Integrating Train Driver Scheduling into Local Search for the Train Unit Shunting Problem

Master's Thesis

Author: Kasper Vaessen

TU Delft

Integrating Train Driver Scheduling into Local Search for the Train Unit Shunting Problem

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Kasper Vaessen

Algorithmics Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Nederlandse Spoorwegen (NS)
Laan van Puntenburg 100
Utrecht, the Netherlands
www.ns.nl

Cover picture: Schréder.

Integrating Train Driver Scheduling into Local Search for the Train Unit Shunting Problem

Author: Kasper Vaessen
Student id: 4994760

Abstract

Automatic planning for railway shunting yards in the Netherlands is a challenging problem, as these yards form a critical link between rolling stock circulation and the timetable. Trains must be decoupled, coupled, serviced, and departed within tight time windows, while infrastructure and train drivers are limited. The resulting planning problem is large-scale, highly constrained, and characterized by strong interdependencies between operational and driver decisions. In the current state-of-the-art approach, driver scheduling is handled by a heuristic procedure embedded within a local search algorithm. While efficient, this restricts the decision space and may yield suboptimal plans.

This thesis investigates how integrating driver-scheduling decisions into the main local search affects solution quality and search performance. Several integration mechanisms are presented, with increasing levels of integration of driver scheduling decisions. Computational experiments on realistic instances show that integrating driver decisions into the local search consistently improves performance compared to the state-of-the-art approach. Under small time budgets, the greatest improvements are obtained when only part of the driver decision space is integrated, allowing the search to correct harmful heuristic decisions while keeping the search space manageable. With larger time budgets, more integrated methods catch up, indicating that the richer decision space can be exploited when sufficient time is available.

To manage the increase in search space complexity, a novel two-stage search scheme is introduced in which the algorithm first optimizes while keeping driver decisions heuristically decided, and subsequently activates explicit driver-related decision variables. Experiments show that dividing the available computation time between these two stages yields better results than allocating the entire time budget to either stage. This demonstrates that progressive enlargement of the decision space can effectively balance model expressiveness and computational tractability.

Thesis Committee:

Committee chair:	Prof. Dr. M.M. de Weerdt, Faculty EEMCS, TU Delft
Committee member:	Prof. Dr. R.M.P. Goverde, Faculty CITG, TU Delft
Advisor:	Ir. A, Berger, Faculty EEMCS, TU Delft
External Advisor:	Ir. S. Hesselmans, NS

Preface

This thesis marks the final chapter of my master's degree in Computer Science at Delft University of Technology. Throughout my study period in Delft, I realised I love to puzzle. To take something that does not work, identify its problems, and use logic to fix it. I also realised that I want to use my skills to address problems with societal impact. In this thesis, I was able to combine both of these passions by improving planning methods for railway shunting operations, therefore contributing to more efficient use of railway infrastructure and supporting reliable and sustainable public transport.

I would like to thank some people, without whom this thesis would not have been possible. First of all, I would like to thank Professor Mathijs de Weerdt for his invaluable feedback and for forcing me to see the bigger picture or an alternative perspective on the material when I needed it. I would also like to thank Sam Hesselmans and Aaron Berger for their daily supervision and for always being available for my questions. Their help really enabled me to quickly get knowledgeable in the topic.

I would also like to thank NS and team HIP for the opportunity to work on this project. Working on a project I knew had a real, practical impact on a societal problem motivated me throughout the 8 months of my thesis.

Finally, I would like to thank Roos and my friends. They were a listening ear when I needed to get something off my chest, provided me with much-needed breaks, and supported me throughout the process.

Kasper Vaessen
Delft, the Netherlands
April, 2026

Contents

Chapter 1

Introduction

The Dutch railway network is experiencing capacity pressure due to rapid growth in train traffic. In 2023, the network had a usage of 70 trains per kilometer of track per day, approximately twice the European average [4]. ProRail has warned that the network is nearing its limits with a potential capacity shortfall in the coming decade if train frequencies continue to rise [24, 29].

A bottleneck in current operations is the limited capacity of train shunting yards, where trains are parked and serviced between their daily runs. During peak hours on regular operating days, most often at night, shunting yards operate close to their capacity [3]. At special times of peak demand, such as during a strike, the yards can reach full occupancy. For example, during the 2020 service reduction due to COVID-19, NS reported that all the shunting yards were operating at or above capacity [2]. Since adding new tracks is often unfeasible in the densely populated Netherlands [29], efficient usage of the available capacity is required.

One of the key planning problems arising from this context is the Train Unit Shunting Problem with Service and Personnel Scheduling (TUSPwSPS) [38]. This problem states: given a set of passenger train units that arrive at a shunting yard, how can we park and service them such that they are ready for their scheduled departures, with all servicing completed and respecting resource limitations? A feasible solution to TUSPwSPS consists of a conflict-free schedule of all activities at the yard over the planning horizon, which is typically 24 hours.

Solving the TUSPwSPS is very complex due to the tight coupling of its subproblems and numerous operational constraints. The problem integrates aspects of multiple well-known NP-hard problems [39]. Crucially, these subproblems cannot be handled in isolation: decisions in one domain (e.g., where a train is parked) directly impact others (e.g., whether a cleaning team can reach it in time, or whether it blocks another train's path) [3]. The interplay between parking order, service timing, and staff logistics creates a vast and highly constrained search space. As a result, the integrated shunting and scheduling problem is intractable to solve exactly for realistic instance sizes. Even simplified formulations lead to enormous models. For example, an integrated MILP model for matching and parking by Kroon et al. required over 400,000 constraints for a single yard scenario, which was intractable

to solve directly [16]. In practice, this complexity forces planners to rely on manual procedures, which are very tedious and time-consuming. The problem's complexity grows combinatorially with the number of trains, available tracks, and required service tasks. Thus, finding plans with exact methods is generally out of reach, reinforcing the need for heuristic or approximate methods.

Consequently, recent research has increasingly focused on heuristic methods capable of producing high-quality, feasible solutions for the Train Unit Shunting Problem and its extensions. One of the most promising approaches is local search, which iteratively improves candidate solutions by exploring their neighborhood through small modifications. Local search has proven effective for large-scale combinatorial optimization problems, where exact methods are infeasible. In the context of shunting, van den Broek et al. [39] demonstrated that a local search algorithm operating on a Partial Order Schedule ($\mathcal{POS}$) representation could efficiently solve the TUSPwSPS to feasibility for most realistic instances. However, while this method successfully integrates service scheduling, it still relies on a heuristic subprocedure, called list scheduling, for personnel planning, which is external to the local search process. This separation leads to a decoupled optimization structure in which driver and staff assignments are computed after each local search iteration. Because the heuristic subprocedure is not an optimal algorithm, it may fail to find feasible personnel schedules even when they exist. This limits the overall feasibility and completeness of the approach, as the combined system can incorrectly classify feasible problem instances as infeasible. To solve this problem, this thesis aims to answer the following question.

> **Can the current state-of-the-art local search with list scheduling approach for the Train Unit Shunting Problem with Service and Personnel Scheduling be improved by incorporating the personnel scheduling subproblem, partially or fully, into the $\mathcal{POS}$-based local search process?**

To answer this question, we will answer these sub-questions:

1. What are the performance and feasibility limitations of the state-of-the-art $\mathcal{POS}$-based local search method that relies on a heuristic list scheduling procedure?
    a) Under what conditions does the List Scheduling Policy lead to infeasible personnel schedules in cases where feasible schedules exist?
    b) To what extent does the List Scheduling Policy degrade the performance of the overall solution, compared to an optimal personnel assignment for the same $\mathcal{POS}$?
2. Which mechanisms for integrating personnel decisions into the $\mathcal{POS}$-based local search can be designed, and how do they structurally affect the search space and evaluation process?

a) What are the structural and computational consequences of embedding different personnel subproblem variants (e.g., partial assignment, full assignment, fixed ordering) into the local search?

b) How can personnel assignment and ordering decisions be formally represented within the local search state and neighborhood structure?

c) How can the enlarged search space be controlled to balance expressiveness and search efficiency?

   i. To what extent does a two-stage search scheme (first optimizing shunting and service structure, then refining driver assignments) improve the trade-off between solution quality and computation time compared to a fully integrated search?

   ii. How effective are partial driver-schedule resets as a perturbation mechanism for escaping local minima induced by personnel-related decisions?

   iii. How does the staged expansion of the decision space conceptually relate to existing variable search space strategies in the literature?

3. How do the proposed integrated strategies compare to the baseline in terms of solution cost and effective search complexity on realistic TUSPwSPS instances under fixed time budgets?

   a) How do the methods differ in operational performance, as measured by conflict costs and penalty costs, for identical computation time budgets?

   b) how does the enlarged search space influence the algorithm's ability to find high-quality solutions within a fixed time budget?

Chapter 2

Problem Description

The problem considered in this thesis is the Train Unit Shunting Problem with Service and Personnel Scheduling (TUSPwSPS). An instance of the problem consists of the infrastructure layout of a shunting yard, a timetable for a certain time period for that yard, and the personnel scheduled for that day.

The infrastructure layout involves the tracks, their connections, and the locations of specific facilities. An example of such an infrastructure layout can be found in Figure 2.1. The timetable consists of all arriving and departing trains, their train unit composition, and the required services for each train unit. A small example of such a timetable is shown in Table 2.1.

All trains consist of electrical self-propelled train units. These units can be coupled and decoupled to form different trains. Each train unit is defined by a train type, which includes information such as its length and coupling times.

Solving the TUSPwSPS amounts to creating a plan to service the trains, assigning personnel to the service tasks, and departing the trains at the departure time defined by the schedule. A detailed description of the exact input and output can be found in Appendix A.1.

To create a plan, we have to solve the following subproblems.

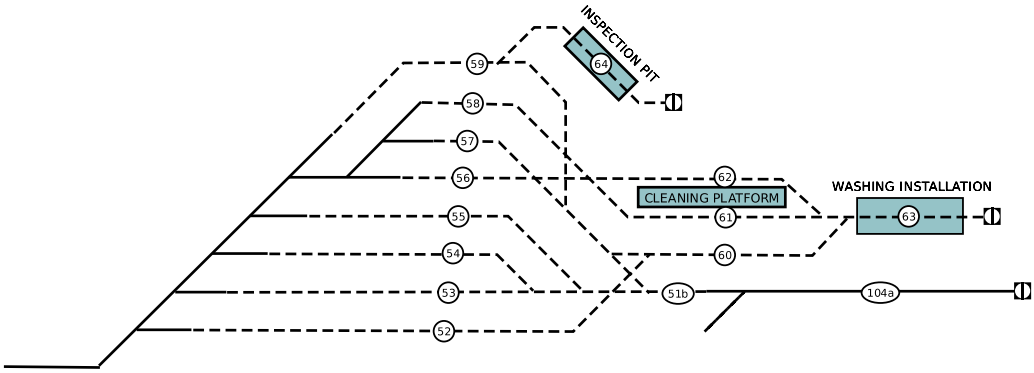


Figure 2.1: Shunting yard "Kleine Binckhorst". Based on: sporenplan.nl

Type	Train	Time	Composition	Required services
Incoming	I_1	22:00	VIRM-4 + VIRM-4	Cleaning (both units)
Incoming	I_2	23:20	SLT-4	Inspection
Incoming	I_3	01:35	VIRM-6	Cleaning
Outgoing	O_1	05:15	VIRM-4	–
Outgoing	O_2	06:40	VIRM-4 + VIRM-6	–
Outgoing	O_3	07:00	SLT-4	–

Table 2.1: Example of a small input schedule with incoming and outgoing trains.

1. **Matching**: All train units that come into the yard must be matched to outgoing trains such that all outgoing trains have the required train unit types in the right composition.

2. **Coupling and Decoupling:** Departing trains may have different train unit compositions from arriving trains. This means that the train units must be decoupled and then coupled to form the outgoing trains defined by the matching procedure.

3. **Parking:** Trains need to be parked whenever they are not moving. A train can only be parked on a track if the length of the track is at least as long as the combined length of that train and all the currently parked trains on that track.

4. **Routing:** For trains to be coupled, parked, serviced, or checked out, they need to be moved to their destination track. The plan should include a route over the infrastructure so that the path is not blocked by other trains and the train arrives on time at its destination track. Train units can be (temporarily) moved to another track to clear a path as many times as needed. Plans with trains leaving too late are considered infeasible.

5. **Service scheduling:** Train units might require certain servicing, such as cleaning and inspections. These services need to be scheduled such that the trains are at an allowed servicing location while being serviced, the servicing is completed before the departure time of that train. Plans that fail to perform all required services are considered infeasible.

6. **Personnel scheduling:** Each activity (movements, coupling and decoupling, and services) must be carried out by one or more staff members. Each activity requires a certain skill set, which only some staff members have. The number of available personnel is fixed for each day. Personnel scheduling entails matching these activities to staff members such that all activities are performed by the required number of eligible staff members, no two activities are scheduled simultaneously for the same staff member, and enough walking time is planned in between activities.

From this point onward, we focus specifically on train drivers when discussing the personnel scheduling component. In current operational practice, drivers form the primary bottleneck within yard personnel planning, as driver availability is typically the most restrictive resource. Therefore, concentrating on drivers captures the dominant source of complexity in the personnel subproblem. Nevertheless, the modeling framework and integration strategies developed in this thesis are, in principle, extendable to other categories of yard personnel with minor modifications.

A more detailed explanation of the problem can be found in the works of van den Broek et al. [38, 39]. However, unlike this work, for simplicity and because this is uncommon in practice, we assume that train drivers do not board other trains to reach their destination faster.

We also impose two extra constraints.

- **Fixed plan**: A fixed part of the plan that may include matching, movements, and handovers (see Appendix A.2). The solution should always include this fixed plan in order to be feasible.
- **Safety margins**: The safety margins are time buffers between certain actions on a specific part of the infrastructure to ensure safety. Solutions that do not include these buffers, when required, are considered infeasible.

Chapter 3

Related Works

The Train Unit Shunting Problem (TUSP) was first introduced by Freling et al. [9]. Due to the combinatorial complexity of the problem, many subsequent studies do not solve the problem in its entirety, but instead address a subset of the subproblems described in Chapter 2 separately [9, 21, 11]. While such decomposition-based approaches reduce computational complexity, they ignore strong interactions between subproblems. As a result, integrated formulations have been proposed to capture these interdependencies explicitly [16]. However, even simple models already exhibited severe scalability issues, illustrating the intrinsic difficulty of solving TUSP as a whole.

Later work, therefore, shifted toward heuristic and metaheuristic approaches. A local search algorithm was introduced by van den Broek et al. [39], operating on a partial order schedule ($\mathcal{POS}$), which compactly represents precedence relations between activities without fixing exact execution times. In the same paper they extend TUSP by incorporating service scheduling, resulting in the Train Unit Shunting Problem with Service Scheduling (TUSPwSS). Jacobsen and Pisinger [15] similarly propose local-search-based metaheuristics for variants of the TUSPwSS. These approaches demonstrate that local search is well-suited to large-scale shunting problems where exact methods are infeasible.

In both of these studies, personnel are modeled as generic capacity resources: activities consume a certain number of workers, but individual staff members are not explicitly scheduled. As a result, walking times and individual availability are ignored. While this abstraction simplifies the problem, it can yield solutions that are infeasible in real-world operations.

This limitation is addressed by van den Broek et al. [38]. By explicitly modeling walking times, they introduced the Train Unit Shunting Problem with Service and Personnel Scheduling (TUSPwSPS). In the same paper, they propose a method for solving the full TUSPwSPS problem [38]. Their approach retains the $\mathcal{POS}$-based local search for shunting and service planning, but treats personnel scheduling and exact timestamp assignment as a separate subproblem. For each candidate $\mathcal{POS}$ generated by the local search, a greedy list scheduling heuristic is used to construct a personnel schedule. A detailed description of this method is provided in Chapter 4.

While the heuristic for staff and driver assignment is incorporated into the cost evaluation of the local search, personnel decisions themselves remain outside the search space. As a result, the local search optimizes over a reduced representation of the problem, while feasibility critically depends on the outcome of an incomplete heuristic. In particular, the greedy list scheduling policy is pessimistic. It may create a schedule with very high tardiness costs, even when a schedule with a lower tardiness does exist. This separation can distort the effective search landscape, introducing or removing local optima that are not inherent to the underlying problem structure [40].

This combination of a metaheuristic and an embedded heuristic subproblem is not unique to this method. In the broader literature, hybrid metaheuristics often combine a main search procedure with either a destroy–repair mechanism, as in Large Neighborhood Search (LNS) [32], or with embedded exact optimization procedures (hybrid metaheuristics) [31, 18, 36]. In these methods, this separation has proven useful for navigating large search spaces. In LNS, however, the heuristic is not merely part of the evaluation step; the destroy–repair procedure itself constitutes the primary search operator, typically combined with multiple repair heuristics to explore different regions of the solution space. Hybrid metaheuristics that embed exact optimization procedures require that the embedded model can be solved sufficiently fast to be invoked repeatedly within the main search loop. The exact subproblem solver then provides optimal substructures that guide the metaheuristic. This reliance on fast solvability is crucial.

In the case of TUSPwSPS, however, the embedded component corresponds to a personnel scheduling problem of substantial computational complexity. From a broader scheduling perspective, the personnel scheduling component of TUSPwSPS belongs to a class of multi-resource scheduling problems with precedence constraints and sequence-dependent setup times, where staff members act as resources. Such problems are known to be strongly NP-hard [20]. Exact approaches for these problems, including mixed-integer programming and branch-and-price formulations, have been studied extensively [23, 38]. While these methods guarantee optimality, they are computationally too expensive to be embedded within an iterative local search framework that evaluates thousands of candidate solutions.

As a consequence, heuristic approaches are commonly employed for personnel scheduling. These include greedy list-scheduling policies and other priority-rule heuristics, in which activities or assignments are selected based on local priority criteria such as earliest due dates, least slack, or earliest feasible start times. Like list scheduling, such heuristics make irrevocable decisions and do not backtrack, which makes them computationally efficient but inherently incomplete in the presence of walking times and tightly coupled constraints. This incompleteness implies that the overall solution's quality may depend more on heuristic artifacts than on the structural quality of the underlying plan, suggesting that integrating a larger portion of the personnel-related decision space into the local search could mitigate these limitations.

Local search methods for the personnel scheduling subproblem have already been

studied. For example, Szabó [35] applies local search techniques to the crew scheduling subproblem in shunting yards. However, these methods still treat personnel scheduling as a separate optimization problem and do not integrate personnel decisions into a unified search space together with shunting and service planning.

Integrating personnel decisions directly into a unified search space significantly increases the problem's dimensionality and combinatorial complexity. Exposing additional assignment and sequencing variables enlarges the neighborhood structure and makes the search landscape more difficult to navigate [30]. Consequently, the added complexity must be mitigated by mechanisms that preserve search efficiency while retaining expressive power.

Beyond the shunting literature, several metaheuristics address this by changing the effective size of the search space during optimization to prevent premature convergence. Some methods dynamically expand and contract the accessible region of the solution space. Variable Landscape Search [22] perturbs not only solutions but also landscape properties, effectively reshaping the region that can be explored. Variable-size extensions [36] temporarily enable or disable subsets of decision variables, causing the search space to grow or shrink over time. Similarly, Variable Neighborhood Search (VNS) [13, 26] increases neighborhood size systematically, enlarging the effectively reachable portion of a fixed representation.

Other approaches iteratively alternate between distinct search spaces. For example, Demir et al. [6] proposes an iterative two-stage method for the Pollution-Routing Problem in which routing and speed decisions are optimized in alternating phases. Although the procedure switches between decision sets, each phase conditions on a partially fixed structure.

In contrast, this thesis adopts a staged expansion, in which additional decision variables are permanently activated over time. The search can therefore first explore the structural space of shunting and service decisions before incorporating detailed personnel assignments. The dimensionality of the search space is not reduced during the search. This enables progressive refinement without discarding previously invested search effort.

In summary, two main gaps remain in the literature.

First, although personnel scheduling is explicitly modeled in TUSPwSPS, it is not integrated into the local search state. Existing approaches separate shunting and service planning from personnel scheduling, treating the latter as a heuristic evaluation step. Consequently, solution quality depends on an incomplete greedy procedure that may distort the search landscape. This thesis addresses this gap by systematically studying how different levels of personnel decision-making can be incorporated into the local search process itself, rather than treating personnel scheduling as a fixed heuristic evaluation step.

Second, metaheuristic methods typically rely on temporary diversification mechanisms, such as changing neighborhoods or iteratively fixing parts of the search space to escape local minima. To the best of our knowledge, a structured approach in which the search space dimensionality is progressively and permanently expanded

during the search has not been studied. This thesis addresses this gap by introducing a two-stage approach that incrementally activates additional decision variables, enabling progressive refinements while preserving earlier improvements and maintaining flexibility in structural decisions.

