished by then, it allows up to one more hour to solve the integer master problem to provide at least a best effort solution, which is very good in many cases. Still, this approach requires two separate timeouts, and it is not clear in the beginning if the full time will be needed on any of them. Recall that an integer solution is calculated at each node in the branching tree, therefore, the second phase is not needed if the root node is completed within the first hour.

We compare BP to the following multi-threaded versions in this section:

BP_M Multi-threaded MIP and LP solving: Since a considerable amount of time is spent solving the relaxed and integer master problem, we use 8 threads for Gurobi to speed it up.

BP+B MIP solving in the background: The main thread deals with the RMP and solving the sub-problem, while solving the integer problems is shifted to a background thread as described in Section 6.2. This has the advantage that no two separate phases are needed if the root node is not completed.

We compare the three variants of B&P and display the results in Table 3. Each entry shows the average of the results for 5 instances of the same size. Column *Opt. Gap* shows the optimality gap in percent. There was only

Table 3. B&P results grouped by size

	BP			BP_M			BP+B		
Size	Best	Opt. Gap	Time [s]	Best	Opt. Gap	Time [s]	Best	Opt. Gap	Time [s]
10	**14 709.2**	0.0	8.9	**14 709.2**	0.0	8.4	**14 709.2**	0.0	7.9
20	30 299.4	0.0	1430.1	30 299.4	0.0	1320.7	**30 294.6**	0.0	1410.4
30	**49 846.4**	0.4	3605.3	49 848.0	0.4	3604.5	**49 846.4**	0.4	3605.8
40	67 017.0	0.4	3612.4	**67 009.2**	0.3	3600.2	67 016.4	0.3	3604.1
50	84 338.8	0.4	3627.7	84 344.8	0.4	3603.1	**84 332.4**	0.4	3626.8
60	**99 754.6**	0.4	4058.5	**99 754.6**	0.4	3633.6	99 720.4	0.4	3602.3
70	118 337.6	-	5959.4	**118 285.6**	-	5147.8	118 472.8	-	3601.5
80	134 925.8	-	6244.4	**134 661.0**	-	5243.1	134 978.0	-	3601.5
90	150 292.8	-	7200.9	**150 172.0**	-	6833.1	150 804.6	-	3601.7
100	168 554.2	-	6624.6	**168 024.4**	-	6301.4	170 070.0	-	3601.4
150	273 629.2	-	7200.4	**266 465.2**	-	7200.5	280 936.4	-	3601.6
200	380 518.0	-	7202.1	**371 942.0**	-	7200.6	390 941.4	-	3603.7
250	520 063.4	-	7217.8	**498 122.4**	-	7208.5	534 937.2	-	3603.1

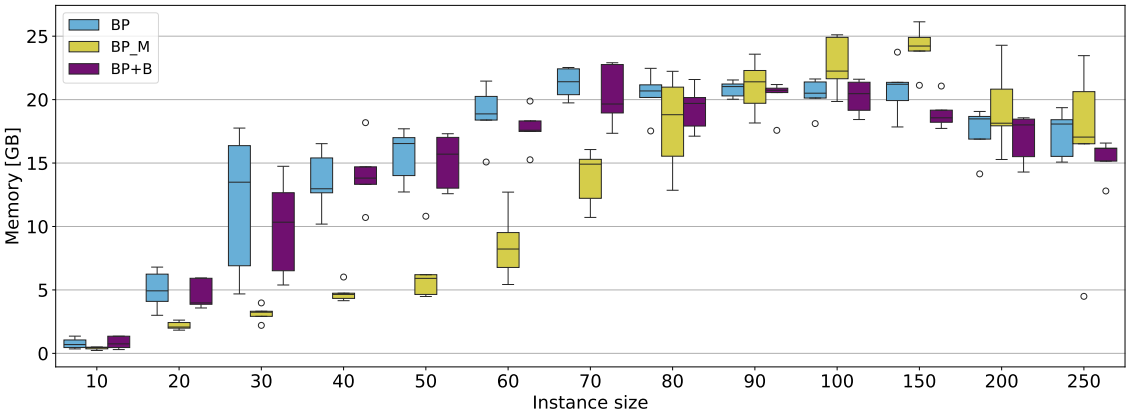


Fig. 8. Memory usage of different B&P variants across the 13 instance sizes

one run per method since B&P is deterministic (except for small differences due to thread timing with BP+B). Figure 8 shows the maximum memory use. We note that in most cases, especially regarding the smaller instances where B&P is most suited, there is no significant difference in the results, however, there are several relevant observations.

Starting from size 70, BP_M consistently provides the best results among the options, with especially large gaps starting at size 150. While there is no indication that using the extra threads helps with solving the RMP, this shows that solving the integer master problem benefits from the additional computational resources, especially for larger problems. This also shows in the runtimes, where the time for the second phase is often much shorter. While these are significant changes, they come at the cost of excessive additional resources (1 vs. 8) threads, and show most benefit on instances where B&P is outperformed by LNS anyway, making it not worth the extra

resources. Of interest regarding memory use is that this variant, compared to the others, uses significantly less memory for instances of size 30 to 70, but overtakes the other variants for larger instances.

The mild parallelization in BP+B again shows very similar results for the smaller instances, while the results for very large instances are slightly worse. Since the integer master problem in the background does not immediately include the most recent new columns, the final result is computed from a slightly inferior set of columns compared to the other methods. However, the great advantage is that the extra runtime of the second phase is not added on top, but in parallel. Therefore, for larger instances, the runtime is half of BP, while the computational effort is the same. Again, however, since the focus of B&P is on smaller instances, this effect is not so relevant in practice, and we use BP for the comparisons with other methods.

A key advantage of B&P is that for smaller instances (where the root node is completed), a bound on the solution quality can be provided. Results for all instances of size 10, and 4 out of 5 instances of size 20 are proven optimal, and very low bounds of up to 0.4 % are provided for instances of up to size 60 (for all instances up to size 50 and 4 out of 5 of size 60). While Lagrangean bounds can also be computed for larger instances, they are too weak and not reported here.

7.2 Large Neighborhood Search

To select the LNS parameters, we thoroughly analyzed the impact of different algorithmic components on a subset of instances from the benchmark set. Since algorithms performs similarly on same-sized instances, we selected one instance per size, skipping the smallest size that can be solved to optimality with BP in seconds. Therefore, we used 12 instances in this part of the evaluation; each result is the average of 5 runs.

For completeness, we compared our manually selected configuration against one produced by an automated parameter tuner. We used irace (version 4.3.c5b213a) with a tuning budget of 180 runs. However, we did not find any statistically significant difference between our configuration and the best configuration identified by irace.

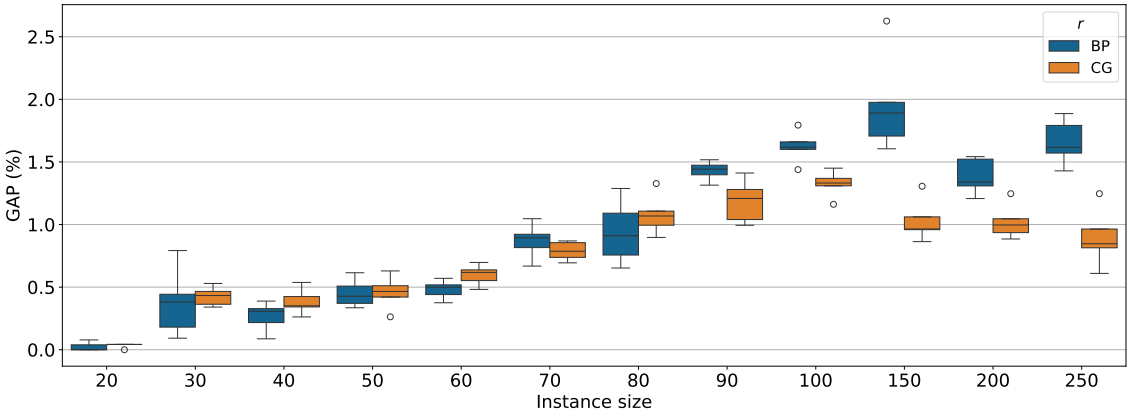
For the experiments in this section, we used an earlier setup with Java (OpenJDK 14.0.1) and CPLEX 12.10 to solve the master problem. The choice of solver does not impact the experimental results, as the behavior of the MIP solver only differs for larger problems, while the sub-problems in LNS are fairly small. In the following experiments, each box plot in a chart represents the results across 10 runs on the representative instance for each size. This allows to evaluate different parameter settings including the effect of instance size as well as the consistency of the results.

We investigate the following components:

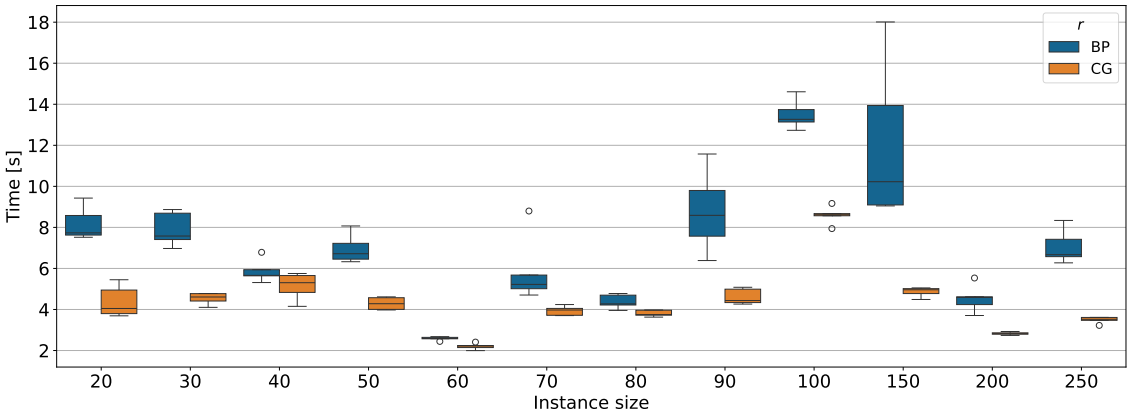
(1) The repair mechanism: r_{BP} or r_{CG}
(2) The initial destruction size k_0
(3) The maximum number of iterations without improvement $n_{\max}$
(4) The destroyer selection
(5) The role of adaptivity

7.2.1 Repair Operator Selection. For fixed $k_0 = 10$, $n_{\max} = 50$, and equal selection of all destroyers, we compared r_{BP} and r_{CG}. Initial experiments showed that the performance of destroy and repair is rather independent from each other, which enables separate evaluation. Each repair operation has a maximum budget of 5 min, but is expected to usually terminate much faster.

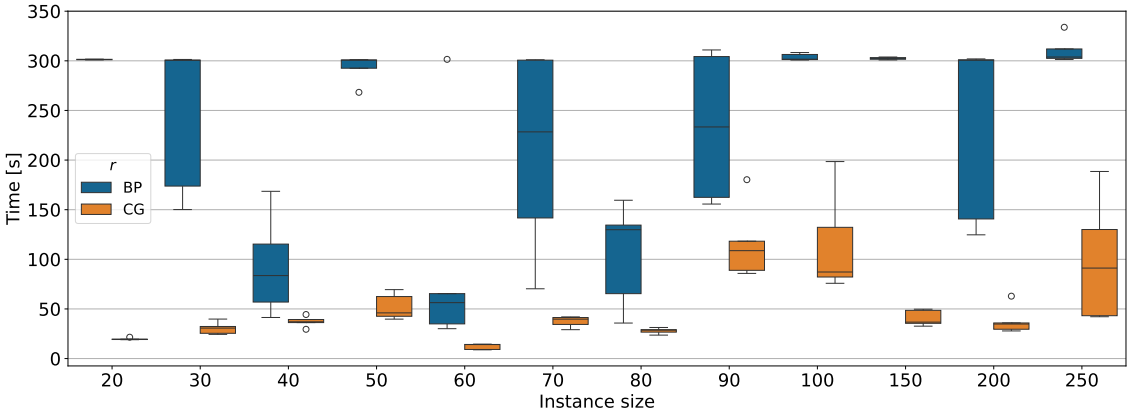
Figure 9a shows the GAP comparison across the 13 instance sizes. The performance is very similar for the smaller instances, but r_{CG} is clearly the better choice for larger instances, showing that the extra time spent on repairing in r_{BP} is not justified. Figure 9b and Figure 9c show that r_{BP} usually takes longer than r_{CG}. While the average time in both cases is under 10 s for most instances, Figure 9b shows that r_{CG} uses significantly lower average values. Figure 9c shows that r_{BP} often reaches the time budget of 5 min, while r_{CG} is mostly below 2 min, showing a much better worst case behavior.



(a) GAP for different repair operators



(b) Average CPU-time of different repair operators



(c) Maximum CPU-time of different repair operators

Fig. 9. Comparison of BP and CG as repair operators across different instance sizes

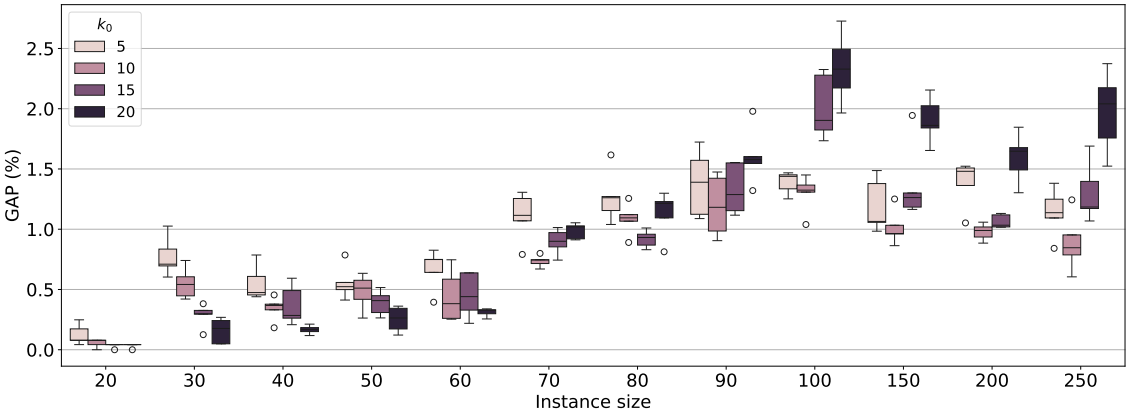


Fig. 10. GAP for different values of k_0 across different instance sizes

7.2.2 Initial Destruction Size. There are several options to set for the destroy operators. First, we investigated the initial destruction size k_0, fixing all other parameters. The size remained constant, r_{CG} and all destroyers are used, and $\rho_i = 1/3$ without adaptivity.

We tested sizes $k_0 \in \{5, 10, 15, 20\}$. Figure 10 shows the results. While $k_0 = 20$ performs slightly better for instances up to size 60 (the black box plots show lower gaps), it is outperformed on larger instances. Overall, $k_0 = 10$ works best for large instances which are the main focus of LNS, therefore, we fix $k_0 = 10$.