Chapter 4

Baseline

NS currently employs a state-of-the-art local search algorithm for the Train Unit Shunting Problem with Service Scheduling (TUSPwSS), as introduced by van den Broek et al. [39], for the partial ordering of the shunting and service planning, combined with a heuristic approach for the train driver scheduling and assigning exact timestamps [38]. A detailed description of the implementation is provided in the corresponding publications [38, 39]. In the next sections, a summary of the method proposed by van den Broek et al. will be provided, along with any extensions to these methods that are currently employed by NS. This method will serve as the baseline for the remainder of the chapters. Finally, the limitations of this method will be discussed, serving as a starting point for the proposed new strategies in this thesis.

4.1 Local search for shunting and service planning

The local search algorithm proposed by van den Broek et al. [39] is a multi-neighborhood local search algorithm [5]. Since a feasible schedule is difficult to find due to the high density of activities and trains in the yard, the feasibility problem is converted into an optimization problem by relaxing certain constraints. This allows for easier exploration at the cost of an increased search space.

The constraints of the problem can be summarized into the following categories:

- **Matching**: All train units that come into the yard must be matched to outgoing trains such that all outgoing trains have the required train unit types in the right composition.
- **Sequencing**: No two activities using the same resource, infrastructure, or train unit can be planned simultaneously. All activities for one train unit must form a continuous chain, with the end location and time of one activity matching the start location and time of the next activity in the chain.
- **Temporal**:

  - **Arrivals and Departures**: Trains must arrive and depart exactly at the times as defined in the timetable.
  - **Driver schedules**: activities are not scheduled for a train driver before or after their shift.

- **Parking**:
  - **Track capacity**: The total length of all trains parked on a track can not exceed the track capacity.
  - **Crossings**: Two trains can not use the same infrastructure element at the same time.

Among these constraints, the matching and sequencing constraints are kept as hard constraints, while the temporal and parking constraints are converted to soft constraints. This is done by computing a weighted sum of these conflicts. From this point on, we will refer to this as the conflict cost. Besides the conflict cost, penalty costs are also added to the total cost function as a secondary cost function. These penalty costs, which include factors such as the number of movements and the time a driver spends at an inaccessible track, help guide the local search to a simpler or preferred solution.

The local search operates on a partial order schedule [28]($\mathcal{POS}$) of all the activities (shunting movements and services). A $\mathcal{POS}$ is a directed acyclic graph where nodes represent activities and arcs represent precedence constraints between these activities. The $\mathcal{POS}$ defines the three types of precedence constraints between activities. Train precedence constraints enforce a strict sequence of activities for each train unit, infrastructure precedence constraints prevent simultaneous use of the same track or infrastructure element, and resource precedence constraints ensure that activities requiring the same service resource do not overlap in time. An example of a $\mathcal{POS}$ can be found in Figure 4.1. This $\mathcal{POS}$ enforces the matching and sequencing constraints, while not specifying anything about the temporal and parking constraints.

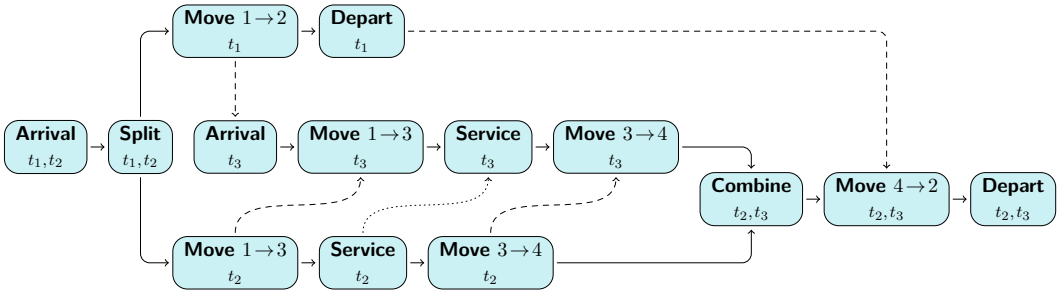
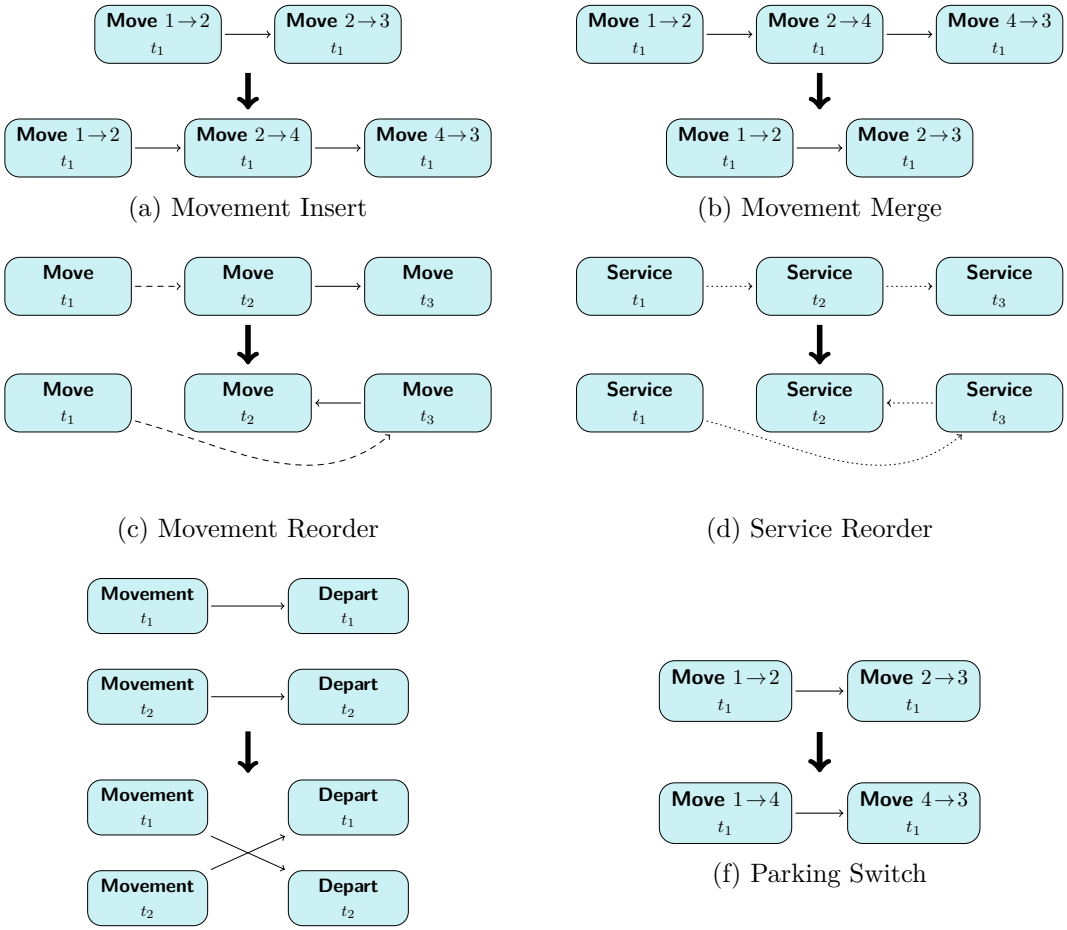


Figure 4.1: An example Partial Order Schedule. Solid arcs represent train unit constraints, dashed arcs represent infrastructure constraints, and dotted arcs represent service resource constraints. t_i denotes train i. 'Move $1 \rightarrow 2$' indicates a train movement from track 1 to track 2.



(a) Movement Insert

(b) Movement Merge

(c) Movement Reorder

(d) Service Reorder

(e) Matching Swap

(f) Parking Switch

Figure 4.2: The 6 most important neighborhoods in the baseline implementation. Solid arcs represent train unit constraints, dashed arcs represent infrastructure constraints, and dotted arcs represent service resource constraints. t_i denotes train i. 'Move $1 \rightarrow 2$' indicates a train movement from track 1 to track 2. Figure based on van den Broek et al. [39].

To select neighboring solutions, different neighborhoods are defined that mutate the $\mathcal{POS}$. In Figure 4.2, a visual representation of the most important neighborhoods can be found. For a full list of all used neighborhoods, refer to Appendix B.

Algorithm 1 describes a multi-neighborhood local search procedure for constructing a schedule for a given problem instance. The function SEARCH(*instance*) takes as input a complete TUSPwSPS instance and returns the best schedule $x^\star$ encountered during the search.

The algorithm maintains two solutions throughout its execution: a current solution x, which represents the present state of the search, and a best-so-far solution $x^\star$, which stores the lowest-cost schedule found at any point. The search is initialized

with a heuristic solution generated by initialSolution(*instance*).

The core of the method is an iterative improvement process that explores the search space using a collection of predefined neighborhoods. In each iteration, these neighborhoods are considered in a ranked order obtained from getRankedNeighborhoods(). This ranking reflects the past performance of neighborhoods and implements the variable ranked list strategy, where neighborhoods that have recently led to accepted moves are prioritized earlier in subsequent iterations.

For a selected neighborhood N, the algorithm generates a set of candidate solutions. Each candidate is then evaluated by constructing a complete schedule, including train driver assignments and exact timestamps, via constructSchedules($\cdot$). This step is crucial, since feasibility and cost can only be assessed on a fully specified schedule.

A random candidate is sampled from the set of candidate solutions until a candidate satisfies the acceptance criterion, after which it is accepted, or the maximum number of samples has been reached. There is also a minimum number of samples for each set of candidate solutions. If multiple candidates satisfy the acceptance criterion within this minimum set, the candidate with the lowest cost is accepted. The acceptance rule, implemented by accept?($\cdot$), is a strict improvement rule in our case. But it could also be a probabilistic rule, such as in simulated annealing.

Once a candidate from neighborhood N is accepted, the algorithm stops exploring further candidates and neighborhoods in the current iteration. The neighborhood N is then promoted in the ranking by calling updateNeighborhoodRanking(N), reinforcing neighborhoods that have proven effective. If the accepted solution improves upon the global best, $x^\star$ is updated accordingly.

If none of the neighborhoods yield an accepted candidate in a given iteration, the search is considered to be stuck in the current region of the search space. In this case, a perturbation step is applied to the current solution. This perturbation introduces larger, non-local changes intended to diversify the search and enable escape from local optima, after which normal neighborhood exploration resumes. Perturbing is done by either backtracking to the best-found solution, switching the weights of the cost function, reordering random movements, or performing a random walk (selecting random candidates from different neighborhoods).

The search continues until either a feasible solution is found or the computational budget is exhausted. Upon termination, the algorithm returns the best found solution $x^\star$.

Algorithm 1 Main Local Search Loop

```
function SEARCH(instance)
    x ← INITIALSOLUTION(instance)
    x* ← x
    while not FEASIBLE(x) or MAXRESOUCESUSED?() do
        ACCEPTED ← FALSE
        for each N ∈ GETRANKEDNEIGHBORHOODS() do
            n_x ← SAMPLENEIGHBORS(N, x)
            c ← 0
            for each s_i ∈ n_x do
                c ← c + 1
                x_i ← CONSTRUCTSCHEDULES(s_i)
                if ACCEPT?(COST(x), COST(x_i)) then
                    x ← x_i
                    if COST(x) < COST(x*) then
                        x* ← x
                    end if
                    ACCEPTED ← TRUE
                    UPDATENEIGHBORHOODRANKING(N)
                end if
                if (c >= MIN_SAMPLES ∧ ACCEPTED) ∨ (c >= MAX_SAMPLES) then
                    break
                end if
            end for
            if ACCEPTED then
                break
            end if
        end for
        if not ACCEPTED then
            x ← PERTURB(x)
        end if
        UPDATEACCEPTPARAMS()
    end while
    return x*
end function
```

4.2 Heuristic for driver scheduling and timestamp assignment

To get the final schedules, described by Algorithm 1 as CONSTRUCTSCHEDULES(s_i), the $\mathcal{POS}$ for shunting and service activities created by the local search in Section 4.1 is extended with driver assignments and exact timestamps. For this subproblem, a heuristic strategy is used: a greedy list scheduling policy. The heuristic is embedded

within the local search framework, being invoked for each considered candidate. This is done because the cost function can only be evaluated on a total schedule with assigned drivers and timestamps.

The list scheduling policy (LSP) operates on a total ordering $L = (a_1, a_2, \ldots, a_n)$ of the $\mathcal{POS}$. This ordered list is not necessarily the order in which the activities will take place, but the order in which they will be processed. When scheduling activity $a_i \in L$, all activities a_j with $j < i$ are treated as fixed, i.e., their start times st_{a_j}, completion times $ct_{a_j} = st_{a_j} + d_{a_j}$, and driver assignments are already determined. Here, d_{a_j} denotes the processing duration of activity a_j.

LSP then assigns the earliest feasible start time to the current activity, taking into account the availability of required resources such as trains, infrastructure, and drivers. The driver is chosen in a straightforward greedy manner, selecting the first available driver.

Within the local search, it is insufficient to terminate LSP with NULL when no feasible schedule is found. Since the local search framework requires a cost value to evaluate and compare solutions, infeasible outputs of LSP must be assigned a meaningful cost. Therefore, no deadlines are enforced. Instead, the heuristic schedules all activities, even if this leads to lateness. In such cases, a cost is computed as the sum of all tardiness values, i.e., the delay between the planned completion time and the latest allowable completion time of each activity. This tardiness cost is a part of the total conflict cost described in Section 4.1. This approach ensures that infeasible solutions (cost > 0) can still be compared and improved upon within the local search process, allowing the algorithm to gradually reduce tardiness and move toward feasibility. The pseudo code for this heuristic can be found in Algorithm 2.

LSP is not complete. Intuitively, incompleteness can arise in two ways. First, multiple valid linearizations of the partial order schedule can exist, leading to alternative feasible assignments that LSP may overlook. Second, the greedy allocation rule may assign a train driver to an activity early on, even though that driver might later be required elsewhere. This effect can be caused by walking times between tasks, since a seemingly feasible early assignment may prevent a driver from reaching a later activity in time. The remainder of this chapter will provide formal proofs of incompleteness.

4.2.1 Incompleteness of LSP

LSP is a greedy constructive heuristic. When scheduling each activity a_i from the linearization L, all prior assignments are irrevocably fixed. This means that LSP does not backtrack: once a decision is made for an earlier activity, it will not be reconsidered, even if it later prevents feasibility for subsequent activities. Proposition 1 proves that LSP is not complete.

Proposition 1. *The list scheduling policy is not complete: there exist instances of the driver scheduling and timestamp assignment subproblem for which a solution with tardiness* $= 0$ *exists, but LSP returns tardiness* > 0.

Algorithm 2 List scheduling policy [38]

function listSchedulingPolicy(s)
  $L \leftarrow$ Linearize(s)
  $S_{\text{trains}}, S_{\text{infra}}, S_{\text{drivers}} \leftarrow$ initSchedules()
  for each $a \in L$ **do**
    $W_{\text{train}} \leftarrow$ getTrainWindow(S_{trains}, a)
    $W_{\text{infra}} \leftarrow$ getInfraWindow(S_{infra}, a)
    $\mathcal{P} \leftarrow$ getDriverCandidates(a)
    for each $p \in \mathcal{P}$ **do**
      $e_p \leftarrow$ readyTime(S_{drivers}, p, a) ▷ The time p is ready at start location of a
      $W^p_{\text{drivers}} \leftarrow$ getDriverWindow(S_{drivers}, p, a) $\cap\, [e_p, \infty)$
    end for
    $(t_a, P_a) \leftarrow$ firstFeasibleStart($W_{\text{train}}, W_{\text{infra}}, \{W^p_{\text{drivers}}\}_{p\in\mathcal{P}}, \text{requiredDrivers}_a$)
    $S_{\text{trains}} \leftarrow$ schedule(S_{trains}, t_a)
    $S_{\text{infra}} \leftarrow$ schedule(S_{infra}, t_a)
    $S_{\text{drivers}} \leftarrow$ schedule($S_{\text{drivers}}, t_a, P_a$)
  end for
  return $S_{\text{trains}}, S_{\text{infra}}, S_{\text{drivers}}$
end function

Proof. Consider the following instance:

- three trains t_1, t_2, t_3,
- tracks r_a, r_b, r_c,
- two drivers d_1, d_2 with identical shifts $[0, 10]$, identical qualifications, and starting location r_a,
- walking times defined as $W_{ab} = 2, W_{bc} = 2, W_{ac} = 2$,
- activities:
  - A_1: prepare train t_1 on r_a; start window $[0, 7]$; duration 2,
  - A_2: prepare train t_2 on r_b; start window $[1, 3]$; duration 2,
  - A_3: prepare train t_3 on r_c; start window $[1, 3]$; duration 2.

These correspond to independent trains on distinct pieces of infrastructure with no shared resources, so there are no precedence arcs between A_1, A_2, and A_3 in the $\mathcal{POS}$.

Zero-tardiness schedule exists.
Assign both tight-window moves first, then the flexible move:

- Driver d_1 walks to track r_b during $[0,2]$.
- Driver d_2 walks to track r_c during $[0,2]$.
- Driver d_1 performs activity A_2 during $[2,4]$.
- Driver d_2 performs activity A_3 during $[2,4]$.
- Driver d_1 walks to track r_a during $[4,6]$.
- Driver d_1 performs activity A_1 during $[6,8]$.

This schedule satisfies all constraints and results in a total tardiness of 0.

Behavior of LSP.
Since there are no precedence constraints, any linearization is valid. Let the linearization be $L = (A_1, A_2, A_3)$.

- LSP processes A_1 first. The earliest start is $t = 0$ with d_1 (first available candidate). So $A_1 : [0,2]$ with d_1.
- LSP processes A_2 second. The earliest start is $t = 2$ with d_2. So $A_2 : [2,4]$ with d_2.
- Finally, LSP processes A_3. The earliest feasible start is $t = 4$ with d_1, since d_1 must walk from r_a to r_c ($W_{ac} = 2$). This means A_3 starts after its latest allowable time window $[1,3]$, resulting in a tardiness of 1.

The total tardiness of the schedule produced by LSP is therefore 1, which indicates that the resulting schedule is infeasible.
A feasible schedule with total tardiness 0 does exist, but LSP fails to find it. Therefore, LSP is not complete. □

Proposition 1 shows incompleteness, which is caused by the linearization of the $\mathcal{POS}$ in LSP. This is not the only reason that LSP is incomplete. It can also be caused by the inherent greedy assignment of LSP, which is shown by Proposition 2

Proposition 2. *There exists a driver-and-timestamp assignment instance and a partial order $\mathcal{POS}$ such that:*
(i) a zero-tardiness schedule exists that respects the $\mathcal{POS}$;
(ii) for every valid linearization L of the $\mathcal{POS}$, LSP produces a schedule with positive total tardiness.
Hence, LSP is not complete regardless of the choice of linearization.

Proof. Consider the following instance:

- two tracks r_a, r_b and one distant location r_c,
- two trains t_1, t_2,

- two drivers d_1, d_2 with identical shifts $[0, 10]$, identical qualifications, and starting locations: d_1 at r_b, d_2 at r_c,
- walking times: $W_{ba} = 2$, $W_{ca} = 3$, $W_{cb} = 5$,
- activities:
  - A_1: move t_1 from r_a to r_c to clear access; start window $[0, \infty]$; duration 1,
  - A_2: move t_2 from r_b to r_a; start window $[0, 4]$; duration 1.

Since t_2 can only depart once the access to r_a is cleared by t_1, there is a precedence $A_1 \prec A_2$ in the $\mathcal{POS}$.

Feasible schedule.
Assign A_1 to d_2 at $[3, 4]$: d_2 walks from r_c to r_a in $W_{ca} = 3$ and arrives exactly at $t = 3$. Execute A_2 at $[4, 5]$ with d_1. This schedule satisfies all constraints and results in a total tardiness of 0.

Behavior of LSP.
Since $A_1 \prec A_2$, any valid linearization schedules A_1 before A_2. When processing A_1, the earliest feasible start is $t = 2$, with d_1 (the first available candidate). When A_2 is considered at its start window $[0, 4]$, neither driver is available: d_1 is still at r_c and only returns to r_b at $t = 8$, while d_2 cannot reach r_b by $t = 4$ ($W_{cb} = 5$). LSP therefore schedules A_2 at its earliest possible time $t = 5$, resulting in a tardiness of 1.

The total tardiness of the schedule produced by LSP is thus 1, indicating that the resulting schedule is infeasible.