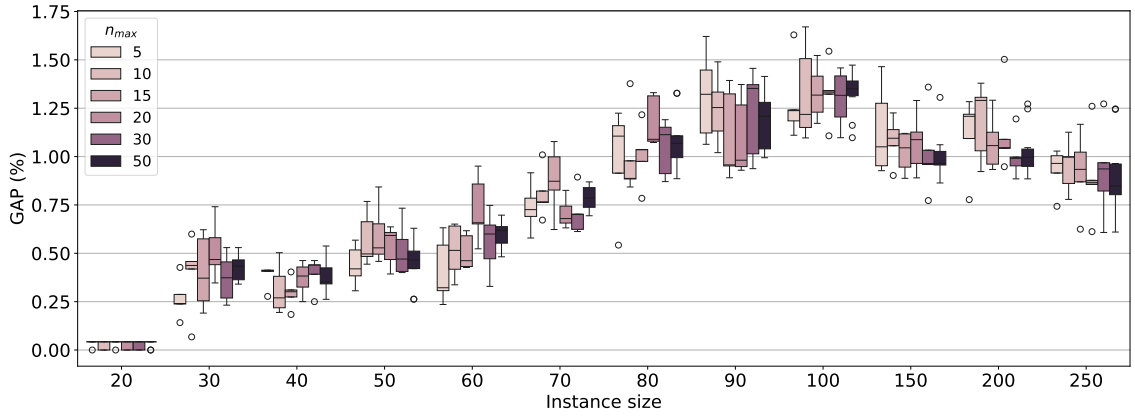
7.2.3 Number of Iterations Without Improvement. Next, we increase k by 1 every $n_{\max}$ iterations without improvement, until reaching the upper bound $k_{\max} = 20$ or finding an improvement. As soon as we find an improvement, the value of k is set back to the initial value $k_0 = 10$. In our experiments, we tested $n_{\max} \in \{5, 10, 15, 20, 30, 50\}$. Figure 11a shows a comparison between the GAPs. We could not detect any significant difference among them. Therefore, we decided to set $n_{\max} = 50$, since it still allows to increase the size when needed, but does not increase it very often. We tried starting with different values for k_0, but found similar results; the initial size matters more than the step size.

Figure 11b shows the impact of $n_{\max}$ on the number of iterations (r_{CG} calls). As expected, larger values of $n_{\max}$ imply more iterations (the darker the color is, the higher it is in the plot). We can explain that by the fact that there are less frequent size changes, hence, more iterations. Notably, for instances of size 150, 200, and 250, the size barely changes, as improvements are frequently found even with the initial size until timeout.

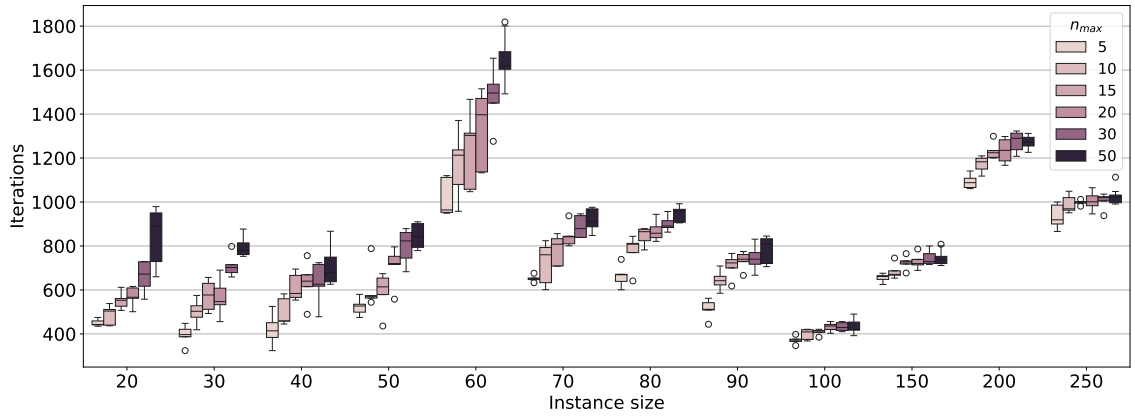
7.2.4 Destroyer Selection. To understand the impact of the destroy operators, we tested all the 7 possible combinations of them. Figure 12 shows that ω_{TR} has the biggest impact on the performance, since the dark blue box plots are almost always the lowest. It shows the best results on its own, with very similar results using it in any other combination, while all combinations without ω_{TR} show significantly worse performance, with higher variation among larger instances.

The advantage of selecting shifts that share tours is that the sub-problems are more likely to allow meaningful optimizations. This hypothesis is further backed by the *success rate* (percentage of iterations where the current solution could be improved) in Figure 13, which shows that, for instances with size 150, 200, and 250, using ω_{TR} results in the highest success rates: the dark blue box plots are entirely above all the other set of destroyers.

While only using ω_{TR} is the best choice, we further investigated several non-uniform weight distributions with high weights for ω_{TR}. Denoting the weights as $(\rho_{\omega_{\text{EU}}}, \rho_{\omega_{\text{EW}}}, \rho_{\omega_{\text{TR}}})$, we conducted experiments with $(5, 5, 90)$,



(a) GAP for different values of $n_{\max}$



(b) Number of iterations for different values of $n_{\max}$

Fig. 11. Comparison for different values of $n_{\max}$ across different instance sizes

$(10, 10, 80)$, and $(25, 25, 50)$. The first two were very similar to ω_{TR}, while $(25, 25, 50)$ started to get slightly worse. In summary, the best choice for the parameters is ω_{TR} as operator, an initial destruction size of $k_0 = 10$, and $n_{\max} = 50$ iterations without improvements.

7.2.5 Adaptivity. To investigate the impact of adaptivity, we conducted experiments by changing the parameter λ in Equation (36), considering all three destroy operators. We tested values $\lambda \in \left\{\frac{1}{3}, \frac{1}{2}, \frac{2}{3}\right\}$. Figure 14 shows that the box plot related to ω_{TR} is lower than the other adaptive variants across instances with size larger than 60. Thus, the adaptivity does not improve the average GAP compared to using only ω_{TR}, and different values of λ do not show significant difference.

7.3 Integration of CG and LNS

In this section, we evaluate and compare seven LNS variants with different levels of Column Generation integration on all 65 instances, summarized in Table 4.

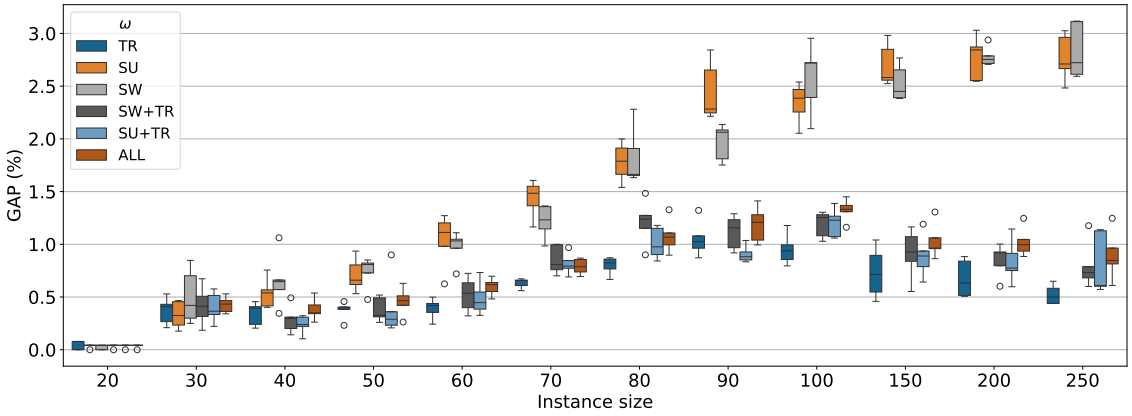


Fig. 12. GAP for different subsets of destroyers across different instance sizes

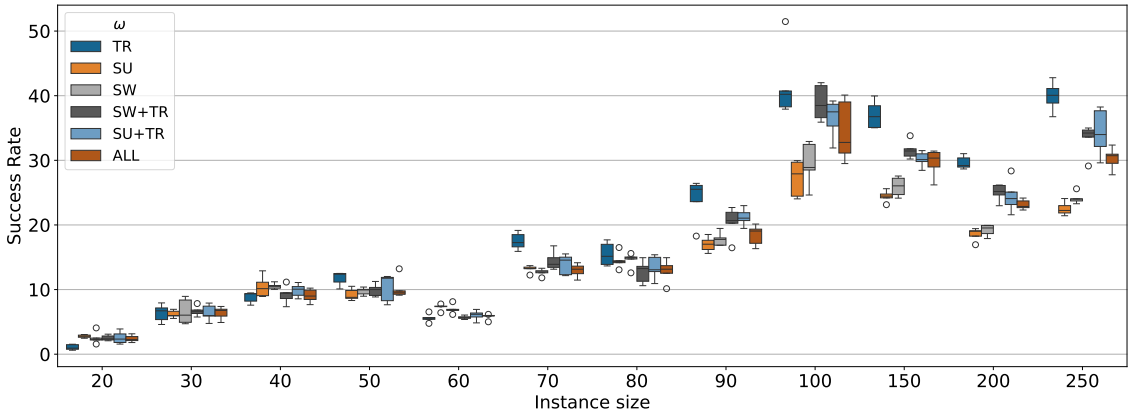


Fig. 13. Success rate for different subsets of destroyers across different instance sizes

The +R variants use the stored columns to initialize each sub-problem according to Equation (38). The +B options apply a background MIP solver according to Section 6.2 on the set of stored columns $\hat{\mathcal{S}}$ with a timeout $t_{bg} = 1$ min for every iteration. For each combination, two different choices for the selection of columns to store (function `select`) are evaluated according to Section 6.3. The first option (B) is to store only the best columns S_i^* for each sub-problem, the other option (F) to store the full set $\mathcal{S}_i$.

7.3.1 Impact on LNS Metrics. Figure 15 shows the number of iterations, average repairing time, and success rate for each of the methods, grouped by size. The trend by size is clear: Smaller instances allow more iterations, since they are faster. For smaller instances, the success rate is low, indicating fast convergence, while for larger instances success rate is high, indicating less time is spent in local optima.

Between different variants of LNS, differences are small, but with several notable patterns. First, LNS and both +B variants show very similar results in all three metrics, as expected since the work in the background thread should not have significant impact on the main thread.

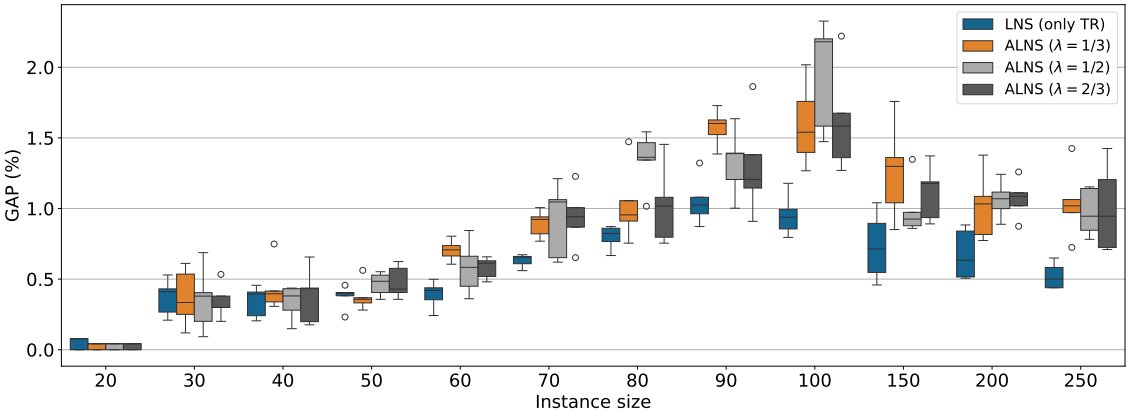


Fig. 14. GAP for adaptive and static LNS across different instance sizes

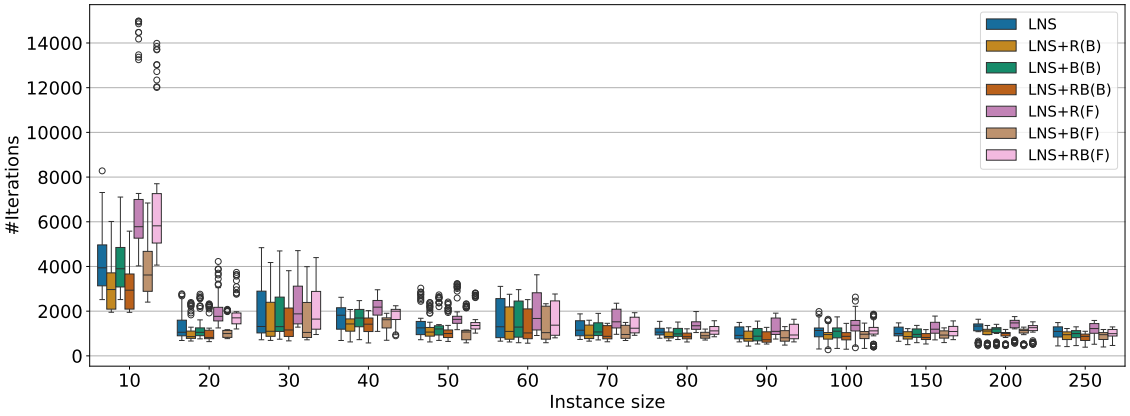
Table 4. LNS variants

Name	Column Reuse	Background Solver	`select`
LNS	no	no	-
LNS+R(B)	yes	no	best
LNS+R(F)	yes	no	full
LNS+B(B)	no	yes	best
LNS+B(F)	no	yes	full
LNS+RB(B)	yes	yes	best
LNS+RB(F)	yes	yes	full

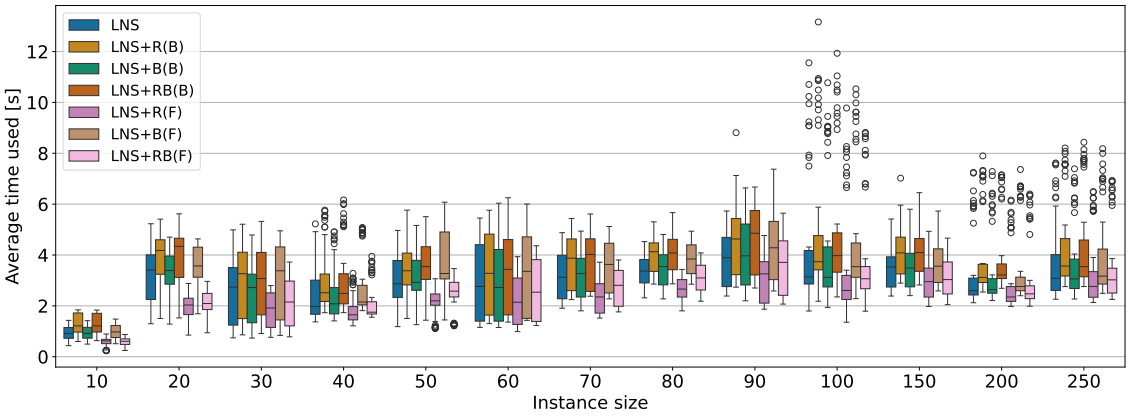
Next, both LNS+R(B) and LNS+RB(B) show fewer and slower iterations, while both LNS+R(F) and LNS+RB(F) show more and faster iterations. This is interesting since column reuse (+R) is supposed to speed up solving the sub-problems. This only seems to work when adding the full set of columns (F), while only adding the best columns (B) actually seems to generate overhead instead of speeding up the search. This might be connected with a similar phenomenon in full B&P. Providing a good initial solution does not improve compared to starting from the trivial initial solution assigning one leg per shift. The reason seems to be that in order to finish column generation at the root node, still a large number of slightly suboptimal columns needs to be produced. Therefore, our assumption is that selecting the subset of stored columns when only few columns are stored (B) runs into the same issue, while the extra overhead of storing all columns (F) is more than compensated by the benefit of having a large set of good columns to start from.