Conclusion.
A feasible schedule with total tardiness 0 does exist, but LSP fails to find it because it greedily assigns A_1 at its earliest possible start with the wrong driver and does not backtrack. Therefore, LSP is incomplete, regardless of the linearization strategy. □

Chapter 5

Embedded Subproblem Variants for Local Search

Chapter 4 shows that the current baseline approach, which combines a $\mathcal{POS}$-based local search with the List Scheduling Policy (LSP) for train driver scheduling, is incomplete with respect to the scheduling subproblem. For a fixed $\mathcal{POS}$, LSP may produce a schedule with higher total tardiness than the optimal schedule, due to its greedy, non-backtracking nature. As a consequence, the overall method can misclassify structurally feasible solutions as infeasible. However, LSP runs in polynomial time and is therefore suitable for repeated evaluation inside the local search. Exact methods, on the other hand, guarantee optimality but may be computationally expensive.

This raises the question of whether the scheduling subproblem can be reformulated such that it becomes easier to solve within the local search framework, thereby improving the quality of fast heuristic evaluation while reducing the computational cost of exact approaches. In particular, if part of the driver decision-making is removed from the embedded heuristic and transferred to the local search state, the remaining subproblem becomes more restricted and therefore easier to solve. This may narrow the gap between LSP and an optimal schedule. To systematically study this idea, we introduce three alternative embedded subproblem variants. Each variant is characterized by its input representation and removes part of the assignment or ordering decisions from the subproblem by assuming that these decisions are provided as input by the local search.

In this chapter, we first formally define the current embedded subproblem solved within the local search. After that, we introduce three alternative subproblems that progressively restrict the remaining decision space: (i) fixing the driver assignment of all activities, (ii) fixing the driver assignment of a subset of activities, and (iii) fixing both the driver assignment and the execution order per driver. To explain each subproblem, we use the running example shown in Figure 5.1. Finally, we conduct an experiment to evaluate the isolated impact of each subproblem variant, assuming that local search can provide an optimal input for each subproblem.

Activity	Description	Start Location	End Location	Duration	Deadline
A_1	Move t_1 from $a \to c$	a	c	1	6
A_2	Move t_2 from $b \to a$	b	a	1	6
A_3	Prepare t_3	b	b	5	6

(a) Activities

Before	After	Reason
A_1	A_2	Clear track a

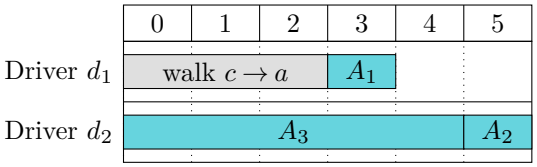
(b) Precedence

Driver	Start	Shift
d_1	c	$[0, 10]$
d_2	b	$[0, 10]$

(c) Drivers

	a	b	c
a	0	2	3
b	2	0	5
c	3	5	0

(d) Walking times



(e) Gantt chart of the optimal feasible schedule. Only driver schedules are shown for conciseness.

Figure 5.1: Example instance and $\mathcal{POS}$

5.1 Partial order schedule

This subproblem corresponds to the current problem solved inside the baseline approach described in Chapter 4. It entails (i) assigning drivers to activities, (ii) determining the execution order of activities within each individual driver's schedule, and (iii) assigning exact start times to all activities, while respecting precedence constraints, walking times, availability windows, and deadlines.

Formally, we denote this scheduling subproblem in Graham's three-field notation [10] as

$$P|p_j, size_j, fix_j, r_j, d_j, S^k_{i,j}, prec, avail_m|\Sigma T_j$$

Here, P represents a parallel-machine environment, where the machines correspond to train drivers, trains, and infrastructure. Activities are modeled as jobs. For each job j:

- $p_j \in \mathbb{N}_{>0}$ denotes its processing time,
- $size_j$ denotes the number of drivers simultaneously required,
- fix_j denotes the set of additional fixed machines (trains and infrastructure) that must be occupied during processing,
- r_j and d_j denote release dates and deadlines induced by arrivals and departures.

Sequence-dependent setup times $S^k_{i,j}$ model transition times between consecutive activities i and j on machine k. If k is a driver, the setup time captures walking time $W_{i,j}$. If k is an infrastructure element, it captures the required safety margin $m_{i,j}$. Since two activities may share both a driver and an infrastructure resource, the effective transition time must satisfy both requirements. Hence,

$$S^k_{i,j} = \max(W^k_{i,j}, m^k_{i,j}),$$

where $W^k_{i,j}$ is nonzero only if k is a driver and $m^k_{i,j}$ is nonzero only if k is an infrastructure element shared by both activities.

The precedence constraints $prec$ are induced by the $\mathcal{POS}$, and $avail_m$ denotes the availability interval $[A_m, B_m]$ of each driver m (non-driver machines have availability $(-\infty, \infty)$). The objective ΣT_j minimizes total tardiness.

This subproblem is strongly NP-hard, which we can proof by considering the decision version of the optimization problem.

Decision problem. Given a bound $K \in \mathbb{N}$, does there exist a non-preemptive schedule assigning to every job j a start time $t_j \in \mathbb{N}_{>0}$ and a set of machines M_j such that

$$\sum_{j \in \mathcal{J}} T_j \leq K,$$

where

$$C_j = t_j + p_j,$$

$$T_j = \max\{0, C_j - d_j\} + \sum_{m \in M_j} \max\{0, C_j - B_m\},$$

and the following constraints hold:

1. **Required machine number.** $|M_j| = size_j, \forall j \in \mathcal{J}$
2. **Simultaneous processing.** Job j occupies all machines in $M_j \cup fix_j$ during the interval $[t_j, t_j + p_j)$. No machine processes more than one job at a time.
3. **Release dates.** If r_j is given, then $t_j \geq r_j$.
4. **Machine Release dates.** $t_j \geq \max_{m \in M_j}(A_m)$
5. **Precedence constraints.** If $j' \prec j$, then $t_j \geq t_{j'} + p_{j'}$.
6. **Setup times.** For every machine k and every pair of consecutive jobs (i, j) scheduled on machine k: $t_j \geq t_i + p_i + S^k_{ij}$. If a machine is idle before job j, use $S_{\varnothing j}$.

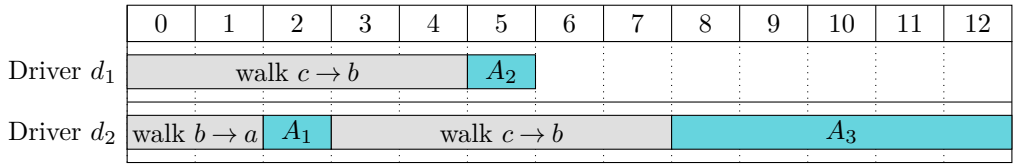
This decision problem is in NP, since a proposed schedule specifies all start times and machine assignments; checking the resource, release, availability, precedence,

and setup constraints, computing all completion times C_j, and evaluating the total tardiness are all doable in time polynomial in the input size.

Also, note that this problem generalizes the classical single-machine total-tardiness problem $1|r_j|\sum T_j$, which is known to be strongly NP-hard [20]. You can obtain $1|r_j|\sum T_j$ as the special case with $M = 1$, $size_j = 1$, $fix_j = \emptyset$ for all j, $S_{ij} = 0$ for all i, j, availability interval $[A_1, B_1] = [0, \infty)$, and no additional precedence constraints. Then the objective $\sum_j T_j$ coincides with the total tardiness defined above. Therefore, the decision problem is strongly NP-hard, and the corresponding optimization problem is strongly NP-hard as well.

Combining both results, the decision version is strongly NP-complete, and the optimization problem $P \mid size_j, r_j, d_j, S_{ij}, prec, avail_m \mid \sum T_j$ is strongly NP-hard.

Since this subproblem is NP-hard, exact algorithms do not scale well to realistic instance sizes in terms of computational complexity. Applying heuristics such as the list scheduling policy (LSP) can be done in polynomial time. However, as shown in Section 4.2.1, this comes at the cost of worse output schedules. This is also evident in the example instance. Since decisions must be made on both the assignment of activities to drivers and the order in which each driver executes their activities in this subproblem, LSP's greedy assignment can lead to the following schedule with total tardiness 6. (Note that multiple schedules could be valid, depending on the linearization method)



5.2 Partial order schedule + driver assignment

This subproblem represents the case where not only the $\mathcal{POS}$ is given, but also which driver will perform each activity. It can be written as follows.

$$P|fix_j, r_j, d_j, S_{ij}, prec, avail_m|\Sigma T_j$$

This subproblem can be modeled in a similar way as subproblem 5.1. In the input, we can remove the $size_j$ of each job. In this case, M_j becomes a constant instead of a variable, and the required machine number constraint is not needed anymore.

This subproblem is also strongly NP-complete. All constraints of the decision version of the subproblem can be verified in polynomial time, in a similar fashion as before. NP-hardness follows since this problem also generalizes the single-machine scheduling problem $1|r_j|\Sigma T_j$ [20]. This is done by reducing in the same way as subproblem 5.1. Since there is only one machine, all jobs have to be assigned to this one machine. Which is also the case for $1|r_j|\Sigma T_j$.

Therefore, the decision version is strongly NP-complete, and the optimization problem is strongly NP-hard.

Even though this subproblem remains strongly NP-hard, fixing the driver assignment already removes a significant source of difficulty for LSP. Once the optimal assignment of activities to drivers is fixed, LSP no longer needs to decide which driver performs which activity. This eliminates poor assignments that would otherwise cause unnecessary walking or idle time. However, LSP still has to determine the execution order of activities assigned to the same driver.

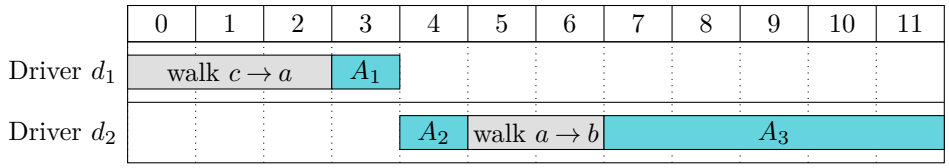
We can see this effect in the example instance. We add the optimal driver assignment to the input:

$$A_1 : d_1$$

$$A_2 : d_2$$

$$A_3 : d_2$$

Using the same linearization method as in Section 5.1, LSP produces the following schedule, with a total tardiness of 5:



This shows that, while LSP still does not yield optimal results, fixing driver assignments can improve the performance of heuristic scheduling.

5.3 Partial order schedule + partial driver assignment

In this subproblem, the $\mathcal{POS}$ is given as before, but only a subset of the activities already has an assigned driver. The remaining activities must still be assigned to available drivers. It can be written as follows.

$$P|size_j, fix_j, r_j, d_j, S_{ij}, prec, avail_m|\Sigma T_j$$

Note that this is the same notation we used for subproblem 5.1. Formally, this subproblem extends subproblem 5.1 as follows. For a subset of jobs $\mathcal{J}_{\text{fixed}} \subseteq \mathcal{J}$, the assigned machine sets M_j are fixed and given as input. For the remaining jobs $\mathcal{J}_{\text{free}} = \mathcal{J} \setminus \mathcal{J}_{\text{fixed}}$, machine sets must still be selected such that all constraints from Section 5.1 are satisfied.

This subproblem is strongly NP-complete. It is in NP because, given a proposed schedule, all constraints can be verified in polynomial time.

To show strong NP-hardness, observe that this problem generalizes both previous subproblems. In particular, it reduces to:

- Subproblem 5.1 if $\mathcal{J}_{\text{fixed}} = \emptyset$;

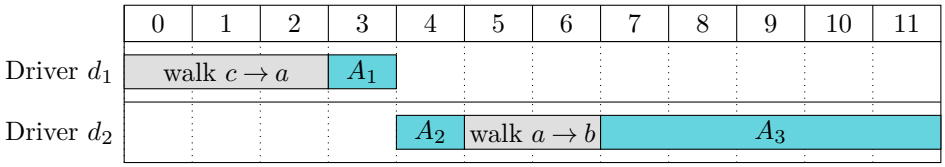
- Subproblem 5.2 if $\mathcal{J}_{\text{fixed}} = \mathcal{J}$.

Since both of these problems are strongly NP-hard, the current subproblem, which generalizes them, must also be strongly NP-hard.

Similar to subproblem 5.2, partially fixing the driver assignment reduces the number of greedy decisions made by LSP and can therefore improve solution quality. Instead of fixing all assignments, only a subset of activities is assigned to specific drivers, while the remaining assignments are left to the heuristic.

Fixing the assignment of the most critical activities often already prevents the most harmful assignment choices. This reduces the likelihood of excessive walking or avoidable tardiness caused by greedy decisions, while still allowing the local search to explore different driver configurations. As a result, this variant offers a balance between flexibility and complexity, in theory being able to achieve the same results as fixed driver assignment, without forcing the local search to make decisions that are less relevant or that LSP can already make well.

We can again see this effect in the example instance. When we only fix $A_1 : d_1$, LSP still creates the following schedule with tardiness 5:



5.4 Partial order schedule + fixed driver ordering

In this subproblem, the $\mathcal{POS}$ is given together with a total ordering of the activities for each driver. That is, for every driver $m \in \{1, \ldots, M\}$ we are given a sequence of jobs

$$(j_1^{(m)}, j_2^{(m)}, \ldots, j_{k_m}^{(m)}),$$

such that:

- each job $j \in \mathcal{J}$ appears in the sequence of exactly $size_j$ drivers (once per required driver),
- the order of jobs in these sequences is fixed and cannot be changed.

In other words, for each driver there is exactly one path through the activity graph. Note that this is already the case for trains and infrastructure: for each train unit and each fixed infrastructure resource, the $\mathcal{POS}$ induces a unique path of activities.

In Graham's notation, we can write this variant as

$$P \mid size_j, fix_j, r_j, d_j, S_{ij}, prec, avail_m, order_m \mid \Sigma T_j,$$

where $order_m$ denotes the fixed processing order of the jobs for each machine (driver) m.

In this variant, once the $\mathcal{POS}$ and the orders $order_m$ for all machines are fixed, there are no remaining combinatorial choices in the sequencing of jobs. The only remaining decision is to assign start times t_j to each job j such that all precedence, release, availability, and setup-time constraints are satisfied. This can be done greedily by assigning each job j its earliest feasible start time t_j.

$$t_j \;=\; \max\Big(r_j,\; \max_{i,j\in\mathcal{J}}\{t_i + p_i + S_{ij}\},\; \max_{m\in M_j} A_m\Big).$$

Computing this start time can be done in polynomial time, which means this subproblem is in P. Also note that this time assignment is also used by LSP, making LSP an optimal algorithm for this subproblem.

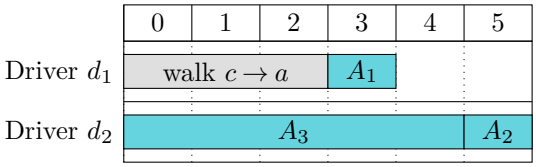
So in the example instance, when we also fix the order per driver:

$$d_1 : A_1$$

$$d_2 : A_3 \to A_2$$

LSP constructs the optimal schedule with tardiness 0. Note that since linearization no longer introduces ambiguity, there is only one valid schedule that LSP can construct, namely the optimal one.

Note that in this subproblem, a strict total ordering of infrastructure usage is assumed. In practice, such a strict total ordering does not always exist, to allow for more flexibility in the local search. In this case, while reducing the number of possible linearizations, there can still be multiple. Therefore, in these cases, LSP is still not optimal.



5.5 Impact of subproblems

As shown in the preceding sections, progressively fixing driver assignments and activity orders reduces the freedom left to greedy scheduling decisions. Subproblems 5.2–5.4 increasingly restrict the combinatorial choices faced by the subproblem solver, making the evaluation of a $\mathcal{POS}$ more stable and more representative of its true scheduling potential.

To determine the computational effect and the impact on tardiness of solving each subproblem with LSP compared to an exact method, a small experiment can be set up, assuming that local search can find an optimal input for each subproblem. This illustrates the isolated effect of each method on the outcome.

Here, we compare the execution time and total tardiness of LSP and an exact constraint model on each subproblem. To create a constraint model, we use the

Python library presented by Lan and Berkhout [19], which provides a generic modeling framework for scheduling problems based on constraint programming principles. This library enables the definition of machines, jobs, precedence constraints, and setup times as fundamental modeling constructs, closely aligning with the notation used in this section. Each driver, train, and piece of infrastructure is modeled as a machine, and each activity is represented as a job that requires a specific subset of these machines for its execution. Walking times and safety margins are incorporated as sequence-dependent setup times between jobs on shared machines.

We run each method on several instances generated specifically for this purpose[1]. We create 4 instance types, each with 30 instances, in two sizes and two difficulties. They are defined as:

- **Instance size.** The *small* and *large* instance classes are defined purely by structural scale parameters of the generated problem. Small instances use fewer tracks, fewer trains, fewer train drivers, and a smaller maximum number of units per train (e.g., 4 tracks, 3 trains, at most 2 units per train). Large instances increase each of these dimensions (e.g., 8 tracks, 5 trains, up to 3 units per train), thereby enlarging the temporal, spatial, and combinatorial search space.

- **Instance difficulty.** The *easy* and *hard* instance classes are defined by the activity density parameter. The activity density parameter $\in [0,1]$ controls how many intermediate activities are generated within each train flow and how long such activity chains become. Concretely, it affects the instance in two ways. First, it determines the maximum number of intermediate steps per flow via

  $$\text{max\_steps} = 3 + \lfloor 3 \cdot \text{activity\_density} \rfloor,$$

  So, higher values allow longer sequences of activities between arrival and departure. Second, at each step, an activity is added with probability equal to the activity density. As a result, low values (e.g. 0.4) lead to short, sparse activity chains with few optional tasks, while high values (e.g. 0.65) produce longer and denser chains containing more activities. Thus, the activity density directly represents the expected fraction of possible intermediate activities that are realized in the instance.

We can then use branch and bound to find the optimal driver assignment and ordering per driver, given the way LSP schedules tasks. This simulates the Local Search, which we assumed to be optimal for this experiment. Because of this assumption, subproblems 5.2 and 5.3 collapse into the same subproblem, since both would find the optimal driver assignment.

The results of this experiment can be found in Figures 5.2 and 5.3. What we can see from this experiment is that assigning drivers optimally, as in subproblems 5.2

[1] Code for instance generation was created using an LLM, see Appendix D.1 for more information.

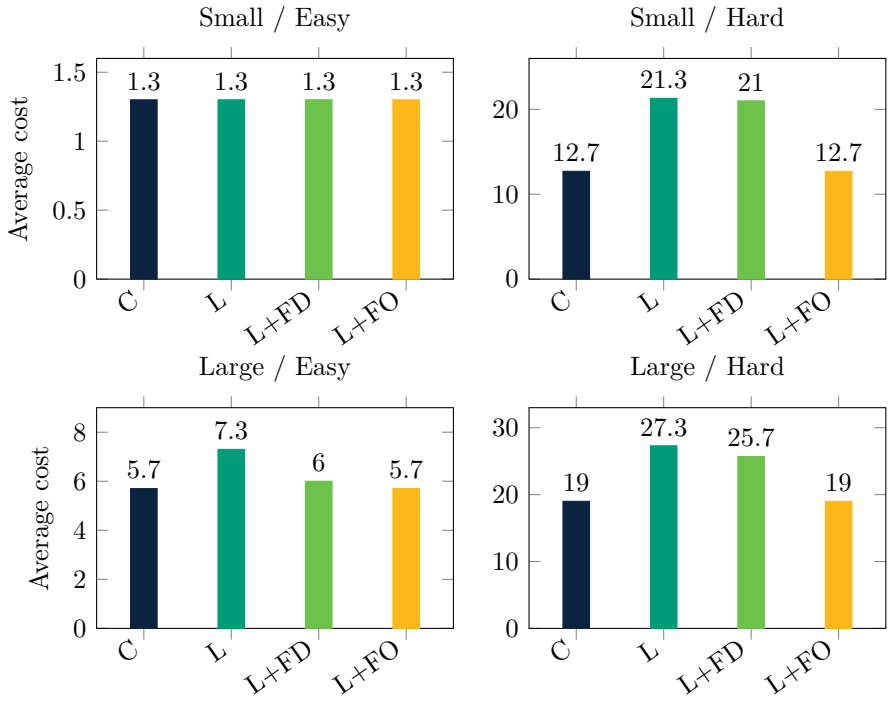


Figure 5.2: Average total tardiness of using LSP and constraint programming on each of the subproblems. Only one bar was shown for the constraint program, since it always finds the optimal solution.

and 5.3, can help LSP achieve better results. Additionally, fixing the activity ordering per driver will make LSP an optimal algorithm, as noted in subproblem 5.4. All while being an order of magnitude faster than the constraint program.