7.3.2 Performance. Figure 16 shows the GAP to the best known solution for the different variants. While metric differences were small, GAP differences are clear. LNS+B(F) and LNS+RB(F) significantly outperform all other methods, which show similar performance among each other. This indicates that adding the background thread (+B) with the full set of columns (F) is the key combination to improve performance.

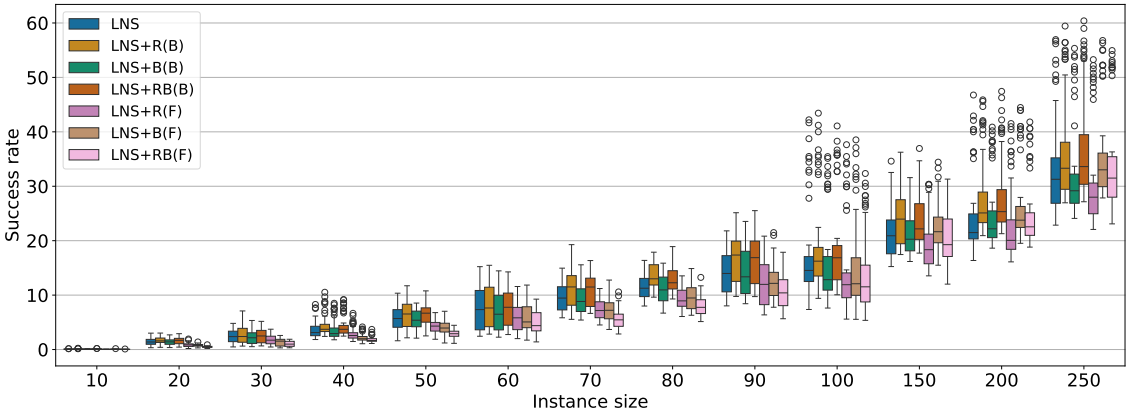
The shape of the graph shows that this improvement is most beneficial for medium to large, but not very large instances. For small instances, all methods perform well, with more notable distinctions starting at size 30.



(a) Number of iterations



(b) Average repairing time



(c) Success rate

Fig. 15. Metrics of different LNS integrations across different instance sizes

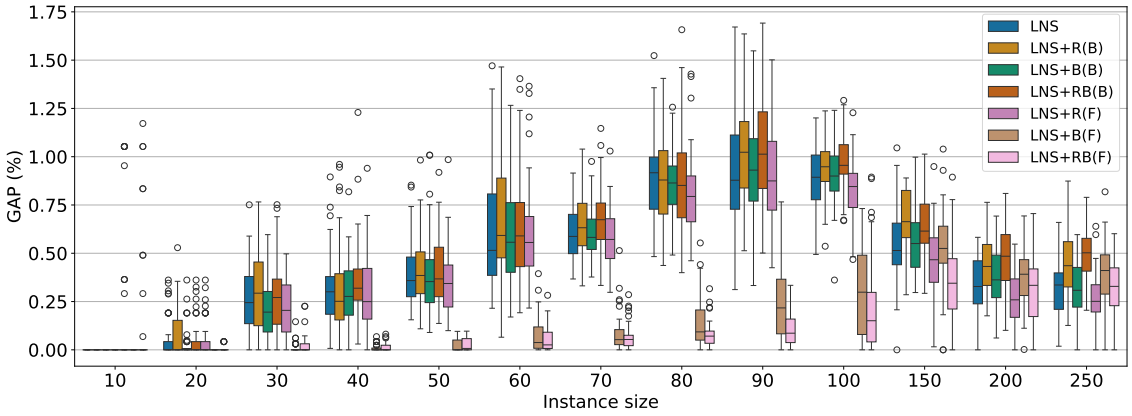


Fig. 16. GAP for different LNS integrations across different instance sizes

Separation grows larger up to around size 90, while for larger instances the gap between methods shrinks, and methods perform similarly for sizes 200 and 250.

Notably, for up to size 90, LNS+B(F) and LNS+RB(F) also show very low standard deviations, making them more stable than the other methods. LNS+B(F) starts to degrade around size 80, while LNS+RB(F) shows further benefits regarding deviations and also slightly better results than LNS+B(F) for sizes 80 to 150.

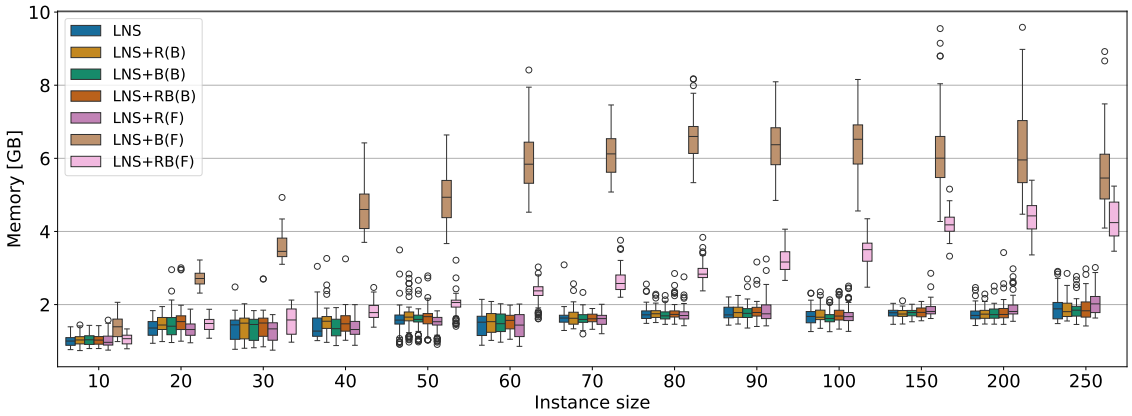
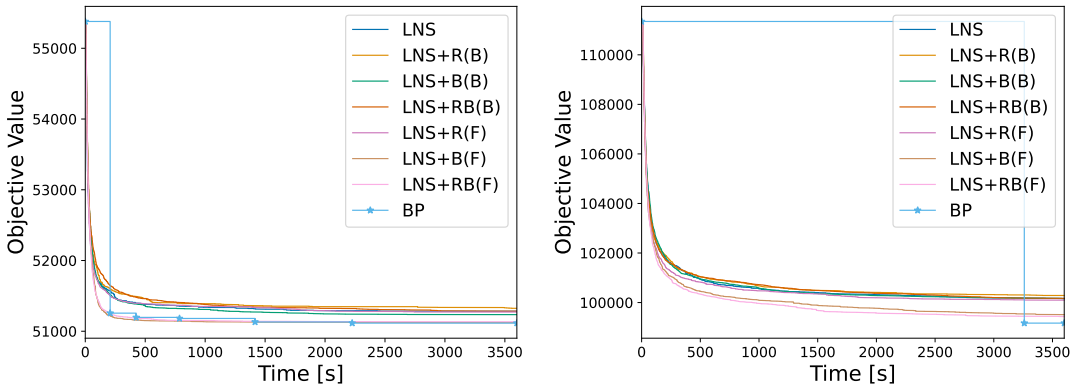


Fig. 17. Memory usage of different LNS integrations across different instance sizes

Figure 17 shows the memory use of the different LNS versions. All versions that do not combine the background thread (+B) with full storage (F) show very similar memory usage. The pure storage, even of the full set of columns, is not significant, since the storage using bit sets in the implementation is very light-weight. However, there is significantly larger memory usage for LNS+B(F) and LNS+RB(F), especially for larger instances, showing that using the background thread with a larger number of columns is what needs extra memory.