The distinction between easy and hard instances mainly affects solution quality: hard instances, generated with higher activity density, exhibit substantially higher optimal costs and a larger performance gap between LSP and the optimal solution. This indicates that increased temporal congestion and tighter precedence chains amplify the impact of greedy assignment decisions. In contrast, the difference between small and large instances primarily affects computational effort. The runtime of the constraint model increases significantly with instance size, while LSP scales only marginally.

While exact formulations are faster on each of the introduced subproblems than on the baseline subproblem, they are still too slow to be used within a local search loop. Note that the instances used for this experiment are still relatively small compared to realistic instances, which have up to 30 tracks and 40 train units per instance. This choice was made to keep the exact formulations computationally feasible. Therefore, the observed differences between small and large instances will

most likely be even more pronounced in realistic settings. This experiment shows that these subproblems can yield a lower cost when used inside the main local search loop, assuming local search is able to find the optimal input for these subproblems. This aligns with the hypothesis made in previous sections in this chapter.

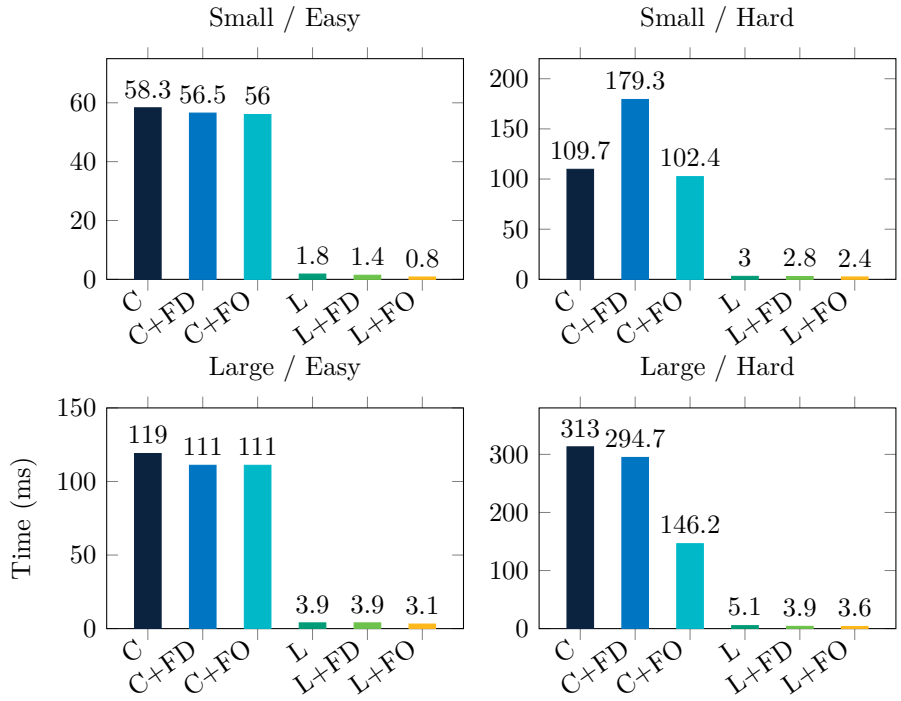


Figure 5.3: Average execution time in ms of using LSP and constraint programming on each of the subproblems.

Chapter 6

Proposed methods

In Chapter 5, it was shown that LSP performs better when fixing the driver assignment. Moreover, fixing the per-driver ordering makes LSP an optimal algorithm, while remaining computationally efficient. Throughout that chapter, we assumed that the local search procedure can provide an optimal or near-optimal input for each of these subproblems. However, this assumption is not guaranteed to be true in practice. Local search itself is a heuristic method, and its effectiveness depends strongly on the structure of the search space it explores. In particular, there is an inherent trade-off between the amount of responsibility assigned to the local search and the size and complexity of its search space. Assigning more decision power to the local search, by including driver decisions directly in the search state, increases flexibility and allows the algorithm to correct poor heuristic choices. At the same time, it significantly increases the search space's complexity, which may slow convergence and increase the risk of getting trapped in local optima [40].

This trade-off directly relates to Research Question 2: "Which mechanisms can be designed to integrate personnel decisions into the $\mathcal{POS}$-based local search, and how do they affect the structure and complexity of the search space?". The methods proposed in this chapter implement different integration strategies, each reflecting a distinct way of redistributing responsibility between the local search and the embedded driver scheduling procedure. Rather than assuming that greater integration is always beneficial, these methods allow us to empirically examine how increasing the expressiveness of the search state affects solution quality and computational effort.

This chapter is structured as follows. Section 6.1 introduces a two-stage search approach to control the increase in search-space complexity. Sections 6.2 and 6.3 describe the Full and Partial Driver Assignment methods, which integrate driver assignment decisions into the local search state. Subsequently, the Partial Driver Ordering method is presented in Section 6.4, extending the integration to per-driver activity sequences. Finally, Section 6.5 introduces a driver reassign perturbation to escape local optima induced by unfavorable driver decisions.

6.1 Two-stage search approach

To manage the increase in search-space complexity introduced by the extra driver decisions of the methods described in Sections 6.2–6.4, this thesis adopts a two-stage local search approach. The central idea is to first find a promising region of the search space in a smaller, less complex search space, before focusing on fine-tuning the driver scheduling.

In the first stage, the local search focuses on optimizing the shunting and service planning. Driver scheduling is handled by the list scheduling policy in this stage. This keeps the local search space comparatively simple and allows the algorithm to efficiently explore high-level structural changes, such as parking choices, movement sequences, and service ordering. The goal of this stage is to obtain a shunting and service plan that is structurally sound and close to feasibility, without committing too early to detailed driver decisions.

In the second stage, responsibility for driver-related decisions is shifted from the embedded heuristic to the local search. Starting from the best solution found in the first stage, the local search is continued with one of the methods described in this chapter. Note that decisions made in the first stage can still be changed in the second stage.

A hyperparameter $f \in [0,1]$ is introduced to control the division of the total time budget between the two stages. Let the total time budget be T. Setting $f = \alpha$ implies that the first stage (baseline method) receives αT, while the second stage (the extended method) receives $(1-\alpha)T$. Hence, f determines how much computational effort is spent on structural exploration versus detailed driver optimization.

By postponing this increase in responsibility until a good structural solution has been identified, the algorithm avoids the combinatorial explosion that would occur if all driver decisions were included from the start, while still allowing changes to the entire plan necessary to address problems arising from driver planning.

6.2 Full Driver Assignment

The full driver assignment method corresponds to subproblem 5.2. In this variant, the local search maintains a complete driver assignment for every activity that requires one. As a result, the list scheduling policy (LSP), explained in Section 4.2, no longer selects drivers during evaluation. Instead, LSP only determines the execution order of activities within each driver (as induced by the linearization L) and assigns the earliest feasible start time t_a for each activity, while respecting the fixed assignments.

Whenever a neighborhood introduces a new activity that requires drivers into the $\mathcal{POS}$, the method does not enumerate all possible driver combinations. Instead, LSP is invoked once to construct a single feasible set of drivers of the required size. This assignment is then stored in the search state and can subsequently be modified only by driver-related neighborhoods.

The main motivation for this design is to prevent a combinatorial explosion in the neighborhood size. If the local search were to consider all possible subsets of drivers of size r_a, the number of candidate moves would grow combinatorially as $\binom{|D|}{r_a}$. Even for moderate values of $|D|$, this leads to a rapid increase in neighborhood size, resulting in poor scalability of the search. By delegating the initial driver selection to LSP, the method keeps neighborhoods compact while still allowing the local search to refine driver decisions afterward. A downside of this approach is that it reintroduces reliance on the incomplete LSP heuristic.

6.2.1 Formal Definition

Let A denote the set of activities that require a driver, let D denote the set of available drivers, and let r_a denote the number of drivers required for $a \in A$. For each activity $a \in A$ and driver $d \in D$, we introduce a binary decision variable

$$x_{a,d} \in \{0,1\},$$

where $x_{a,d} = 1$ indicates that activity a is assigned to driver d.

In the full driver assignment method, each activity must be assigned exactly the required number of drivers:

$$\sum_{d \in D} x_{a,d} = r_a \qquad \forall a \in A.$$

Thus, the driver assignment is fixed in the search state. During evaluation, LSP does not modify these assignments.

6.2.2 Neighborhoods

The following driver-related neighborhoods are added to the local search. The neighborhoods operate directly on the variables $x_{a,d}$ while leaving the underlying $\mathcal{POS}$ unchanged. A visual representation can be found in Figures 6.1a and 6.1b.

1. **Driver swap:** Select two distinct activities $a_1, a_2 \in A$ and distinct drivers $d_1, d_2 \in D$ such that:

   $$x_{a_1,d_1} = 1, \quad x_{a_2,d_2} = 1, \quad x_{a_1,d_2} = 0, \quad x_{a_2,d_1} = 0.$$

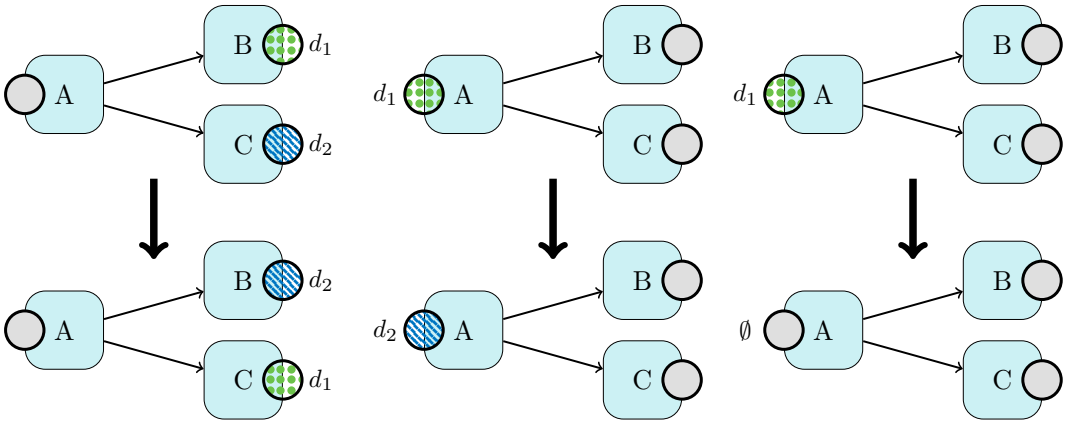
   Swap these assignments by setting

   $$x_{a_1,d_1} = 0, \quad x_{a_2,d_2} = 0, \quad x_{a_1,d_2} = 1, \quad x_{a_2,d_1} = 1.$$

2. **Driver switch:** Select an activity $a \in A$ and a driver $d \in D$ with $x_{a,d} = 0$. Set $x_{a,d} = 1$ and remove one existing assignment $x_{a,d'} = 1$ for some $d' \neq d$ so that

   $$\sum_{d' \in D} x_{a,d'} = r_a$$

   remains satisfied.



(a) Driver Swap (b) Driver Switch (c) Driver Unassign

Figure 6.1: Driver neighborhoods used in the local search for Full Driver Assignment (only 6.1a and 6.1b) and Partial Driver Assignment. Tags indicate fixed driver assignments for activities (dotted-green: d_1, striped-blue: d_2, solid-gray: unassigned).

6.3 Partial Driver Assignment

The partial driver assignment method corresponds to subproblem 5.3 and can be viewed as a relaxation of the full assignment approach. Instead of maintaining a complete driver assignment for every activity, the local search fixes driver assignments only for a subset of activities. The remaining activities are left partially or fully unassigned and are completed by LSP during evaluation, together with the activity ordering within a driver schedule and feasible start times.

The goal is to improve the schedules produced by LSP without exposing the entire driver decision space to the local search. By selectively fixing critical assignments, the method can correct poor early choices of LSP while keeping the search space smaller than in the full assignment variant.

6.3.1 Formal Definition

Using the same sets A and D, required driver counts r_a, and decision variables $x_{a,d}$ defined above, the full equality constraint is relaxed to

$$\sum_{d \in D} x_{a,d} \leq r_a \qquad \forall a \in A.$$

During evaluation, LSP assigns drivers to every activity a for which

$$\sum_{d \in D} x_{a,d} < r_a,$$

until the required number of drivers is reached.

6.3.2 Neighborhoods

The partial driver assignment method reuses the driver neighborhoods introduced for the full assignment variant, with a slight modification of the switch operation and the addition of an explicit unassign operation. As before, all neighborhoods operate on the variables $x_{a,d}$ while leaving the underlying $\mathcal{POS}$ unchanged. A visualisation of these neighborhoods can be found in Figure 6.1.

1. **Driver swap:** Identical to the driver swap neighborhood defined for the full driver assignment method.

2. **Driver switch (extended):** Select an activity $a \in A$ and a driver $d \in D$ with $x_{a,d} = 0$, and set
$$x_{a,d} = 1.$$
To maintain $\sum_{d' \in D} x_{a,d'} \leq r_a$, remove **at most** one existing assignment $x_{a,d'} = 1$ with $d' \neq d$. If the activity currently has fewer than r_a assigned drivers, removal is optional; if it already has r_a, one assignment must be removed.

   This change allows the number of drivers assigned to one activity to grow.

3. **Driver unassign:** Select an activity $a \in A$ and a driver $d \in D$ with $x_{a,d} = 1$. The operation removes the fixed assignment by setting
$$x_{a,d} = 0.$$

6.4 Partial Driver Ordering

The partial driver ordering method corresponds to subproblem 5.4. In this variant, the local search fixes not only driver assignments but also parts of the activity order executed by each driver. As a result, even more scheduling responsibility is shifted from the evaluation procedure to the local search.

Instead of determining the full order of activities for every driver, the method fixes only selected precedence relations between activities. The remaining ordering decisions are still determined during evaluation by the LSP linearization. This design prevents the search space from growing excessively complex while still allowing the local search to control important ordering decisions.

6.4.1 Formal Definition

To represent ordering decisions, we replace the assignment variables $x_{a,d}$ with binary precedence variables. For every pair of activities $a_1, a_2 \in A$ and driver $d \in D$, define
$$p^d_{a_1,a_2} \in \{0,1\},$$
where $p^d_{a_1,a_2} = 1$ indicates that activity a_1 must be executed before activity a_2 by driver d. If $p^d_{a_1,a_2} = 0$, no ordering between a_1 and a_2 is enforced for driver d.

If a driver d only has a single activity a assigned to them, this can be encoded as $p^d_{a,a} = 1$.

Note that, in contrast to subproblem 5.4, a total ordering per driver is not enforced. Meaning that we do **not** enforce that:

$$\forall a_1 \in A,\, \exists d \in D,\, \exists a_2 \in A : \, p^d_{a_1,a_2} = 1 \vee p^d_{a_2,a_1} = 1.$$

This prevents an excessive increase in the number of ordering decisions exposed to the local search.

Although a total order over all activities is not enforced, the precedence constraints of each individual driver form a single connected, acyclic chain. That is, for each driver d, the set

$$P_d = \{(a_i, a_j) \mid p^d_{a_i,a_j} = 1\}$$

induces a directed path over the activities assigned to d. Consequently, each activity assigned to a driver has at most one predecessor and at most one successor with respect to that driver. Also, assignment and ordering coincide: an activity a is assigned to driver d if and only if it lies on the chain P_d. Hence, a driver cannot have multiple disjoint assigned activities; every assigned activity must be part of this single chain.

It is possible that the baseline method's existing neighborhoods introduce precedence constraints between activities that conflict with driver precedence constraints. In such cases, those precedence constraints take priority, and conflicting driver precedence relations are removed if necessary. The neighborhoods introduced for this method each prevent cyclic precedence constraints. This is explained per neighborhood in Section 6.4.2.

6.4.2 Neighborhoods

The partial driver ordering method extends the neighborhoods introduced in Section 6.3. In addition to modifying driver assignments, the neighborhoods now also update the driver precedence variables. A visual representation is shown in Figure 6.2.

1. **Driver swap:**

   This operation extends the driver swap neighborhood. Instead of exchanging only driver assignments, it swaps one endpoint of two driver precedence relations while preserving the direction of both relations.

   Formally, consider two precedence relations

   $$p^{d_1}_{a_1,a_2} = 1 \quad \wedge \quad p^{d_2}_{a_3,a_4} = 1.$$

The operation selects $x \in \{a_1, a_2\}$ and $y \in \{a_3, a_4\}$ and exchanges them, producing one of the following relations:

$$\begin{aligned} p^{d_1}_{a_1,a_4} = 1 \quad &\wedge \quad p^{d_2}_{a_3,a_2} = 1, \\ p^{d_1}_{a_1,a_3} = 1 \quad &\wedge \quad p^{d_2}_{a_2,a_4} = 1, \\ p^{d_1}_{a_4,a_2} = 1 \quad &\wedge \quad p^{d_2}_{a_3,a_1} = 1, \\ p^{d_1}_{a_3,a_2} = 1 \quad &\wedge \quad p^{d_2}_{a_1,a_4} = 1. \end{aligned}$$

The operation is feasible only if the resulting precedence graph remains acyclic.

2. **Driver switch:**

   Select an activity $a \in A$ and a driver $d \in D$ such that a is not currently on the chain induced by P_d.

   If driver d has no activities assigned, the operation assigns a to d by setting

   $$p^d_{a,a} = 1.$$

   If driver d has exactly one activity b assigned, represented by $p^d_{b,b} = 1$, the operation replaces

   $$p^d_{b,b} = 1$$

   by either

   $$p^d_{a,b} = 1$$

   or

   $$p^d_{b,a} = 1,$$

   Otherwise, select a precedence relation $p^d_{b,c} = 1$, representing a position in the chain of driver d. The operation inserts a between b and c by replacing

   $$p^d_{b,c} = 1$$

   by

   $$p^d_{b,a} = 1 \quad \wedge \quad p^d_{a,c} = 1 \quad \wedge \quad p^d_{b,c} = 0.$$

   Let D_a denote the set of drivers whose chain currently contains activity a. If $|D_a| > r_a$ after the insertion, activity a must be removed from one or more other chains. If $0 < |D_a| \leq r_a$, removing a from another chain is optional. Removing a from the chain of some driver $d' \in D(a) \setminus \{d\}$ is performed by deleting the precedence relations incident to a for driver d' and reconnecting its predecessor and successor if both exist. If a has only a predecessor b (respectively only a successor c) on the chain of d', the relation $p^{d'}_{b,a} = 1$ (respectively $p^{d'}_{a,c} = 1$) is removed without adding a new relation. If a is the only activity on the chain of d', the relation $p^{d'}_{a,a} = 1$ is removed.

   The operation is feasible only if the resulting precedence structure remains acyclic.

3. **Driver unassign:**

   Select an activity $a \in A$ that is currently assigned to driver $d \in D$. Let $p^d_{b,a}$ and $p^d_{a,c}$ denote the precedence relations involving a in the chain of d, if they exist.

   If both relations exist, the operation removes a from the chain by setting

   $$p^d_{b,a} = 0 \quad \text{and} \quad p^d_{a,c} = 0,$$

   and introducing

   $$p^d_{b,c} = 1,$$

   thereby reconnecting the chain. If only one of the relations exists, say $p^d_{b,a} = 1$ or $p^d_{a,c} = 1$, it is replaced by the corresponding self-relation

   $$p^d_{b,b} = 1 \quad \text{or} \quad p^d_{c,c} = 1.$$

   If a is the only activity on the chain of d, represented by $p^d_{a,a} = 1$, this relation is removed.

4. **Driver activity reorder:**

   Select a driver $d \in D$ and two distinct activities $a_1, a_2 \in A$ on the chain induced by P_d such that

   $$p^d_{a_1,a_2} = 1.$$

   The operation swaps the order of a_1 and a_2 by replacing

   $$p^d_{a_1,a_2} = 1$$

   together with any existing relations

   $$p^d_{a,a_1} = 1 \quad \text{and} \quad p^d_{a_2,b} = 1,$$

   by

   $$p^d_{a_2,a_1} = 1,$$

   and, if applicable,

   $$p^d_{a,a_2} = 1 \quad \text{and} \quad p^d_{a_1,b} = 1.$$

   The operation is feasible only if the resulting precedence structure remains acyclic.

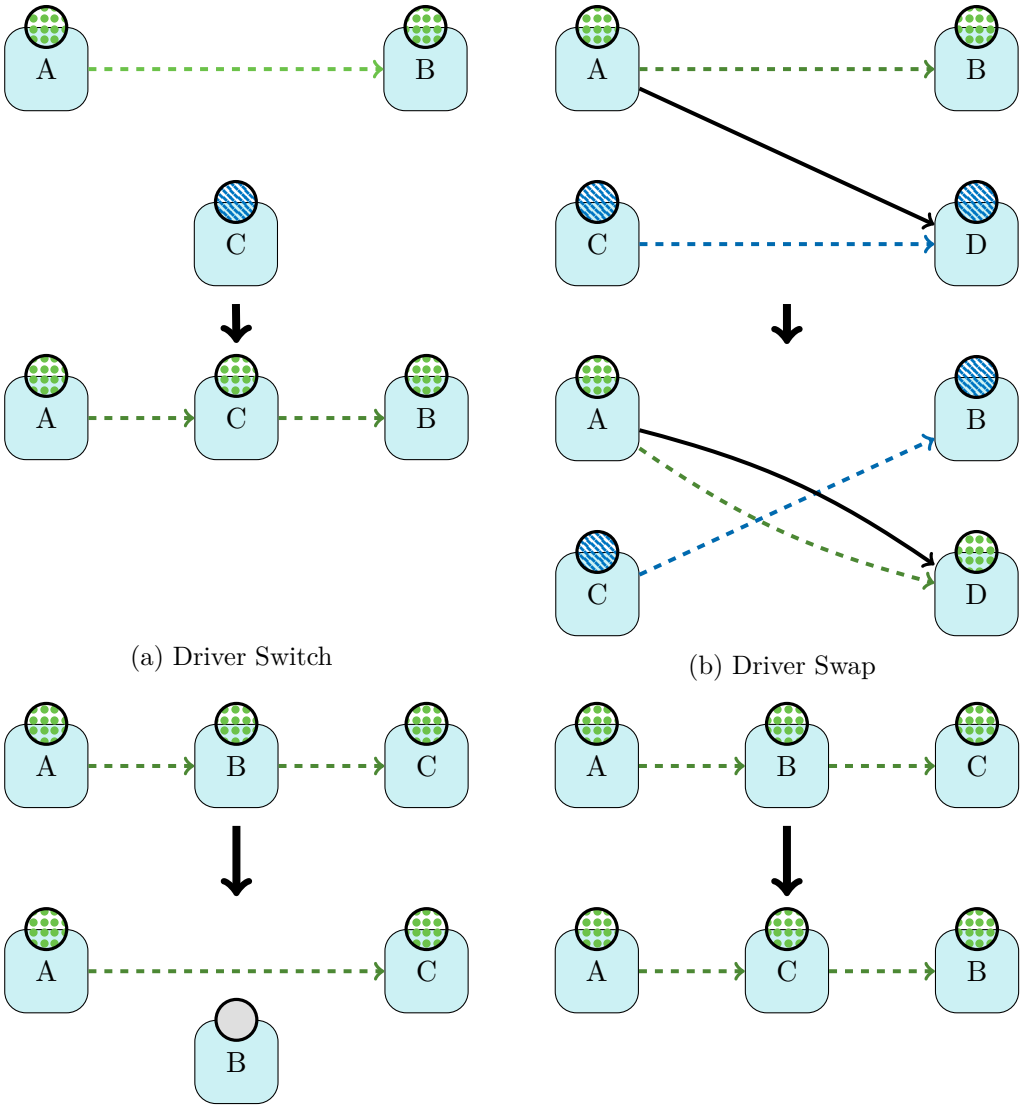


Figure 6.2: Driver neighborhoods introduced in the local search for Partial Driver Ordering. A, B, and C are the activities. Dashed colored arcs indicate driver precedence constraints. Solid arcs indicate other precedence constraints. Tags indicate fixed driver assignments for activities (dotted-green: d_1, striped-blue: d_2, solid-gray: unassigned).

6.5 Driver Reassign Perturbation

To escape local optima induced by unfavorable driver decisions, we introduce a driver reassignment perturbation. This perturbation partially destroys the current driver assignment and ordering decisions and lets LSP reconstruct them, similar to large

neighborhood search [25, 27].

Let

$$X = \{(a,d) \mid a \in A_{\text{driver}},\ a \text{ is currently assigned to } d\}$$

denote the set of existing activity–driver combinations in the current solution. The perturbation samples a fixed fraction $\rho \in (0,1)$ of the pairs in X. For every selected pair (a,d), the assignment of driver d to activity a is removed analogous to the driver unassign neighborhoods described in Sections 6.3.2 and 6.4.2

By removing assignments in this way, parts of both the driver allocation and the driver activity order are destroyed. The perturbed solution is subsequently repaired by LSP, which reconstructs feasible assignments and restores consistent driver activity chains. This allows the search to revise clusters of unfavorable driver assignment and ordering decisions that are difficult to resolve using local neighborhoods alone.

6.6 Overview of the Proposed Methods

The proposed methods differ in how driver-related decisions are divided between the local search and LSP. Table 6.1 summarizes these structural differences.

Method	**Driver assignment**	**Driver ordering**	**Search space**
Baseline	heuristic	heuristic	small
Partial Driver Assignment	mixed	heuristic	medium
Full Driver Assignment	local search	heuristic	large
Partial Driver Ordering	mixed	mixed	large

Table 6.1: Structural differences between the proposed methods. "Heuristic" means the decision is made entirely by LSP. "Local search" means the decision is fully fixed in the search state. "Mixed" means part of the decision is fixed in the search state and the remainder is completed by LSP.

From top to bottom, an increasing portion of the driver decision space is controlled directly by the local search. This reduces the combinatorial responsibility of LSP and makes the evaluation of a given $\mathcal{POS}$ less dependent on greedy choices. At the same time, it enlarges the search space size and complexity.

This leads to a budget-dependent expectation for the computational experiments in Chapter 7. With sufficiently large time budgets, stronger integration (e.g. Partial Driver Ordering) may yield better results because the search has enough time to exploit the richer decision space. Under tighter time budgets, methods with less integration (e.g. Partial Driver Assignment) may perform better, as they maintain a smaller and more navigable search space while still reducing harmful heuristic decisions.

The two-stage search and the driver reassignment perturbation are expected to improve performance in all methods by gradually activating integration and helping the search escape from locally poor driver configurations.

Chapter 7

Results

This chapter presents the computational results of the methods introduced in Chapter 6. The purpose of this chapter is not only to evaluate the proposed methods, but also to determine how their performance should be assessed and how the most effective configuration choices can be identified before making a final comparison with the baseline method (Chapter 4). To this end, we first introduce the experimental setup, including the hardware setup, instance set, and the performance metrics used for comparison. We then study, in sequence, the allocation of computational budget between the two stages, the effect of the driver-reassign perturbation, and finally the comparative performance of the resulting method configurations.

7.1 Experimental Setup

7.1.1 Hardware Setup

Due to the large runtimes of each run, which are 30, 60, and 90 minutes, the total runtime to test one method is multiple days on a single CPU. To finish the experiments in a reasonable time, the high-performance computer at TU Delft is used to run the experiments for this thesis [7]. We use a node with 64 cores (2x Intel XEON E5-6448Y 32C). This means we can run 64 experiments simultaneously, drastically reducing the total runtime.

Note that we set a time budget instead of an iteration budget. The reason for this is twofold. The first reason is that in practice, there is also a time budget, since a plan needs to be created before a certain deadline. The second reason is that methods might have different runtimes per iteration. So if one method performs many computations per iteration, it gets more total compute power when an iteration budget is used. Therefore, a time budget is the fairest comparison.

Since we are using a time budget, we need to ensure that each method has the same amount of computing power available. This is done by assigning equal amounts of dedicated cores to each method. When comparing n different methods with each other, we assign $\lfloor \frac{64}{n} \rfloor$ cores to each method. This means all methods run simultaneously on the same instance rather than sequentially. This eliminates

any time-dependent discrepancies between methods on the same run. We also add a dummy task to the end of each experiment to ensure all CPUs remain busy throughout. This is to ensure that all runs are executed with equal resources.

Although all runs are executed in separate `.NET` processes and therefore do not share runtime components such as garbage collection, they still compete for shared hardware resources on the same node, such as CPU capacity and memory bandwidth. However, the node provides abundant main memory capacity, far exceeding the actual usage of the experiments, so memory pressure is not expected to influence the results. Any remaining shared-resource effects are therefore assumed to introduce only negligible variability and are not considered to materially affect the experimental conclusions.

7.1.2 Instance Set

We test each method on 26 real-life instances over 3 different hubs. An overview of these hubs and their properties can be found in Table 7.1. These hubs were chosen to be diverse in their difficulty. With the hardest being Nijmegen, the easiest being Vlissingen, and Enkhuizen being somewhat in between. Difficulty is mostly determined by the instance size here. Each instance is run 12 times with different seeds.

We split the instances into two sets: the validation set and the test set. The validation set is used to run experiments in which parameters are fine-tuned, while the test set is used only for final evaluation.

In general, evaluating the algorithms on a larger set of instances is desirable, as this provides a more reliable and general indication of their performance. However, it was decided to test only 26 instances for two reasons. First, obtaining real-life instances is a very time-consuming task. Second, the runtimes of the algorithms are quite long, so including more instances would lead to unreasonable overall experimental runtimes.

Hub	**#instances**		**#tracks**	**infra length (m)**	$\overline{\text{\#drivers}}$	$\overline{\text{\#train units}}$
	test	**val**				
Nijmegen	6	5	29	8,207	5.63	33
Enkhuizen	6	6	8	2,508	1.00	19
Vlissingen	2	1	10	2,443	1.31	9

Table 7.1: The three hubs on which the experiments are run. #instances indicates the number of test and validation instances. #tracks is the number of parkable tracks in the yard. infra length (m) is the total length of all tracks. $\overline{\text{\#drivers}}$ denotes the average number of drivers available simultaneously, and $\overline{\text{\#train units}}$ the average number of incoming train units present in the instances.

7.1.3 Evaluation Metrics

To evaluate the methods proposed in Chapter 6, we would ideally measure the number of solved instances, since this is essentially a satisfiability problem, not an optimization problem. However, this would cause a problem with the size of our instance set. Since we only test on 12 instances for the validation set and 14 for the test set, most of which cannot be solved by any of the methods, using the number of solved instances as a metric would require a much larger dataset with more instances that can be solved by some (but not all) of the tested methods.

To still be able to test the performance of each method, we instead look at the conflict and penalty costs introduced in Chapter 4. As a reminder, conflict costs are associated with violations of constraints imposed by the problem (e.g., lateness, exceeding track capacity). Penalty costs are associated with preferred or simpler solutions (e.g., number of movements). Even though low costs do not give a theoretical guarantee that an instance is close to being solved, they have proven to be an effective performance metric within Dutch Railways. Conflict costs are the primary cost metric used for decision-making, since this corresponds with violations of constraints. The penalty cost will be used as a secondary metric to provide additional insight into structural differences between methods. In particular, it helps to interpret how methods trade off feasibility-related improvements against solution simplicity.

The primary comparison between methods is based on statistical hypothesis testing over paired runs. For every instance $i \in I$ and seed $s \in S$, we compare the obtained costs $C_{i,s}^m$ between two methods using the Wilcoxon Matched-Pairs Rank-Sum test [8, 14] and the proportion test as described by Fox and Long [8]. We use a significance level of $p < 0.05$.

The Wilcoxon test evaluates whether one method consistently outperforms another by ranking the absolute paired differences and taking their signs into account. The Wilcoxon test is our primary metric, since it considers both direction and magnitude of differences.

When the Wilcoxon test does not indicate a significant difference, we additionally apply the proportion test. This test evaluates whether the proportion of wins of one method over another is significantly different from 0.5. Although weaker than the Wilcoxon test, it still provides information about consistent directional differences in performance.

The statistical tests induce a partial order over the set of methods. In the results section, conclusions are primarily based on this partial order.

To support and interpret the statistical results, we additionally report these descriptive performance metrics where needed. In these metrics, we will denote the set of all methods as M, and the set of all seeds as $\mathcal{S}$

1. **Mean Cost** This metric is the most simple. Let $C_{i,s}^m$ denote the lowest found cost of instance i on seed $s \in S$ for method $m \in M$. Then the mean cost is

defined as

$$\bar{C}_i^m = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} C_{i,s}^m$$

When looking at single instances, the mean cost is the most informative metric. The problem with this metric is the inherent difficulty differences between instances. I.e., some instances are harder to solve than others, resulting in higher costs. This means we can not directly compare the results between instances.

2. **Normalized Gap to the Best Found Solution.** Let C_i^{best} denote the minimum cost found for all methods on instance i.

   i.e.

   $$C_i^{\text{best}} = \min_{m \in M} \min_{s \in \mathcal{S}} C_{i,s}^m$$

   Then the Normalized Gap to the Best Found Solution [33, 12] is defined as

   $$NGB_i^m = \frac{\bar{C}_i^m - C_i^{\text{best}}}{\max(C_i^{\text{best}}, 1)}.$$

   where the $\max(C_i^{\text{best}}, 1)$ is used instead of C_i^{best} to prevent division by zero.

   This metric expresses how far a method is from the best-performing method on that instance, while being scale-independent across instances. Note that since C_i^{best} is constant over one instance for all methods, this is essentially a normalized (and shifted) version of the mean cost. Allowing for direct comparison between instances. A downside of this normalization is that when C_i^{best} is close to zero, even small absolute cost differences can lead to relatively large normalized gaps.

3. **Binary Marginal Contribution.** This metric counts how often method m is strictly necessary to obtain the best known solution. More precisely, BMC_m measures the number of instance-seed pairs for which the best cost found over all methods cannot be reproduced when method m is removed from the set of methods. As such, it quantifies the unique contribution of a method to the overall best-found solutions, rather than its average performance. It can be formally defined as follows.

   For any subset of methods $A \subseteq M$, define the best cost on instance–seed pair (i, s) within that subset as

   $$C_{i,s}^{\min}(A) = \min_{m \in A} C_{i,s}^m.$$

   Then

   $$BMC_m = \sum_{i \in I} \sum_{s \in \mathcal{S}} \mathbf{1}\Big(C_{i,s}^{\min}(M \setminus \{m\}) > C_{i,s}^{\min}(M)\Big),$$

   where $\mathbf{1}(\cdot)$ is the indicator function.

This definition is based on the definition of König et al. [17].

While BMC_m is easy to aggregate over instances and highlights complementarity between methods, it does not capture the magnitude of performance differences. A method that is slightly better than others is weighted the same as a method that yields a substantially better solution.

4. **Quantitative Marginal Contribution** Whereas Binary Marginal Contribution (BMC) counts the number of occurrences a method was solely responsible for finding the best solution, Quantitative Marginal Contribution (QMC) [1, 17] indicates by how much better it was.

   It is defined as:

$$QMC_m = \sum_{i \in I} \sum_{s \in \mathcal{S}} \left(C_{i,s}^{\min}(M \setminus \{m\}) - C_{i,s}^{\min}(M) \right).$$

5. **Best-of-k Win Probability.** In practice, an instance is run k times with different random seeds, and the best solution is selected. To evaluate performance under this protocol, this metric estimates the probability that a method produces the best final solution when run k times. Note that this metric is quite similar to the more widely known Best-of-k metric [37]. However, here we estimate the probability that the Best-of-k is equal to the best found solution.

   Let $\mathcal{S}_k$ denote the set of all subsets of $\mathcal{S}$ that contain exactly k elements:

$$\mathcal{S}_k = \{S \subseteq \mathcal{S} \mid |S| = k\}$$

   For a subset of seeds $S \in \mathcal{S}_k$, define the best-of-k cost

$$C_{i,T}^m = \min_{s \in S} C_{i,s}^m.$$

   The Best-of-k Win Probability is

$$BWP_m(k) = \frac{1}{|I|\,|\mathcal{S}_k|} \sum_{i \in I} \sum_{T \in \mathcal{S}_k} \mathbf{1}\left(C_{i,T}^m = C_i^{\text{best}} \right).$$

   In the results from Section 7.2, we will always use $k = 4$, since this is used in practice by NS.

7.2 Computational Results

In this section, we investigate whether the proposed extensions improve upon the baseline at all, under which stage division this improvement is strongest, whether periodically perturbing driver assignments is beneficial, and how the resulting method variants compare in terms of both conflict and penalty cost.

Since preliminary experiments indicated that the stage division has a larger effect on performance than the perturbation, we first determine whether the two-stage approach is beneficial at all and, if so, how the budget should be divided between the two stages. We then analyze the perturbation effect using these settings. Jointly testing all combinations was computationally infeasible with the available resources, so the chosen ordering prioritizes the factor expected to have the greatest impact. Both configuration experiments are conducted on the validation instance set, after which the selected configurations are evaluated on the test set in a direct comparison between the proposed methods and the baseline.

7.2.1 Optimal Stage Division

To evaluate the effect of the proposed two-stage method and to identify an appropriate allocation of computational budget between the two stages, we conduct experiments for different values of the hyperparameter f defined in Section 6.1. We consider $f \in \{0.0, 0.25, 0.50, 0.75, 1.0\}$. Here, $f = 0.0$ corresponds to allocating the entire budget to the extended method, while $f = 1.0$ assigns the full budget to the baseline method.

The results of this experiment can be found in Figure 7.1. The figure shows a partial-order graph for each method-budget combination aggregated over all hubs. An edge from node $f = \alpha$ to node $f = \beta$ indicates that $f = \alpha$ performs significantly better than $f = \beta$ according to the Wilcoxon or Proportion test.

First, we observe that neither $f = 0$ nor $f = 1$ is ever the unique non-dominated configuration, and both are frequently dominated by intermediate values of f. This provides direct evidence that a staged enlargement of the search space yields better performance than committing all computational effort to either structural exploration ($f = 1$) or driver integration ($f = 0$).

The remaining question is which value of f should be selected for each method–time budget combination. When the partial-order graph contains a node that has no incoming edges and dominates a strict superset of the nodes dominated by any other node, we treat this node as the best choice.

Using this criterion, Figure 7.1 yields a clear best value for f for 6 of the method-budget combinations:

- Partial Driver Assignment: $f = 0.75$ (30 min), $f = 0.5$ (60 min), $f = 0.5$ (90 min)
- Full Driver Assignment: $f = 0.75$ (60 min)
- Partial Driver Ordering: $f = 0.75$ (60 min), $f = 0.75$ (90 min)

For the remaining three combinations, there are 2 or more nodes that are not dominated and dominate an equal number of other nodes. For these, we will use the descriptive performance metrics to determine a value for f. These metrics can be found in Table 7.2. Here, it is evident that for the Full Driver Assignment (90 min),

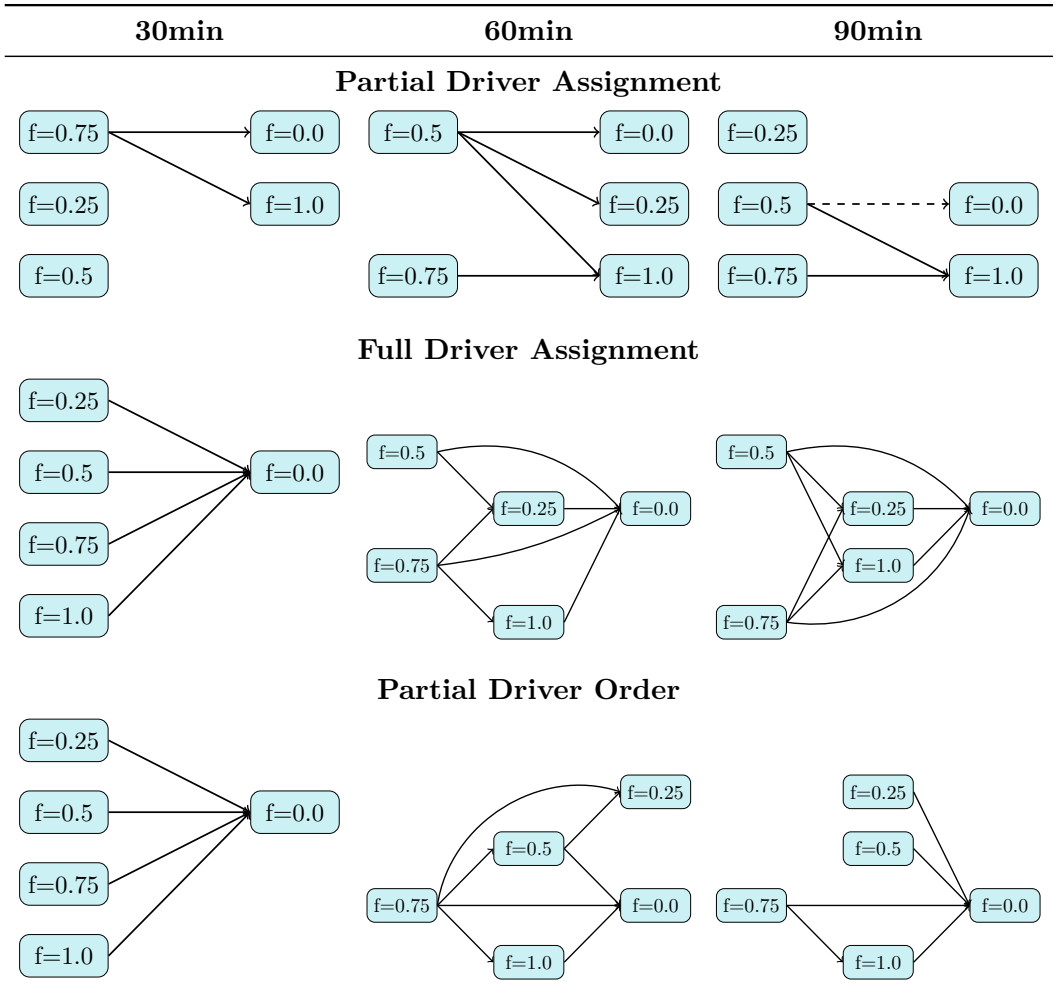


Figure 7.1: Partial ordering on conflict cost over all hubs. Solid arrow indicates a significant ($p < 0.05$) difference according to at least the Wilcoxon test, and the dashed arrow indicates a significant difference only according to the proportional difference. $f = \alpha$ means that the the first stage gets $\alpha * time\_budget$ and the second stage gets $(1 - \alpha) * time\_budget$.