While the quality of the results showed the large impact of the background thread and only minor improvements for column reuse, in comparison LNS+RB(F) uses significantly less memory than LNS+B(F). Upon closer investigation,

the number of columns in the background thread grows much slower in LNS+RB(F) than in LNS+B(F). For example, in the instance realistic_100_49, the average number of columns differs by a factor of more than 3, from 2.0×10^6 in LNS+B(F) to 6.7×10^5 in LNS+RB(F). The suspected reason is that reusing the columns prevents the sub-problems from generating a large number of distinct suboptimal columns which would clutter the background thread, instead reusing the same ones again. This is also consistent with the slight performance benefit of LNS+B(F) regarding the GAP for sizes 80 to 150. While LNS+B(F) already has the background thread cluttered with too many sub-optimal columns, LNS+RB(F) avoids this bottleneck for longer.



(a) Realistic_30_12

(b) Realistic_60_28

(c) Realistic_100_49

(d) Realistic_250_64

Fig. 18. Convergence plots

7.3.3 Convergence Plots. To analyze the behavior of the different methods over their runtime, Figure 18 shows the convergence plots for four instances of different sizes to highlight different behaviors. The trajectories of the LNS versions are the average trajectories over 10 runs to avoid showing outliers.

The first three instances all show the dominance of LNS+B(F) and LNS+RB(F) over the other LNS methods. The trajectories of these methods separate from the others very early in the search, and consistently stay ahead until the end.

Table 5. Results for different solution methods grouped by size

	BP	LNS		LNS+RB(F)		CMSA	
Size		Min	Avg	Min	Avg	Min	Avg
10	**14 709.2**	14 709.2	**14 709.2**	14 709.2	14 728.2	14 867.4	14 879.7
20	30 299.4	30 294.6	30 312.3	30 294.6	**30 295.9**	30 695.8	30 745.9
30	**49 846.4**	49 867.0	49 966.4	49 841.0	49 851.2	50 731.4	50 817.2
40	67 017.0	67 023.8	67 165.7	66 957.6	**66 966.7**	68 394.8	68 499.9
50	84 338.8	84 415.4	84 583.7	84 251.4	**84 272.2**	86 219.0	86 389.2
60	99 754.6	100 006.4	100 246.4	99 673.6	**99 704.7**	102 596.2	102 822.9
70	118 337.6	118 512.2	118 698.9	117 991.2	**118 051.3**	120 935.6	121 141.9
80	134 925.8	134 827.6	135 178.8	133 983.4	**134 081.6**	138 406.8	138 760.3
90	150 292.8	150 282.6	150 721.6	149 340.2	**149 494.4**	154 692.6	155 078.3
100	168 554.2	166 397.4	166 811.5	165 331.4	**165 698.9**	171 159.4	171 786.7
150	273 629.2	254 742.2	255 449.1	254 195.8	**255 017.6**	263 079.2	263 387.7
200	380 518.0	337 479.0	338 226.9	337 221.8	**338 161.2**	348 608.6	349 017.0
250	520 063.4	426 908.4	427 866.7	426 739.4	**427 804.0**	438 811.4	439 234.5

Figure 18a shows the behavior of BP for a smaller instance. While the results increase in steps (at the end of some nodes in the branching tree), the overall result and trajectory are comparable to the best LNS versions.

Figure 18b shows the behavior of BP for a mid-sized instance. While it can outperform the LNS versions in the end, it provides no solution for a long time, leading to a much better any-time behavior of LNS.

For the larger instances (Figure 18c), BP only provides a solution with extra runtime or by using the background thread (BP+B).

Figure 18d shows the behavior of LNS for a very large instance. Here, the versions perform similarly, and the trajectories flatten out less, indicating that LNS is still improving regularly without hitting too many local optima, while the problem solved in the background thread gets larger and therefore harder to solve.

7.4 Comparison of Different Methods

Table 5 shows the comparison of the final versions of BP and LNS, and the best performing integration LNS+RB(F), as well as the previous state of the art CMSA (Rosati et al. 2023). CMSA outperforms all earlier meta-heuristics on this problem. Note that CMSA was run on a more powerful machine than ours: an AMD Ryzen Threadripper PRO 3975WX (32 cores, with a base clock frequency of 3.5 GHz, 64 GB of RAM). For the stochastic methods, the minimum and average across 10 runs are presented, for BP the result of one run. The table shows minimum and average per size, with each row averaging the 5 instances of this particular size. Detailed results for these and other tested methods per instance are available in the appendix. The best performing method is highlighted in bold. Figure 19 shows the gaps for these methods (average is used for stochastic methods).

The results indicate that BP is the best method to use for small instances, as it can solve very small instances to optimality in a few seconds, and provides high-quality solutions with known low gaps to the optimum for up to size 30. BP starts to struggle for large instances.

While BP still outperforms CMSA for instances of size up to 90, starting with size 40, the integrated method LNS+RB(F) already outperforms all other methods, and consistently provides the best results across all further sizes, making it the new state of the art for this problem.

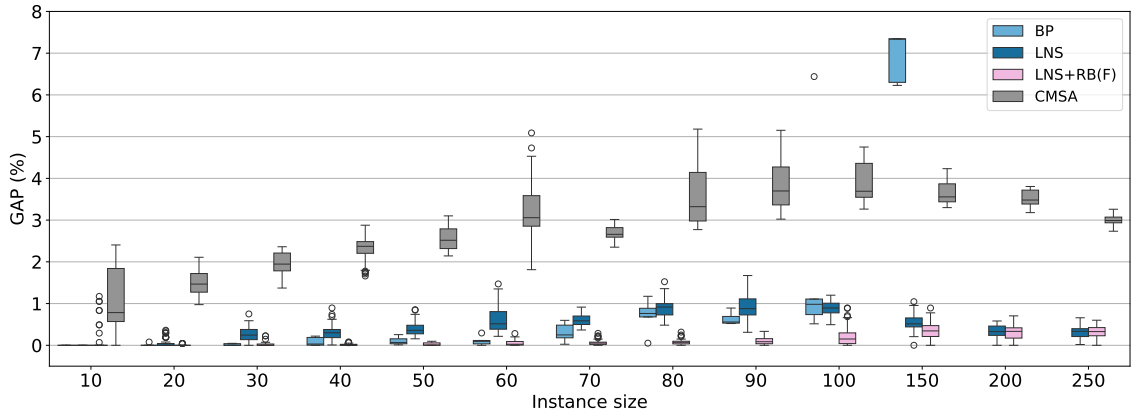


Fig. 19. Comparing GAP across instance sizes for BP, LNS variants and CMSA.

Since LNS+RB(F) uses the background thread, and therefore more resources both regarding computation and memory, the most efficient method is LNS, which only needs one thread, and uses very moderate memory consumption. Still, in a very competitive environment, small improvements like those gained by LNS+RB(F) translate to large savings over time, and the moderate additional resource consumption is worth the additional benefit in practice.

7.4.1 Statistical Significance. To add a well-founded view of the results, we perform a statistical analysis. For this purpose, we use the tool SCMAMP (Calvo 2025) ran with R (version 4.2.2). For our comparison, we consider BP, all our variants of LNS, as well as other results from the literature: CMSA (Rosati et al. 2023), Simulated Annealing, Hill Climbing (Kletzander and Musliu 2020), and Tabu Search (Kletzander, Mazzoli, et al. 2022). The analysis proceeds in two steps: the *omnibus* test, and *post-hoc* test.