$f = 0.5$ performs slightly better than $f = 0.75$ for 3 out of 4 metrics. Therefore, $f = 0.75$ is chosen for subsequent experiments. For Full Driver Assignment (30 min) and Partial Driver Ordering (30 min), no clear conclusion can be drawn. In these cases, the total time budget is limited, while the second stage introduces additional driver-related decision variables and neighborhoods. As a result, allocating a large portion of the budget to the second stage increases the dimensionality of the search without evidence that this yields better performance.

Therefore, when the evidence does not clearly favor one configuration, we adopt a conservative tie-breaking rule: we select the largest f that is not equal to the baseline

Setting	f	BMC	QMC	$\overline{NGB}$	BWP_4
Full Driver Assignment (30 min)					
	0.25	27	**558.0**	**0.86**	0.47
	0.50	**28**	342.0	0.87	0.45
	0.75	9	90.0	0.96	**0.55**
	1.00	19	264.0	6.92	0.54
Full Driver Ordering (30 min)					
	0.25	**27**	503.0	**0.79**	0.50
	0.50	25	**507.0**	0.83	0.52
	0.75	19	229.0	0.81	**0.56**
	1.00	15	217.0	1.04	0.54
Full Driver Assignment (90 min)					
	0.50	**22**	**303.0**	4.62	**0.52**
	0.75	11	75.0	**4.50**	0.45

Table 7.2: Binary Marginal Contribution (BMC), Quantitative Marginal Contribution (QMC), mean Normalized Gap to the Best Found Solution ($\overline{NGB}$), and Best-of-4 Win Probability (BWP_4) for different values of f where no significant difference was detected by the Wilcoxon or proportion test. Bold values indicate the best result within each block.

($f = 1.0$). This retains the two-stage structure while allocating the majority of the time budget to the structurally simpler first stage, thereby limiting unnecessary expansion of the search space under tight computational budgets. Consequently, $f = 0.75$ is used in the subsequent experiments for these cases.

Overall, the results exhibit a consistent pattern across methods and time budgets. Extreme allocations ($f = 0$ or $f = 1$) are never preferred, while intermediate values of f frequently dominate. The selected values of f, summarized in Table 7.3, are used in all subsequent experiments.

	30 min	60 min	90 min
Partial Driver Assignment	0.75	0.50	0.50
Full Driver Assignment	0.75	0.75	0.50
Partial Driver Ordering	0.75	0.75	0.75

Table 7.3: Selected values of f for each method–budget combination.

7.2.2 Driver Reassign Perturbation

To test the effect of the driver reassign perturbation described in Section 6.5 on the overall performance, we run each method-budget combination with and without the perturbation. For simplicity, each experiment is run with a single destroy fraction of $\rho = 0.3$. This value was taken from general guidelines from Large Neighborhood Search (LNS) literature [34].

The results of this experiment did not indicate any significant difference according to the Wilcoxon or the Proportion test. However, we will still examine descriptive performance metrics to identify differences. These can be found in Table 7.4.

Setting	Pertubation	BMC	QMC	$\overline{NGB}$	BWP_4
Partial Driver Assignment					
30 min	no	32	603.0	**0.66**	0.56
	yes	**41**	**661.0**	0.68	**0.61**
60 min	no	37	597.0	0.66	0.54
	yes	**45**	**832.0**	**0.60**	**0.56**
90 min	no	36	536.0	0.59	0.52
	yes	**37**	**543.0**	**0.57**	**0.57**
Full Driver Assignment					
30 min	no	**34**	**515.0**	**0.77**	**0.60**
	yes	31	495.0	0.81	0.52
60 min	no	**36**	**600.0**	**0.63**	**0.59**
	yes	29	427.0	0.65	0.54
90 min	no	**40**	**528.0**	**0.59**	**0.56**
	yes	30	428.0	0.73	0.55
Partial Driver Ordering					
30 min	no	**40**	**1176.0**	**0.67**	0.61
	yes	31	556.0	0.83	**0.63**
60 min	no	28	**621.0**	**0.69**	**0.55**
	yes	**43**	612.0	0.76	0.52
90 min	no	28	547.0	0.71	**0.59**
	yes	**33**	**1424.0**	**0.63**	0.52

Table 7.4: Effect of perturbation for each time budget and method. Bold values indicate the better result within each pair. Results are aggregated over all hubs.

The descriptive results in Table 7.4 indicate mixed results. For the Partial Driver Assignment, the perturbation appears to yield a small but consistent advantage. Although these improvements are not statistically significant, they occur in the

majority of budgets and across multiple metrics. This suggests that, for Partial Driver Assignment, periodically destroying part of the driver decisions can help escape locally suboptimal assignment patterns.

For the Full Driver Assignment, the perturbation clearly deteriorates performance. Across all three time budgets, the non-perturbed version dominates on BMC, QMC, $\overline{NGB}$, and BWP_4. For Partial Driver Ordering, the results are less conclusive but lean slightly toward worse performance with perturbation. For this method, we therefore choose not to use the perturbation. Therefore, the perturbation will only be used for the Partial Driver Assignment in the following experiment.

7.2.3 Method Comparison

To test the proposed methods against each other and the baseline, a final experiment is conducted on the test set using the configurations found in Sections 7.2.1 and 7.2.2. We will compare the performance over all hubs and for each hub described in Table 7.1 individually.

The partial order graph generated by the Wilcoxon and Proportion test can be found in Figure 7.2.

The partial-order graphs in Figure 7.2 show that, on conflict cost, all three proposed extensions significantly outperform the baseline in the aggregated (overall) comparison for all time budgets. For 30 and 60 minutes, Partial Driver Assignment is the strongest method overall: it dominates both the baseline and the other proposed variants (with the comparison against Full Driver Assignment at 30 minutes being significant only under the proportion test). For 90 minutes, all three proposed methods dominate the baseline, but no statistically significant differences are observed among them. Hence, increased computational budget reduces the performance gap between the proposed variants, while the baseline remains consistently dominated.

The hub-specific results clarify the source of these improvements. Nijmegen, the hardest hub, drives most of the significant differences: for all time budgets, each proposed method dominates the baseline, and at 60 minutes, Partial Driver Assignment also dominates Partial Driver Ordering. This indicates that exposing driver-related decisions to the local search is particularly beneficial in larger instances where the greedy nature of the embedded heuristic most strongly limits progress. In Enkhuizen, improvements are present but less uniform across budgets: at 30 minutes, both Partial Driver Assignment and Full Driver Assignment dominate the baseline, at 60 minutes, Partial Driver Assignment dominates the baseline (with Partial Driver Ordering only weakly significant), and at 90 minutes, only Full Driver Assignment dominates the baseline. Finally, for Vlissingen, no significant differences are detected for any budget. Given the small number of instances for this hub and relatively low difficulty (Table 7.1), this suggests that the baseline already performs near the attainable conflict-cost range.

To complement the statistical comparisons, we also report the relative improvement of each method with respect to the baseline in terms of mean conflict cost.

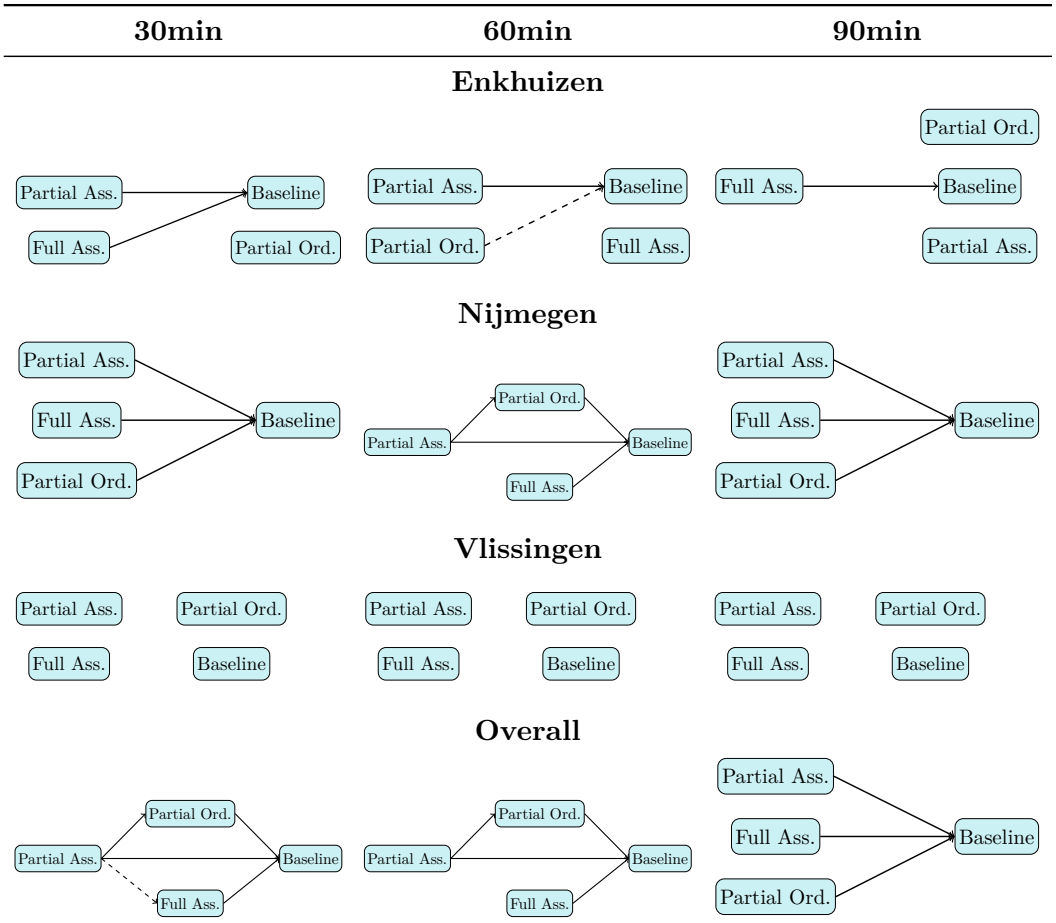


Figure 7.2: Partial ordering on conflict cost per hub and overall performance. Solid arrow indicates a significant ($p < 0.05$) difference according to at least the Wilcoxon test, and the dashed arrow indicates a significant difference only according to the proportional difference.

Let $\bar{C}_h^{\text{base}}$ denote the mean cost of the baseline on hub h, and $\bar{C}_h^m$ the mean cost of method m on that hub. The relative improvement is computed as

$$\text{Improvement}_h^m = \frac{\bar{C}_h^{\text{base}} - \bar{C}_h^m}{\bar{C}_h^{\text{base}}} \cdot 100\%.$$

Table 7.5 reports the resulting improvements averaged over all instances within each hub, as well as the overall average across hubs.

Overall, Partial Driver Assignment yields the largest improvements relative to the baseline across all time budgets, with an average reduction in conflict cost between 9.4% and 12.2%. Full Driver Assignment also consistently improves upon the baseline, although the improvements are smaller. Partial Driver Ordering exhibits a more unstable behavior across hubs and time budgets. In some cases, such as

Time	Hub	Partial Assignment	Full Assignment	Partial Ordering
30 min	Enkhuizen	19.9%	15.5%	-78.8%
	Nijmegen	7.7%	5.6%	5.2%
	Vlissingen	-3.6%	-1.7%	3.4%
	Overall	10.3%	8.1%	-25.2%
60 min	Enkhuizen	10.6%	10.8%	20.1%
	Nijmegen	12.8%	7.6%	5.6%
	Vlissingen	0.0%	-2.8%	-13.4%
	Overall	9.4%	6.8%	6.7%
90 min	Enkhuizen	17.5%	11.9%	-103.5%
	Nijmegen	12.9%	9.9%	3.6%
	Vlissingen	0.0%	0.4%	0.0%
	Overall	12.2%	8.8%	-35.4%

Table 7.5: Average relative improvement in conflict cost compared to the baseline method. Positive values indicate lower cost than the baseline.

Enkhuizen with a 60-minute budget, it achieves substantial improvements relative to the baseline, while in other settings it performs noticeably worse. This variability is also reflected in the overall averages, where strong improvements on certain instances are offset by large deteriorations on others. In particular, the extreme negative improvements observed for some settings arise when the method occasionally produces schedules with substantially higher conflict costs than the baseline. Since the reported values are averages, such outliers can strongly influence the resulting percentages. This contrasts with the statistical analysis, which showed that Partial Driver Ordering often outperforms the baseline in paired comparisons; however, those tests capture the consistency of improvements rather than their magnitude, whereas the average improvement metric is more sensitive to large negative deviations.

Overall, these results support the central hypothesis of this thesis: integrating (parts of) the driver scheduling subproblem into the local search state improves performance under equal time budgets, with percentual cost improvements of 6.8% to 12.2% for the assignment methods. Moreover, the strongest and most consistent gains are achieved by the simplest integration strategy (Partial Driver Assignment), which aligns with the motivation that enlarging the decision space should be done selectively to avoid unnecessary search complexity.

The penalty cost partial orders in Figure 7.3 do not mirror the conflict-cost results in terms of the ranking among the proposed methods. While conflict cost favored Partial Driver Assignment overall, penalty cost consistently favors Full Driver Assignment. The common conclusion across both objectives is that all proposed methods improve over the baseline: in the overall comparison, each proposed method dominates the baseline for all time budgets.

These results primarily reflect structural simplicity and preferences rather than

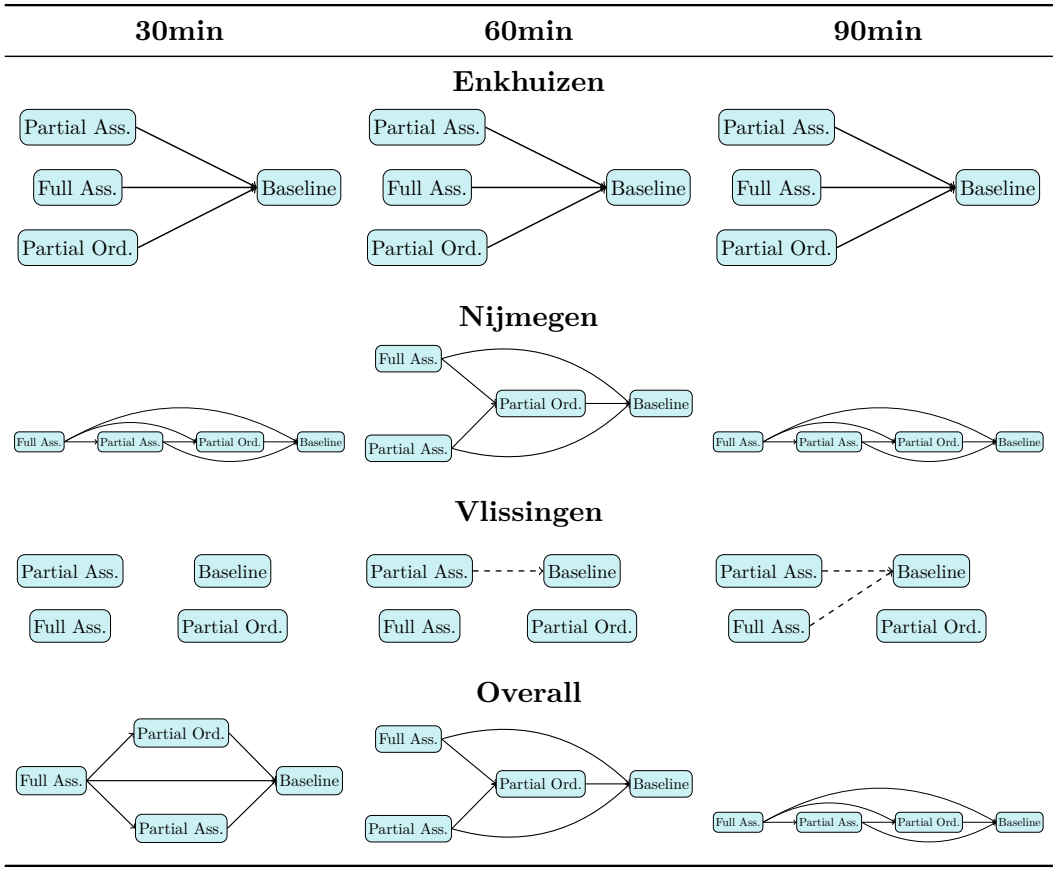


Figure 7.3: Partial ordering on penalty cost per hub and overall performance. Solid arrow indicates a significant ($p < 0.05$) difference according to at least the Wilcoxon test, and the dashed arrow indicates a significant difference only according to the proportional difference.

feasibility. In the aggregated results, Full Driver Assignment dominates Partial Driver Ordering and Partial Driver Assignment for all time budgets, indicating that explicitly integrating driver assignment decisions into the search produces solutions with lower structural penalties (e.g., fewer movements), even when this does not translate to the best conflict-cost performance. This pattern is again largely caused by Nijmegen, where Full Driver Assignment dominates both other proposed methods, and the baseline for all budgets, and Partial Driver Assignment dominates Partial Driver Ordering, which in turn dominates the baseline. In Enkhuizen, all proposed methods outperform the baseline, but no significant differences are observed among the proposed variants. In Vlissingen, only weak (proportion-test) differences appear, consistent with limited variation on this small and relatively easy hub. Similar results appear when we look at the structural and preference components of the penalty cost separately. The results of this can be found in Appendix C.

Overall, the penalty-cost analysis complements the conflict-cost results by show-

ing how the different integrations shape the solutions: Partial Driver Assignment is most effective at improving feasibility, whereas Full Driver Assignment yields structurally simpler and more preferred solutions most consistently.

Chapter 8

Discussion

This thesis shows that integrating driver scheduling decisions directly into the $\mathcal{POS}$-based local search can significantly improve solution quality for the TUSPwSPS. By reducing the reliance on an incomplete greedy scheduling heuristic, the proposed approach yields structurally better schedules and lower conflict costs across a range of realistic instances.

This chapter discusses how these findings answer the research questions stated in Chapter 1. It then reflects on the main limitations of the study and highlights several broader insights and potential risks associated with the proposed methods.

8.1 Answers to the Research Questions

This thesis began with the question of whether the current state of the art for solving the TUSPwSPS could be improved by incorporating the driver scheduling subproblem into the $\mathcal{POS}$-based local search process. In this section, this question and the other questions introduced in Chapter 1 are discussed by summarizing the main findings of this thesis.

A first step in answering the main research question is understanding the limitations of the current state-of-the-art approach. Section 4.2.1 formally proved that the List Scheduling Policy (LSP) is incomplete. It was shown that LSP can produce positive tardiness even when a zero-tardiness schedule exists. This incompleteness arises from two factors: unfavorable linearizations of the $\mathcal{POS}$ and irrevocable greedy driver assignments, which directly answer subquestion 1a.

The extent to which this incompleteness degrades overall solution quality, as addressed in subquestion 1b, was analyzed through an experiment in Section 5.5. By comparing the LSP output with an exact constraint model across different embedded subproblem variants, it was shown that the gap between the heuristic and optimal driver assignment can be substantial when no driver decisions are fixed. Where, for very small and simple instances, this gap is nearly non-existent, it increases significantly as the problem grows in size or becomes more constrained.

In the same experiment, under the explicit assumption that the local search

provides an optimal $\mathcal{POS}$ as input to the scheduling subproblem, it was shown that progressively fixing driver assignments and orderings reduces the combinatorial choices left to LSP, therefore making LSP closer to an optimal algorithm. When no assignments are fixed, LSP must decide both which driver performs an activity and in which order. When assignments are fixed (Subproblems 5.2 and 5.3) only ordering decisions remain. When per-driver orderings are fixed (Subproblem 5.4), LSP becomes optimal for that subproblem, while being considerably faster than the optimal constraint program. This answers subquestion 2a.