In the omnibus test, we check whether at least one of the algorithms performs differently than the others. To do that, following the guidelines of Calvo (Calvo and Santafé 2016), we use the Friedman test with Iman and Davenport extension. We formulate the null hypothesis H_0: *for every instance, the average objective function values are identical on all the algorithms.* We obtained a corrected statistic of $F = 167.04$, with $(12, 768)$ degrees of freedom. With a significance level of $\alpha = 0.05$, the test rejects the null hypothesis with a p-value smaller than 1×10^{-15}. Therefore, we conclude that there is strong statistical evidence that at least one algorithm performs differently than the rest. Thus, we conduct a post-hoc test to detect differences by pairs.

In the post-hoc test, we conduct an all pair-wise comparison using the Shaffer's static procedure. Figure 20 shows the *Critical Difference* (CD) plot Figure 20 at significance level of $\alpha = 0.05$. Each considered algorithm is placed on the horizontal axis according to its average ranking for the instances (lower is better). Algorithms within the critical difference threshold (2.2697) are statistically equivalent. In the CD plot, a bold horizontal bar marks equivalent algorithms.

The results show three different performance tiers:

(1) *Top Tier*: LNS+RB(F) and LNS+B(F) form a statistically equivalent group that has the best performance against all the other methods.
(2) *Middle Tier*: The remaining LNS variants, and BP show intermediate performance, outperforming CMSA and traditional metaheuristics.

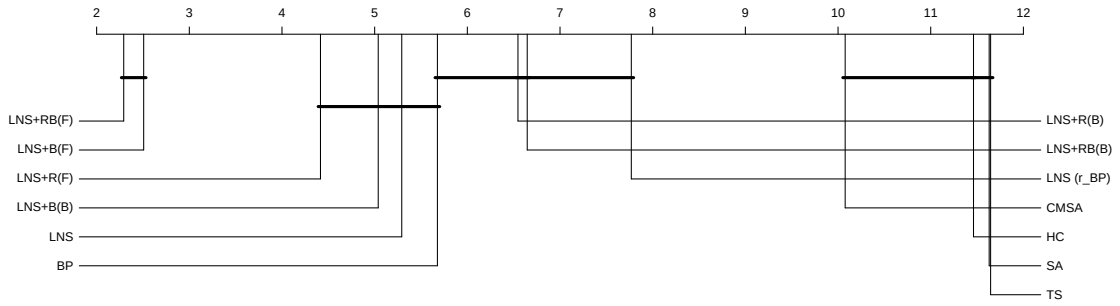


Fig. 20. Critical Difference plot

(3) *Low Tier*: CMSA, Hill Climbing, Simulated Annealing and Tabu Search are statistically indistinguishable from each other.

8 Conclusions

In this article we provided a comprehensive study of Branch and Price (B&P) and Large Neighborhood Search (LNS) for the Bus Driver Scheduling Problem (BDSP). We defined a high-dimensional Resource Constrained Shortest Path Problem (RCSPP) as the sub-problem for B&P that is able to represent the complex break constraints in our model, and introduced several novel adaptations to solve this sub-problem efficiently, including splitting the sub-problem, exponential arc throttling, and the usage of k-d trees. With this method we are able to solve small problems to optimality in seconds, and provide results with known optimality gaps of less than one percent for mid-sized instances, constituting the state of the art for these instances.

For LNS, we introduced the novel destroy operator ω_{TR} that exploits the structure of the problem, and thoroughly evaluated several different design choices for both the destroy and repair operators, as well as the use of adaptivity. The results scale well even to very large instances, and can outperform previous meta-heuristic solution methods.

Finally, we proposed a novel tight integration between LNS and Column Generation (CG), where the best version of the integration LNS+RB(F) stores all columns from each sub-problem in LNS, reusing them for future sub-problems, and solving the integer master problem with all columns in a background thread. The evaluation showed that the background thread helps to significantly improve the results of LNS, especially for mid- to large-sized instances, while the column reuse is very beneficial to reduce the memory usage of the background thread. Overall, LNS+RB(F) is the new state of the art for instances of medium to large size.

While this article evaluates the methods on this complex version of BDSP, both the concepts introduced to solve high-dimensional RCSPPs, and the integration of LNS and CG are general and can be applied to other scenarios, or to other optimization problems.

As future work, we would like to apply the new concepts to different optimization problems, both Bus Driver Scheduling with different rule sets, as well as entirely different problems. For the BDSP, future work could focus especially of the very large instances, where the benefits of LNS+RB(F) seem to degrade, and investigate further improvements of the integration like eliminating sub-optimal columns from the background thread, or more choices between accepting the best or full set of columns.

Acknowledgments

This project is partially funded by the Doctoral Program Vienna Graduate School on Computational Optimization (VGSCO), Austrian Science Foundation (FWF), under grant No W1260-N35. In addition, this research was funded

in part by FWF Cluster of Excellence [10.55776/COE12]. The financial support by the Austrian Federal Ministry of Labour and Economy, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged. The project was also supported by a NSF Leap-HI award NSF-1854684.

References

E. Balas and M. W. Padberg. 1976. "Set partitioning: A survey." *SIAM review*, 18, 4, 710–760.

C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. Savelsbergh, and P. H. Vance. 1998. "Branch-and-price: Column generation for solving huge integer programs." *Operations research*, 46, 3, 316–329.

A. Beer, J. Gaertner, N. Musliu, W. Schafhauser, and W. Slany. 2008. "Scheduling Breaks in Shift Plans for Call Centers." English. In: *Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling*, 1–17.

A. Beer, J. Gärtner, N. Musliu, W. Schafhauser, and W. Slany. Mar. 2010. "An AI-Based Break-Scheduling System for Supervisory Personnel." *IEEE Intelligent Systems*, 25, 2, (Mar. 2010), 60–73.

J. L. Bentley. 1980. "Multidimensional divide-and-conquer." *Communications of the ACM*, 23, 4, 214–229.

B. Calvo. 2025. *scmamp: Statistical Comparison of Multiple Algorithms in Multiple Problems*. https://github.com/b0rxa/scmamp/tree/3cf4d8b9759769cdf20771afa0efc33a5265c7f9. Accessed: 2025-01-30. (2025).

B. Calvo and G. Santafé. 2016. "scmamp: Statistical Comparison of Multiple Algorithms in Multiple Problems." *The R Journal*, 8, 1, 248–256. doi:10.32614/RJ-2016-017.

L. d. Carmo Martins and G. P. Silva. 2019. "An Adaptive Large Neighborhood Search Heuristic to Solve the Crew Scheduling Problem." In: *Smart and Digital Cities*. Springer, 45–64. doi:10.1007/978-3-030-12255-34.

S. Chen, Y. Shen, X. Su, and H. Chen. Apr. 2013. "A Crew Scheduling with Chinese Meal Break Rules." en. *Journal of Transportation Systems Engineering and Information Technology*, 13, 2, (Apr. 2013), 90–95.

W.-M. Chen, H.-K. Hwang, and T.-H. Tsai. 2012. "Maxima-finding algorithms for multidimensional samples: A two-phase approach." *Computational Geometry*, 45, 1-2, 33–53.

A. A. Constantino, C. F. de Mendonca, S. A. de Araujo, D. Landa-Silva, R. Calvi, and A. F. dos Santos. 2017. "Solving a large real-world bus driver scheduling problem with a multi-assignment based heuristic algorithm." *Journal of Universal Computer Science*, 23, 5. https://nottingham-repository.worktribe.com/output/862152.