In Chapter 6, these subproblems were implemented by introducing local search neighborhoods: Driver Swap, Driver Switch, Driver Unassign, and Driver Activity Reorder, which can change the driver decision variables, answering subquestion 2b. This way, driver assignment and ordering decisions become direct decisions of the local search, rather than outcomes of the evaluation heuristic. The search space becomes larger due to the added driver-related variables and neighborhoods, but the evaluation of a given $\mathcal{POS}$ becomes more stable and more representative of its true scheduling potential, since fewer decisions are delegated to a suboptimal greedy procedure.

Because exposing these additional driver decisions enlarges the search space, subquestion 2c concerns how this increase in complexity can be controlled. To address this, a two-stage search was proposed in Section 6.1, in which the $\mathcal{POS}$ is optimized first, using LSP for all driver scheduling, and driver-related neighborhoods are activated only in a second stage. The experiments in Section 7.2.1 show that allocating the entire time budget either to the baseline or to the fully integrated method is never optimal. Hence, enlarging the search space only after a structurally strong $\mathcal{POS}$ has been identified leads to better performance than exposing all driver decisions from the start.

Conceptually, this approach is related to variable search space strategies discussed in Chapter 3, where metaheuristics modify the effective search space during optimization. However, the mechanism differs: methods such as Variable Neighborhood Search or variable-size metaheuristics typically alter neighborhoods or temporarily enable subsets of variables while keeping the representation fixed. In contrast, the two-stage method permanently enlarges the representation by activating additional driver-related decision variables after the first stage. However, no experiments were conducted directly comparing the proposed two-stage approach with these alternative strategies.

The driver reassign perturbation, evaluated in Section 7.2.2, showed mixed results. It yielded small but consistent improvements for Partial Driver Assignment, suggesting that periodically destroying part of the driver decisions can help escape locally suboptimal assignment patterns in that setting. In contrast, for Full Driver Assignment, performance consistently deteriorated, and for Partial Driver Ordering, results were inconclusive but slightly negative.

All these methods were evaluated in Chapter 7, directly addressing subquestion 3a. The comparisons in Section 7.2.3 show that all proposed methods significantly outperform the baseline on conflict cost across the test set. The improvements

are most pronounced on the largest hub (Nijmegen), indicating that integration is particularly beneficial when driver decisions strongly interact with shunting structure. To complement the statistical analysis, Chapter 7 also reports the average relative improvement of each method compared to the baseline. These percentages give an intuitive indication of effect size, but should be interpreted cautiously because they are sensitive to outliers: a small number of substantially worse runs can noticeably affect the mean, even when a method outperforms the baseline in most paired comparisons. Overall, Partial Driver Assignment achieved the largest average improvements (about 9-12% across time budgets), followed by Full Driver Assignment (7-9%). Partial Driver Ordering showed more variable results, with positive improvements in some settings but negative averages in others due to a few runs with substantially higher conflict costs.

Lastly, the statistical analysis of the penalty cost results shows that integrating driver assignments leads to structurally simpler and preferred solutions, and that this effect is strongest for Full Driver Assignment.

At the same time, the experiments on stage division in Section 7.2.1 demonstrate that allocating the entire budget to the fully integrated search is not optimal. This directly relates to subquestion 3b, which concerns the effect of the enlarged search space on performance under a fixed time budget. The experiment indicates that including all driver-related decisions from the start makes the search space substantially more complex to navigate, thereby reducing the local search's ability to efficiently identify high-quality regions within the available runtime.

This interpretation is reinforced by the overall performance comparison: under limited time budgets, Partial Driver Assignment consistently achieves the best results in terms of conflict cost. This shows that integrating part of the driver decision space improves performance, but that fully exposing all driver decisions does not pay off under limited time budgets. This effect disappears under sufficiently high budgets.

8.2 Limitations

This research has several limitations that should be taken into account when interpreting the results.

First, the instance set is relatively small, both in absolute size and in the number of distinct hubs. Although the selected hubs differ in scale and difficulty, the total number of instances remains limited due to practical constraints in obtaining and running realistic data. Moreover, performance was evaluated primarily on conflict and penalty costs, rather than on the number of fully feasible (zero-conflict) solutions. Ideally, feasibility would be the main evaluation criterion, since TUSPwSPS is fundamentally a satisfiability problem. However, given the limited number of instances and the fact that most instances are not solved to feasibility by any method within the tested time budgets, a feasibility-based comparison would require a substantially larger and more diverse instance set.

Second, the driver reassign perturbation was evaluated using only a single value for the destroy parameter ρ. The effectiveness of large-neighborhood-style perturbations can be sensitive to this parameter. Testing multiple values could provide deeper insight into the robustness and potential of this mechanism.

Third, all experiments were conducted under time budgets of 30, 60, and 90 minutes. In practice, in offline planning contexts, longer runtimes may be available. The relative performance of the more complex methods, in particular Full Driver Assignment and Partial Driver Ordering, may change under substantially larger computational budgets.

Lastly, the embedded scheduling procedure was limited to the List Scheduling Policy (LSP). Although LSP reflects the current state of the art in practice, alternative heuristics or hybrid exact–heuristic approaches were not considered. Consequently, the conclusions primarily concern the interaction between the local search and this specific heuristic.

8.3 Notable Findings and Insights

Beyond the direct answers to the research questions, several broader insights emerge from this study. These insights are not only about the specific integration of driver scheduling into the $\mathcal{POS}$-based local search, but also the more general interaction between metaheuristics and embedded heuristics in tightly coupled optimization problems.

First, it is not necessarily optimal to move all combinatorial complexity into the local search. Chapter 5 showed that fixing more decisions reduces the gap between the heuristic and an exact subproblem solution under the assumption that the metaheuristic behaves optimally. However, the experiments in Section 7.2.1 demonstrate that allocating the entire time budget to the fully integrated search is not optimal. Enlarging the search space improves evaluation accuracy, but also increases dimensionality and slows exploration. This highlights a fundamental trade-off: reducing subheuristic restrictions comes at the cost of a more complex search landscape, and beyond a certain point this trade-off becomes counterproductive under fixed time budgets.

Second, the two-stage approach provides an effective way to manage this trade-off. As shown in Section 7.2.1, intermediate allocations of the time budget consistently dominate the extreme cases. Structurally optimizing the $\mathcal{POS}$ first and only then exposing additional decision variables leads to better performance than integrating all decisions from the start. Conceptually, this can be interpreted as a progressive enlargement of the search space: the algorithm first identifies promising regions in a lower-dimensional space and subsequently refines them in a higher-dimensional one. This concept of multi-stage search could potentially be useful for other problems with tightly coupled subproblems. However, more research testing this on other problems would be required to confirm this.

Finally, the study shows that the boundary between metaheuristic search and

embedded heuristic evaluation is itself a design choice. When evaluation relies heavily on an incomplete greedy procedure, the search landscape may contain artificial local optima induced by heuristic artifacts rather than by the intrinsic problem structure. By shifting part of the decision-making responsibility into the search state, these distortions are reduced and the cost landscape becomes more representative of the underlying combinatorial structure. This insight extends beyond the specific setting studied here: in hybrid algorithms, deciding where decisions are made is a critical step in the design process.

8.4 Stakeholders and Risk Analysis

This research affects multiple stakeholders within railway operations. While the primary objective of this research is to improve solution quality for the TUSPwSPS, the practical consequences extend beyond algorithmic performance and should be considered.

This research primarily affects planners within NS who are responsible for constructing and validating shunting yard schedules. Improved optimization methods can support planners by automatically generating plans with fewer conflicts and better coordination between shunting and driver activities. As a result, planners are expected to spend less time manually repairing infeasible plans and more time focusing on improving the efficiency of operational plans. At the same time, increased automation may change the nature of the planner's role. Instead of manually constructing schedules, planners may increasingly focus on supervising, validating, and adjusting automatically generated solutions. While this shift can increase productivity, it also requires planners to trust and understand the automated planning tools.

In addition, improved planning quality may reduce the number of drivers required to execute daily operations at a yard, thereby lowering operational costs. In practice, NS faces a structural shortage of train drivers. Therefore, requiring fewer drivers at a specific yard does not imply workforce reductions. Instead, the freed capacity can be used elsewhere in the network where drivers are needed. Nevertheless, relying on tightly optimized staffing levels may reduce operational flexibility in the event of unexpected disruptions, such as delays or temporary staff unavailability.

Another important stakeholder is the development team at NS responsible for maintaining and improving the planning software used in practice. The methods proposed in this thesis introduce additional algorithmic components and decision variables into the planning system. While this slightly increases the software's technical complexity, the risks associated with it are expected to be limited, as the proposed methods extend an existing local search framework rather than introducing an entirely new system architecture. The primary benefit for the development team is that this research provides new algorithmic building blocks that can be integrated into the existing planning tools, potentially improving solution quality and expanding the capabilities of future planning systems.

Chapter 9

Conclusion

This thesis investigated how the integration of driver scheduling into a $\mathcal{POS}$-based local search framework affects the performance of solving the Train Unit Shunting Problem with Service and Personnel Scheduling (TUSPwSPS).

The starting point of this research was the observation that the current state-of-the-art approach relies on an incomplete List Scheduling Policy (LSP) for driver scheduling. It was formally proven that LSP can produce suboptimal schedules. Consequently, the local search might not accept a candidate solution, even though the underlying $\mathcal{POS}$ is structurally good.

This limitation was analyzed more systematically by studying alternative embedded subproblem variants. These variants progressively moved driver-related decisions from the heuristic evaluation step into the search space itself. The central hypothesis was that enlarging the decision space of the local search by fixing driver assignments and driver orderings would reduce the risk of misleading cost evaluations induced by LSP. Although most of these embedded subproblems remain strongly NP-hard, an empirical analysis revealed a consistent pattern. Under the explicit assumption that the local search is able to provide (near-)optimal inputs for the respective subproblem, the comparison between LSP and an exact constraint model showed that integrating more driver-related decisions into the search space systematically reduces the gap between the heuristic outcome and the optimal subproblem solution. I.e., the more structure is fixed by the local search, the more reliably LSP reflects the optimal schedule of a given $\mathcal{POS}$.

Based on these insights, concrete local search extensions corresponding to the different subproblem variants were introduced: Partial Driver Assignment, Full Driver Assignment, and Partial Driver Ordering. Each method redistributes responsibility between the local search and the embedded scheduling procedure in a controlled manner. To manage the resulting increase in search space complexity, a two-stage search strategy was proposed: the operational structure is optimized first, while keeping driver decisions implicit, and subsequently additional driver-related decision variables are activated.

The computational results confirm the main hypothesis. Across realistic instances and equal time budgets, integrating driver decisions into the local search

consistently reduces conflict cost relative to the baseline. In particular, with lower time budgets, Partial Driver Assignment yields the strongest and most consistent improvements, especially on the largest and most constrained hub, Nijmegen. This is likely because Partial Driver Assignment introduces additional flexibility while keeping the search space relatively small. As a result, the local search can correct harmful greedy driver assignments made by LSP without incurring the full combinatorial complexity associated with methods that expose more driver-related decisions. As the time budget increases, the methods with higher levels of integration gradually catch up, indicating that the richer decision space can be exploited when sufficient search time is available.

Another experiment showed that extreme allocations of the time budget to either purely structural search or fully integrated search were never optimal. Instead, the best results were obtained by dividing the budget between both stages, which supports the proposed staged enlargement of the search space.

In a broader context, this thesis also empirically studied how the boundary between a metaheuristic and an embedded heuristic subproblem influences performance. The results show that selectively moving decisions from a greedy evaluation procedure into the main search state can improve feasibility, provided that the search space expansion is carefully controlled.

In summary, this thesis demonstrates that the incompleteness of the decoupled scheduling procedure is a structural limitation of the current state-of-the-art, and that moving part of this complexity into the $\mathcal{POS}$-based local search leads to significant performance improvements. By carefully controlling the size of the extended search space through staged integration, it is possible to achieve better performance without incurring prohibitive computational overhead.

9.1 Directions for Future Research

The results of this thesis open several directions for further investigation.

First, the trade-off between integrating complexity into the metaheuristic and the resulting increase in search space deserves deeper theoretical and empirical analysis. In this thesis, driver-related decisions were progressively moved from the embedded heuristic into the local search state, and the experiments showed that selective integration can improve performance. However, the precise relationship between search space expansion, landscape structure, and convergence behavior remains largely unexplored. Future work could analyze how different levels of integration affect landscape ruggedness and whether similar benefits arise in other tightly coupled planning problems beyond the TUSPwSPS.

Second, the proposed staged search scheme could be evaluated more broadly for optimization problems that consist of multiple tightly coupled domains. In this thesis, a two-stage enlargement of the search space was implemented within a $\mathcal{POS}$-based local search framework for the TUSPwSPS, but many real-world problems combine interacting decision layers, such as routing and scheduling, assignment and

sequencing, or design and operation. Future research could investigate whether progressively enlarging the decision space, rather than exposing all decision variables at once, improves convergence and solution quality across different metaheuristics, such as simulated annealing or large neighborhood search. Furthermore, instead of restricting attention to exactly two stages, multi-stage schemes could be considered in which additional decision domains are exposed gradually over several stages. Such progressive expansion may offer finer control over the trade-off between search space size and coordination across domains, and could potentially be combined with adaptive criteria that determine when transitions between stages are beneficial.

Third, longer computational budgets should be investigated. The experiments in this thesis were conducted with limited time constraints to keep the experiments computationally tractable, even though the computational budgets might be higher in practice. These time constraints may limit the potential of the more complex methods, especially Full Driver Assignment and Partial Driver Ordering. With substantially longer runtimes, the larger search spaces introduced by these methods may become advantageous, potentially altering the relative performance ranking observed in Chapter 7.

Finally, additional integration variants that further expose driver-related decisions to the local search could be investigated. One possible extension of Full Driver Assignment would be to let the local search also determine the drivers for newly introduced activities, rather than having the LSP assign them during evaluation. This would remove the remaining reliance on the heuristic assignment step, at the cost of introducing a large number of additional decisions in the search space. Similarly, a fully integrated ordering variant could be studied, in which the local search maintains a complete activity ordering for each driver. While both extensions substantially enlarge the decision space, they may become beneficial under significantly larger computational budgets where the search has enough time to exploit the additional expressiveness.

Bibliography

[1] Aaron Berger, Nils Eberhardt, Annelot Willemijn Bosman, Henning Duwe, Holger H. Hoos, and Jan N. van Rijn. Empirical Analysis of Upper Bounds for Robustness Distributions Using Adversarial Attacks. In Yingqian Zhang, Milan Hladik, and Hossein Moosaei, editors, *Learning and Intelligent Optimization*, pages 119–134, Cham, 2026. Springer Nature Switzerland. ISBN 978-3-032-09156-7. doi: 10.1007/978-3-032-09156-7_9.

[2] Paul van den Bogaard. NS over de uitdaging van het fithouden van stilstaande treinen, April 2020. URL `https://www.spoorpro.nl/materieel/2020/04/20/ns-over-de-uitdaging-van-het-fithouden-van-stilstaande-treinen/`.

[3] R. W. van den Broek. Train Shunting and Service Scheduling: an integrated local search approach. Master's thesis, Universiteit Utrecht, 2016. URL `https://studenttheses.uu.nl/handle/20.500.12932/24118`. Accepted: 2016-09-05T17:01:11Z.

[4] European Commission. Prime Infrastructure - Report 2023, 2023. URL `https://enim-rail.eu/8th-prime_benchmarking_report/`.

[5] Daniel Delahaye, Supatcha Chaimatanan, and Marcel Mongeau. Simulated Annealing: From Basics to Applications. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, pages 1–35. Springer International Publishing, Cham, 2019. ISBN 978-3-319-91086-4. doi: 10.1007/978-3-319-91086-4_1. URL `https://doi.org/10.1007/978-3-319-91086-4_1`.

[6] Emrah Demir, Tolga Bektaş, and Gilbert Laporte. An adaptive large neighborhood search heuristic for the Pollution-Routing Problem. *European Journal of Operational Research*, 223(2):346–359, December 2012. ISSN 0377-2217. doi: 10.1016/j.ejor.2012.06.044. URL `https://www.sciencedirect.com/science/article/pii/S0377221712004997`.

[7] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2), 2024. URL https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2. tex.ark: ark:/44463/DelftBluePhase2.

[8] M. Fox and D. Long. The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research*, 20:1–59, December 2003. ISSN 1076-9757. doi: 10.1613/jair.1240. URL http://arxiv.org/abs/1106.5998. arXiv:1106.5998 [cs].

[9] Richard Freling, Ramon Lentink, Leo Kroon, and Dennis Huisman. Shunting of Passenger Train UNits in a Railway Station. *Transportation Science*, 39 (2):261–272, May 2005. ISSN 00411655. doi: 10.1287/trsc.1030.0076. URL https://repub.eur.nl/pub/14171/.

[10] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. In P. L. Hammer, E. L. Johnson, and B. H. Korte, editors, *Annals of Discrete Mathematics*, volume 5 of *Discrete Optimization II*, pages 287–326. Elsevier, January 1979. doi: 10.1016/S0167-5060(08)70356-X. URL https://www.sciencedirect.com/science/article/pii/S016750600870356X.

[11] R. Haijema, C.w. Duin, and N.m. van Dijk. Train Shunting: A Practical Heuristic Inspired by Dynamic Programming. In *Planning in Intelligent Systems*, pages 437–475. John Wiley & Sons, Ltd, 2006. ISBN 978-0-471-78126-4. doi: 10.1002/0471781266.ch16. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/0471781266.ch16. Section: 16 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/0471781266.ch16.

[12] A. Hanif Halim, I. Ismail, and Swagatam Das. Performance assessment of the metaheuristic optimization algorithms: an exhaustive review. *Artificial Intelligence Review*, 54(3):2323–2409, March 2021. ISSN 1573-7462. doi: 10.1007/s10462-020-09906-6. URL https://doi.org/10.1007/s10462-020-09906-6.

[13] Pierre Hansen and Nenad Mladenović. An Introduction to Variable Neighborhood Search. In Stefan Voß, Silvano Martello, Ibrahim H. Osman, and Catherine Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 433–458. Springer US, Boston, MA, 1999. ISBN 978-1-4615-5775-3. doi: 10.1007/978-1-4615-5775-3_30. URL https://doi.org/10.1007/978-1-4615-5775-3_30.

[14] Kim van den Houten, Léon Planken, Esteban Freydell, David M. J. Tax, and Mathijs de Weerdt. Proactive and Reactive Constraint Programming for Stochastic Project Scheduling with Maximal Time-Lags, March 2025. URL http://arxiv.org/abs/2409.09107. arXiv:2409.09107 [cs].

[15] Per Munk Jacobsen and David Pisinger. Train shunting at a workshop area. *Flexible Services and Manufacturing Journal*, 23(2):156–180, June 2011. ISSN

1936-6590. doi: 10.1007/s10696-011-9096-1. URL https://doi.org/10.1007/s10696-011-9096-1.

[16] Leo G. Kroon, Ramon M. Lentink, and Alexander Schrijver. Shunting of Passenger Train Units: An Integrated Approach. *Transportation Science*, 42(4): 436–449, November 2008. ISSN 0041-1655. doi: 10.1287/trsc.1080.0243. URL https://pubsonline.informs.org/doi/10.1287/trsc.1080.0243.

[17] Matthias König, Annelot W. Bosman, Holger H. Hoos, and Jan N. van Rijn. Critically Assessing the State of the Art in Neural Network Verification. *Journal of Machine Learning Research*, 25(12):1–53, 2024. ISSN 1533-7928. URL http://jmlr.org/papers/v25/23-0119.html.

[18] Carolina Lagos, Guillermo Guerrero, Enrique Cabrera, Stefanie Niklander, Franklin Johnson Parejas, Fernando Paredes, and Jorge Vega. A Matheuristic Approach Combining Local Search and Mathematical Programming. *Scientific Programming*, 2016:1–7, January 2016. doi: 10.1155/2016/1506084.

[19] Leon Lan and Joost Berkhout. PyJobShop: Solving scheduling problems with constraint programming in Python, February 2025. URL http://arxiv.org/abs/2502.13483. arXiv:2502.13483 [math].

[20] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of Machine Scheduling Problems. In P. L. Hammer, E. L. Johnson, B. H. Korte, and G. L. Nemhauser, editors, *Annals of Discrete Mathematics*, volume 1 of *Studies in Integer Programming*, pages 343–362. Elsevier, January 1977. doi: 10.1016/S0167-5060(08)70743-X. URL https://www.sciencedirect.com/science/article/pii/S016750600870743X.

[21] RM Lentink, PJ Fioole, LG Kroon, and C van 't Woudt. Applying Operations Research Techniques to Planning of Train Shunting. In W.M.C. van Wezel and R.J.J.M. Jorna, editors, *Planning in Intelligent Systems*, Wiley Interscience, pages 415–436. John Wiley & Sons Inc., Hoboken, 2006. ISBN 978-0-471-73427-7.

[22] Rustam Mussabayev and Ravil Mussabayev. Variable Landscape Search: A Novel Metaheuristic Paradigm for Unlocking Hidden Dimensions in Global Optimization, August 2024. URL http://arxiv.org/abs/2408.03895. arXiv:2408.03895 [math].

[23] Luuk van Nes. Planning Drivers for Shunting Yards. Master's thesis, Universiteit Utrecht, 2024. URL https://studenttheses.uu.nl/handle/20.500.12932/45880. Accepted: 2024-02-01T01:01:52Z.

[24] NOS. ProRail: het spoor begint vol te raken, August 2018. URL https://nos.nl/artikel/2246806-prorail-het-spoor-begint-vol-te-raken.

[25] Mireille Palpant, Christian Artigues, and Philippe Michelon. LSSPER: Solving the Resource-Constrained Project Scheduling Problem with Large Neighbourhood Search. *Annals of Operations Research*, 131(1):237–257, October 2004. ISSN 1572-9338. doi: 10.1023/B:ANOR.0000039521.26237.62. URL https://doi.org/10.1023/B:ANOR.0000039521.26237.62.

[26] David Pisinger and Stefan Ropke. A general heuristic for vehicle routing problems. *Computers & Operations Research*, 34(8):2403–2435, August 2007. ISSN 0305-0548. doi: 10.1016/j.cor.2005.09.012. URL https://www.sciencedirect.com/science/article/pii/S0305054805003023.

[27] David Pisinger and Stefan Ropke. Large Neighborhood Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, pages 99–127. Springer International Publishing, Cham, 2019. ISBN 978-3-319-91086-4. doi: 10.1007/978-3-319-91086-4_4. URL https://doi.org/10.1007/978-3-319-91086-4_4.

[28] Nicola Policella, Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. From precedence constraint posting to partial order schedules: A CSP approach to Robust Scheduling. *AI Commun.*, 20(3):163–180, August 2007. ISSN 0921-7126.

[29] ProRail. Recordaantal treinkilometers in nieuwe dienstregeling 2019, August 2020. URL https://www.prorail.nl/nieuws/recordaantal-treinkilometers-in-nieuwe-dienstregeling-2019.

[30] C.R. Reeves. Landscapes, operators and heuristic search. *Annals of Operations Research*, 86(0):473–490, January 1999. ISSN 1572-9338. doi: 10.1023/A:1018983524911. URL https://doi.org/10.1023/A:1018983524911.

[31] Ulrike Ritzinger and Jakob Puchinger. Hybrid Metaheuristics for Dynamic and Stochastic Vehicle Routing. In El-Ghazali Talbi, editor, *Hybrid Metaheuristics*, pages 77–95. Springer, Berlin, Heidelberg, 2013. ISBN 978-3-642-30671-6. doi: 10.1007/978-3-642-30671-6_2. URL https://doi.org/10.1007/978-3-642-30671-6_2.

[32] Stefan Ropke and David Pisinger. An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. *Transportation Science*, 40(4):455–472, November 2006. ISSN 0041-1655. doi: 10.1287/trsc.1050.0135. URL https://pubsonline.informs.org/doi/10.1287/trsc.1050.0135.

[33] Hartmut Stadtler. Linear and Mixed Integer Programming. In Hartmut Stadtler, Christoph Kilger, and Herbert Meyr, editors, *Supply Chain Management and Advanced Planning: Concepts, Models, Software, and Case Studies*, pages 525–536. Springer, Berlin, Heidelberg, 2015. ISBN 978-3-642-55309-

7. doi: 10.1007/978-3-642-55309-7_30. URL https://doi.org/10.1007/978-3-642-55309-7_30.

[34] Irumi Sugimori, Katsumi Inoue, Hidetomo Nabeshima, Torsten Schaub, Takehide Soh, Naoyuki Tamura, and Mutsunori Banbara. Large Neighborhood Prioritized Search for Combinatorial Optimization with Answer Set Programming, May 2024. URL http://arxiv.org/abs/2405.11305. arXiv:2405.11305 [cs].

[35] Kees Szabó. Scheduling mechanics on a Shunting Yard: skills, synchronization and train movements. Master's thesis, Universiteit Utrecht, 2023. URL https://studenttheses.uu.nl/handle/20.500.12932/43524. Accepted: 2023-02-09T01:01:05Z.

[36] El-Ghazali Talbi. Metaheuristics for (Variable-Size) Mixed Optimization Problems: A Unified Taxonomy and Survey, January 2024. URL http://arxiv.org/abs/2401.03880. arXiv:2401.03880 [cs].

[37] Yunhao Tang, Kunhao Zheng, Gabriel Synnaeve, and Rémi Munos. Optimizing Language Models for Inference Time Objectives using Reinforcement Learning, March 2025. URL https://arxiv.org/abs/2503.19595v2.

[38] Roel van den Broek, Han Hoogeveen, and Marjan van den Akker. Personnel Scheduling on Railway Yards. In *20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2020)*, volume 85 of *Open Access Series in Informatics (OASIcs)*, pages 12:1–12:15, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-170-2. doi: 10.4230/OASIcs.ATMOS.2020.12. URL https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.ATMOS.2020.12.

[39] Roel van den Broek, Han Hoogeveen, Marjan van den Akker, and Bob Huisman. A Local Search Algorithm for Train Unit Shunting with Service Scheduling. *Transportation Science*, 56(1):141–161, January 2022. ISSN 0041-1655. doi: 10.1287/trsc.2021.1090. URL https://pubsonline.informs.org/doi/10.1287/trsc.2021.1090.

[40] Vesselin K. Vassilev, Terence C. Fogarty, and Julian F. Miller. Smoothness, Ruggedness and Neutrality of Fitness Landscapes: from Theory to Application. In Ashish Ghosh and Shigeyoshi Tsutsui, editors, *Advances in Evolutionary Computing: Theory and Applications*, pages 3–44. Springer, Berlin, Heidelberg, 2003. ISBN 978-3-642-18965-4. doi: 10.1007/978-3-642-18965-4_1. URL https://doi.org/10.1007/978-3-642-18965-4_1.

Appendix A

Input and Output Format

This appendix describes the input and output format of the TUSPwSPS. The input consists of two parts. First, a set of constants describing the infrastructure, rolling stock, and operational parameters of the yard. Second, a scenario describing the specific trains, services, and personnel available during the planning horizon. The output of the algorithm is a detailed activity schedule along with the corresponding personnel plan.

A.1 Inputs

A.1.1 Constants

The constants describe the static properties of the shunting yard and the rolling stock that do not change between scenarios.

- **Infrastructure elements.** The yard infrastructure consists of tracks and connecting elements.
  - **Tracks**
    - length
    - start infrastructure element (Which infrastructure element this track connects to at its start.)
    - end infrastructure element (Which infrastructure element this track connects to at its end.)
    - whether parking is allowed
    - whether the track is electrified
    - whether reversal is allowed
    - whether multiple shunting units are allowed simultaneously
    - movement penalties in both directions
  - **Switches** Defined by a left unit and a right unit.

  - **Crossings** Defined by four connected units: top, right, bottom, and left.
  - **Foreign tracks** Tracks that connect the yard to the external railway network, with defined ingoing and outgoing directions.
  - **Bumpers** Terminal elements representing the end of a track.

- **Facility locations.** The tracks on which specific facilities (such as cleaning or maintenance facilities) are located.

- **Driving times.** The travel time required for train movements between connected infrastructure elements.

- **Safety margins.** Minimum time buffers required between certain infrastructure usages to ensure safe operations.

- **Walking distances.** Distances between tracks and staff areas, which determine walking times for personnel between activities.

- **Driver shift settings.**

  - buffer before the start of a shift
  - buffer after the end of a shift

- **Train unit types.** Each train unit type is defined by:

  - number of carriages
  - whether electricity is required
  - length
  - coupling duration
  - decoupling duration

- **Reversal times.** The time required to reverse train units for different unit combinations. These times may differ depending on whether one or two drivers are involved.

- **Service types.** For each train unit type, the following properties are specified:

  - allowed service locations
  - service duration
  - required facility

A.1.2 Scenario

The scenario describes the dynamic elements of the planning instance for the considered time horizon.

- **Incoming trains.** For each incoming train the following information is given:
  - arrival time
  - incoming foreign track
  - composition of train units
- **Outgoing trains.** For each outgoing train the following information is specified:
  - departure time
  - outgoing foreign track
  - required train unit types
- **Required services.** For every train unit, the services that must be performed before departure.
- **Available facilities and personnel.** For each resource the following properties are specified:
  - start time
  - end time
  - number of available units
  - resource type
- **Fixed part of the plan.** A predefined part of the schedule that must be included in every solution. This may include predetermined matchings or already scheduled actions.
- **Departure preparation requirement.** A flag indicating whether departure preparation activities must be scheduled.

A.2 Output

The output of the algorithm consists of a complete activity schedule together with a detailed personnel plan.

- **Activity schedule.** A list of activities describing the operational plan for the yard.
  - **Movement**

    - train units involved
    - path of the movement
    - assigned driver(s)
    - end orientation
    - start and end timestamps
  - **Handover** (check-in or check-out)
    - timestamp
  - **Service**
    - service type
    - facility used
    - train units involved
    - position (track)
    - start and end timestamps
  - **Parking**
    - parking position
    - start and end timestamps
  - **Composition** (split or combine)
    - composition type (split or combine)
    - groups of train units
    - assigned driver(s)
    - position (track)
    - start and end timestamps

- **Personnel planning.** A schedule for each individual staff member that specifies the activities they perform. This schedule also includes buffer times and walking times between consecutive activities.

Appendix B

Baseline Implementation Neighborhoods

This appendix shows the full list of neighborhoods used in the baseline method described in Chapter 4. The neighborhoods marked with an asterisk correspond to the original neighborhoods proposed by van den Broek et al. [39].

1. **Departure preparation driver count:** Change the required number of workers for a departure preparation task to either one or two drivers.

2. **Departure preparation moment:** Move an existing departure preparation task onto the moment and track of a nearby parking, service, arrival, or departure activity, when that spot is allowed for the preparation.

3. ***Movement:** Either shift a train movement earlier or later in the $\mathcal{POS}$, or merge the two movements from and to a parking activity. I.e, if a sequence is $a \to b, \text{park}_b, b \to c$, it becomes $a \to c$.

4. **Expanded movement shift:** Similar to the train movement shift in 3, but also considers shifting the activity further back or forward.

5. **Movement split:** Split a movement with a reversal into two separate movements by inserting a parking activity. I.e, if a sequence is $a \to b \to a$, it becomes $a \to b, \text{park}_b, b \to a$.

6. **Movement split and shift:** First, split a movement like in 5. Then shift the movement before or after the parking to earlier or later, respectively.

7. **Movement split and parking switch:** First, split a movement like in 5. Then perform a parking switch like in 8.

8. ***Parking switch:** Change the parking location or the entry side of a parking activity. I.e, if a sequence is $a \to b, \text{park}_b, b \to c$, it becomes $a \to d, \text{park}_d, d \to c$.

9. **Expanded parking switch:** Similar to parking switch in 8, but also moves adjacent activities of the parking activity to the same track.
10. **Parking swap:** Swap the parking locations of two parking activities.
11. ***Parking insert:** Insert a parking activity after a train movement activity.
12. **Parking insert and return:** Insert a parking activity after a train movement activity and return to the original track.
13. **Parking insert and return only when parking:** Like Parking insert and return, but only does this for incoming trains that have the sequence arrival, park, departure.
14. ***Service machine order:** Change the service order within a resource. I.e, if a resource performs the service activities in the order a, b, c, it can become b, a, c.
15. **Service machine swap:** Swap the resources of two service activities that have the same facility/staffing requirements.
16. ***Service machine switch:** Switch the resource of a service activity.
17. **Service train order:** Change the order of services required by a train.
18. **Split shift:** Shift a split activity to earlier or later.
19. **Split merge:** Merges two split activities by moving the train from one of the split activities to the location of the other split activity. I.e., if there are two split activities: $\{t_1, t_2, t_3\} \rightarrow \{t_1\}, \{t_2, t_3\}$ and later $\{t_2, t_3\} \rightarrow \{t_2\}, \{t_3\}$, they get merged into one split activity: $\{t_1, t_2, t_3\} \rightarrow \{t_1\}, \{t_2\}, \{t_3\}$.
20. **Split split and shift:** The opposite of the split-merge in 19. It splits a split activity into two separate split activities and moves one to earlier or later.
21. ***Matching swap:** Swap the matching of train units between two trains.

Appendix C

Extra results

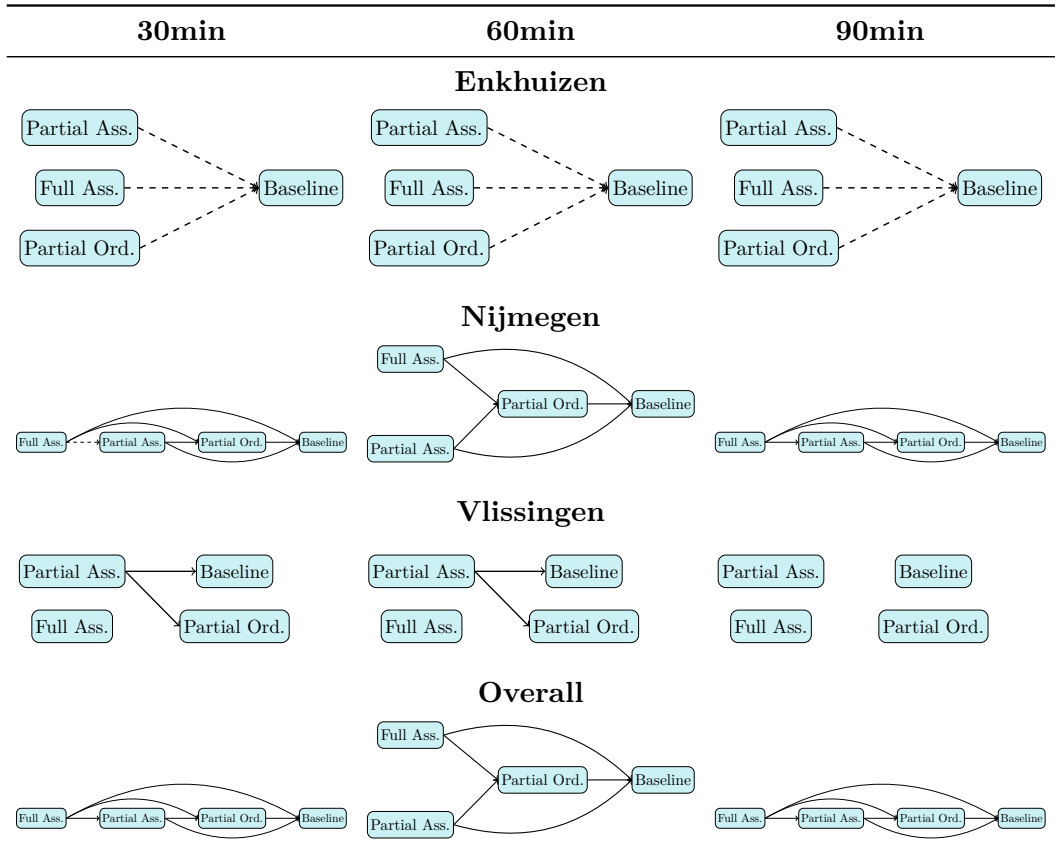


Figure C.1: Partial ordering on preference component of penalty cost per hub and overall performance. Solid arrow indicates a significant ($p < 0.05$) difference according to at least the Wilcoxon test, and the dashed arrow indicates a significant difference only according to the proportional difference.

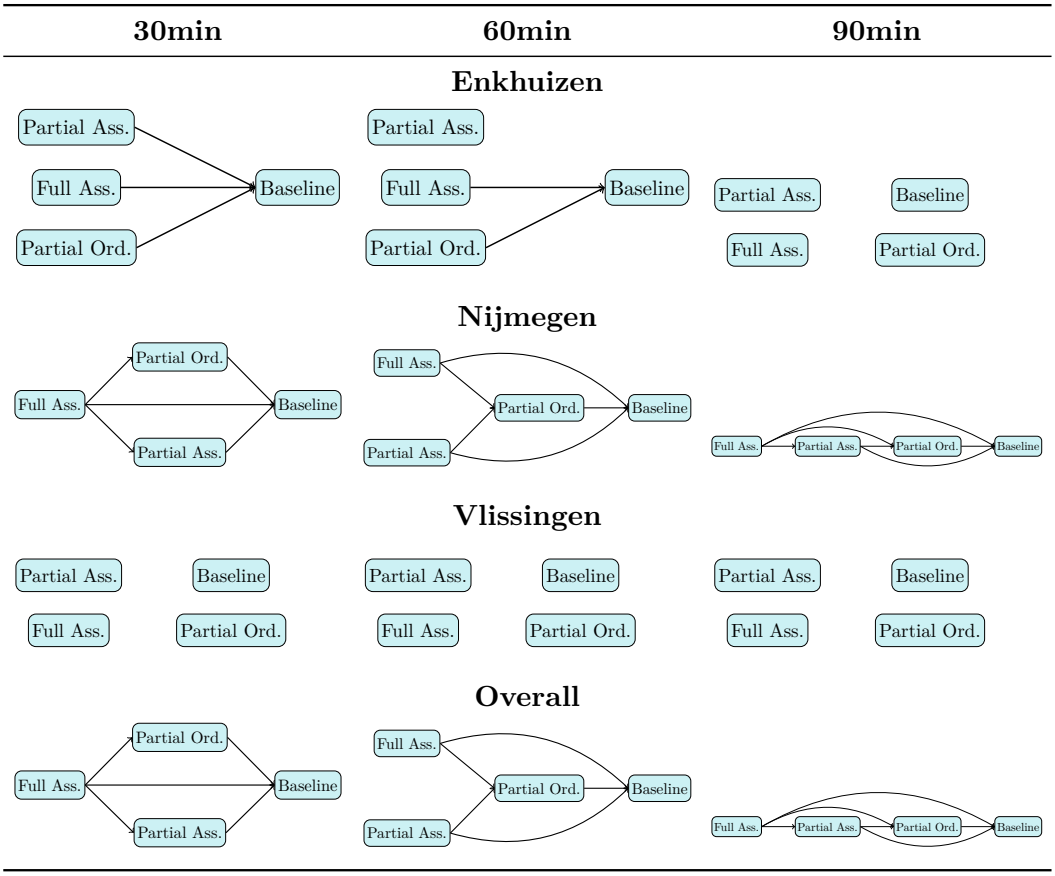


Figure C.2: Partial ordering on structural simplicity component of penalty cost per hub and overall performance. Solid arrow indicates a significant ($p < 0.05$) difference according to at least the Wilcoxon test, and the dashed arrow indicates a significant difference only according to the proportional difference.

Appendix D

Declaration of AI usage

In the creation of this thesis, generative AI tools have been used for two purposes. The first purpose is to rewrite certain parts of the text that were deemed unclear, solely to improve clarity and readability, without contributing to the formulation of ideas, hypotheses, methodological choices, or conclusions.

The second reason is to generate code for the test instance generation. Its usage is described in Appendix D.1.

D.1 Partial Order Schedule test instance generation

To generate the instances used for the small-scale experiments in Section 5.5, an LLM (ChatGPT 5.1) was used to create the code for the instance generator. All code was manually verified before usage. Below the prompt of this request can be found.

```
I have python code for instance generation. However it doesnt produce
very realistic instances.

Normally an instance has a certain flow. namely: there are x incoming
trains at specific times. These trains need to potentially be split
into subunits and potentially recomined to form other trains. They
may also need some kind of service like cleaning or maintainace.

So a flow of one train should look like:
- arrive activity of length 0, release time is arrival time
- perform activities: either clean, maintenance, park, move train,
split, or combine. These have a set length and no release/deadline
times. But they should happen after the arrival and before the
departure.
- depart activity of length: 0, release time: departure time,
deadline: departure time
```

also:
- there should be precendence constraints between the activities. precedence constraints happen when: they use the same train unit (i.e. a train unit needs to be cleaned before departure, or split before cleaned), or when they use the same track (i.e. one train needs to move out of the way befoe another can move onto it.).
- a chain of activities presedence constraints should be split when: a train is split into multiple units (produces a new independent flow for each unit.) and should be merged when two trains are merged.
- make sure walking times are always semetric.
- make sure the location of activities make sense, i.e. if there is a cleaning activity on track a, a train needs to have been moved to that track.

I also attached my thesis where the theory is described if you need it. It is fine if you completely rewrite the generation code. Please ask questions if anything is ambigious.