R. De Leone, P. Festa, and E. Marchitto. Aug. 2011. "A Bus Driver Scheduling Problem: a new mathematical model and a GRASP approximate solution." en. *Journal of Heuristics*, 17, 4, (Aug. 2011), 441–466.

M. Desrochers and F. Soumis. Feb. 1989. "A Column Generation Approach to the Urban Transit Crew Scheduling Problem." en. *Transportation Science*, 23, 1, (Feb. 1989), 1–13.

A. Ernst, H. Jiang, M. Krishnamoorthy, and D. Sier. Feb. 2004. "Staff scheduling and rostering: A review of applications, methods and models." en. *European Journal of Operational Research*, 153, 1, (Feb. 2004), 3–27.

J. K. Fichte, T. Geibinger, M. Hecher, and M. Schlögel. 2024. "Parallel empirical evaluations: Resilience despite concurrency." In: *Proceedings of the AAAI Conference on Artificial Intelligence* 8. Vol. 38, 8004–8012.

N. Frohner, E. Mugdan, L. Kletzander, and N. Musliu. 2024. "A Decision Support System Prototype for Automated Bus Driver Scheduling." In: *Proceedings of the 14th International Conference on the Practice and Theory of Automated Timetabling, PATAT 2024*, 259–262.

J. D. Hunter. 2007. "Matplotlib: A 2D graphics environment." *Computing in Science & Engineering*, 9, 3, 90–95. doi:10.1109/MCSE.2007.55.

O. Ibarra-Rojas, F. Delgado, R. Giesen, and J. Muñoz. July 2015. "Planning, operation, and control of bus transport systems: A literature review." en. *Transportation Research Part B: Methodological*, 77, (July 2015), 38–75.

S. Irnich. 2008. "Resource extension functions: Properties, inversion, and generalization to segments." *OR Spectrum*, 30, 1, 113–148.

S. Irnich and G. Desaulniers. 2005. "Shortest path problems with resource constraints." In: *Column generation*. Springer, 33–65.

L. Kletzander, T. M. Mazzoli, and N. Musliu. July 2022. "Metaheuristic algorithms for the bus driver scheduling problem with complex break constraints." In: *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, (July 2022). doi:https://dl.acm.org/doi/10.1145/3512290.3528876.

L. Kletzander and N. Musliu. 2023a. "Dynamic weight setting for personnel scheduling with many objectives." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 33, 509–517.

L. Kletzander and N. Musliu. June 2023b. "Large-State Reinforcement Learning for Hyper-Heuristics." *Proceedings of the AAAI Conference on Artificial Intelligence*, 37, 10, (June 2023), 12444–12452. doi:10.1609/aaai.v37i10.26466.

L. Kletzander and N. Musliu. June 2020. "Solving Large Real-Life Bus Driver Scheduling Problems with Complex Break Constraints." *Proceedings of the International Conference on Automated Planning and Scheduling*, 30, 1, (June 2020), 421–429. https://ojs.aaai.org/index.php/ICAPS/article/view/6688.

L. Kletzander, N. Musliu, and P. Van Hentenryck. May 2021. "Branch and Price for Bus Driver Scheduling with Complex Break Constraints." *Proceedings of the AAAI Conference on Artificial Intelligence*, 35, 13, (May 2021), 11853–11861. https://ojs.aaai.org/index.php/AAAI/article/view/17408.

J. Li and R. S. Kwan. June 2003. "A fuzzy genetic algorithm for driver scheduling." en. *European Journal of Operational Research*, 147, 2, (June 2003), 334–344.

D.-Y. Lin and C.-L. Hsu. Dec. 2016. "A column generation algorithm for the bus driver scheduling problem: Bus Driver Scheduling Problem." en. *Journal of Advanced Transportation*, 50, 8, (Dec. 2016), 1598–1615.

H. R. Lourenço, J. P. Paixão, and R. Portugal. Aug. 2001. "Multiobjective Metaheuristics for the Bus Driver Scheduling Problem." en. *Transportation Science*, 35, 3, (Aug. 2001), 331–343.

S. Martello and P. Toth. Jan. 1986. "A heuristic approach to the bus driver scheduling problem." en. *European Journal of Operational Research*, 24, 1, (Jan. 1986), 106–117.

T. M. Mazzoli, L. Kletzander, P. Van Hentenryck, and N. Musliu. 2024. "Investigating Large Neighbourhood Search for Bus Driver Scheduling." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 34, 360–368.

R. Portugal, H. R. Lourenço, and J. P. Paixão. June 2009. "Driver scheduling problem modelling." en. *Public Transport*, 1, 2, (June 2009), 103–120.

A. Respício, M. Moz, and M. Vaz Pato. 2013. "Enhanced genetic algorithms for a bi-objective bus driver rostering problem: a computational study." *International Transactions in Operational Research*, 20, 4, 443–470.

S. Ropke and D. Pisinger. Nov. 2006. "An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows." *Transportation Science*, 40, 4, (Nov. 2006), 455–472. doi:10.1287/trsc.1050.0135.

R. M. Rosati, L. Kletzander, C. Blum, N. Musliu, and A. Schaerf. 2023. "Construct, Merge, Solve and Adapt Applied to a Bus Driver Scheduling Problem with Complex Break Constraints." In: *AIxIA 2022 – Advances in Artificial Intelligence*. Springer International Publishing, 254–267. doi:10.1007/978-3-031-27181-618.

P. Shaw. 1998. "Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems." In: *Principles and Practice of Constraint Programming — CP98*. Springer Berlin Heidelberg, 417–431. doi:10.1007/3-540-49481-230.

Y. Shen and R. S. K. Kwan. 2001. "Tabu Search for Driver Scheduling." In: *Computer-Aided Scheduling of Public Transport*. Vol. 505. Ed. by G. Fandel, W. Trockel, C. D. Aliprantis, D. Kovenock, S. Voß, and J. R. Daduna. Springer Berlin Heidelberg, Berlin, Heidelberg, 121–135. Retrieved Nov. 20, 2019 from.

B. M. Smith and A. Wren. 1988. "A bus crew scheduling system using a set covering formulation." *Transportation Research Part A: General*, 22, 2, 97–108. doi:https://doi.org/10.1016/0191-2607(88)90022-2.

A. Tóth and M. Krész. June 2013. "An efficient solution approach for real-world driver scheduling problems in urban bus transportation." en. *Central European Journal of Operations Research*, 21, S1, (June 2013), 75–94.

J. Van den Bergh, J. Beliën, P. De Bruecker, E. Demeulemeester, and L. De Boeck. May 2013. "Personnel scheduling: A literature review." en. *European Journal of Operational Research*, 226, 3, (May 2013), 367–385.

M. L. Waskom. 2021. "seaborn: statistical data visualization." *Journal of Open Source Software*, 6, 60, 3021. doi:10.21105/joss.03021.

M. Widl and N. Musliu. June 2014. "The break scheduling problem: complexity results and practical algorithms." en. *Memetic Computing*, 6, 2, (June 2014), 97–112.

WKO. 2019. *Kollektivvertrag Autobusbetriebe, Arbeiter/innen / Angestellte*. https://www.wko.at/service/kollektivvertrag/kv-private-autobusbetriebe-2019.html. Accessed: 2024-04-10. (2019).

A. Wren. 2004. *Scheduling vehicles and their drivers-forty years' experience*. Tech. rep. University of Leed, 27–40.

A. Wren and J.-M. Rousseau. 1995. "Bus Driver Scheduling — An Overview." In: *Computer-Aided Transit Scheduling*. Vol. 430. Ed. by G. Fandel, W. Trockel, J. R. Daduna, I. Branco, and J. M. P. Paixão. Springer Berlin Heidelberg, Berlin, Heidelberg, 173–187.

Received 01 May 2025; accepted 06 March 2026