# MASTERARBEIT

Titel der Masterarbeit

## A Large Neighborhood Search Metaheuristic for the Personal Planning Problem

Verfasst von

Piotr Matl, BSc

angestrebter akademischer Grad

Master of Science (MSc)

Wien, 2014

| | |
|---|---|
| Studienkennzahl lt. Studienblatt: | A 066 914 |
| Studienrichtung lt. Studienblatt: | Masterstudium Internationale Betriebswirtschaft |
| Betreut von: | O.Univ.Prof. Dipl-Ing. Dr. Richard F. Hartl |

# Acknowledgements

I would like to thank O.Univ.Prof. Dipl.-Ing. Dr. Richard Hartl for giving me the opportunity to write this thesis at the Chair of Production and Operations Management at the University of Vienna. Working on this project has been a valuable experience and would otherwise not have been possible. I am also very grateful to Dr. Fabien Tricoire for offering me the chance to work on this topic, and for his constructive feedback on the algorithm implementation and for reviewing this thesis.

My thanks also go out to Dr. Pamela Nolz and Dr. Ulrike Ritzinger from the Austrian Institute of Technology for their reliable support during all phases of this project, as well as to DI Markus Straub for his effective technical assistance. Great teamwork made this adventure that much more enjoyable. The financial support from the AIT is also gratefully acknowledged.

Last but not least, my caring parents. It just would not have been the same experience without them asking every week: "*So are you finally done with that project? When are you visiting?!*"

# Contents

**List of Figures** **iii**

**List of Tables** **v**

**List of Algorithms** **vii**

**1 Introduction** **1**

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Problem Description . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.3 Related Research . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**2 The Personal Planning Problem** **7**

2.1 Input Data and Pre-Processing . . . . . . . . . . . . . . . . . . . . . 7

2.2 Mathematical Model . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.2.1 Decision Variables . . . . . . . . . . . . . . . . . . . . . . . . 9

2.2.2 Objective Functions . . . . . . . . . . . . . . . . . . . . . . . 9

2.2.3 Constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.3 Selecting an Appropriate Metaheuristic . . . . . . . . . . . . . . . . 13

2.3.1 Defining Characteristics of the PPP . . . . . . . . . . . . . . . 13

2.3.2 Challenges for Local Search Methods . . . . . . . . . . . . . . 14

2.3.3 Challenges for Evolutionary Methods . . . . . . . . . . . . . . 15

**3 Large Neighborhood Search** **17**

3.1 General Framework and Extensions . . . . . . . . . . . . . . . . . . 17

3.2 Recent Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

3.2.1 LNS with Exact Methods . . . . . . . . . . . . . . . . . . . . 19

3.2.2 Purely Heuristic LNS . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2.3 Multi-Objective LNS . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.3 Advantages of LNS for Solving the PPP . . . . . . . . . . . . . . . . . . 23
3.3.1 Reduced Need for Complex Feasibility Evaluations . . . . . . . 24
3.3.2 Navigation Across Disconnected Solution Spaces . . . . . . . . 25
3.3.3 Synergy with Generating an Approximation Front . . . . . . . 26

**4 The Proposed Metaheuristic 27**
4.1 High-Level Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.1.1 Exploration Phase . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.1.2 Consolidation Phase . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.1.3 Refinement Phase . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.2 Medium-Level Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.2.1 Destroy Operators . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.2.2 Repair Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.2.3 Efficiency Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.3 Low-Level Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.3.1 Timing Update . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.3.2 Slack Update . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.3.3 Insertion Feasibility Evaluation . . . . . . . . . . . . . . . . . . 47

**5 Computational Experiments 51**
5.1 Data Sets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.2 Parameter Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.3 Instances with Exact Reference Sets . . . . . . . . . . . . . . . . . . . . 54
5.4 Instances with Approximate Reference Sets . . . . . . . . . . . . . . . . 57

**6 Conclusion 63**

**Bibliography 65**

**Abstract (English) 69**

**Abstract (German) 70**

**Curriculum Vitae 71**

# List of Figures


1 Processing Tasks and Locations into Visits . . . . . . . . . . . . . . . . . . 8

2 A Typical Approximation Set for a PPP Instance . . . . . . . . . . . . . . 28

3 Overview of Destroy Operator Characteristics . . . . . . . . . . . . . . . 32

4 Overview of Repair Operator Characteristics . . . . . . . . . . . . . . . . 36

5 Example Timing Update (1/4) . . . . . . . . . . . . . . . . . . . . . . . . . 41

6 Example Timing Update (2/4) . . . . . . . . . . . . . . . . . . . . . . . . . 42

7 Example Timing Update (3/4) . . . . . . . . . . . . . . . . . . . . . . . . . 42

8 Example Timing Update (4/4) . . . . . . . . . . . . . . . . . . . . . . . . . 43

9 Example Insertion (1/3) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

10 Example Insertion (2/3) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

11 Example Insertion (3/3) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

12 Approximation Set of Run with Worst Hypervolume % . . . . . . . . . 56

13 Approximation Set of Run with Worst Epsilon Indicator . . . . . . . . . 57

14 Average Front Size over 10 Runs . . . . . . . . . . . . . . . . . . . . . . . 58

15 Average Run Time over 10 Runs, in Seconds . . . . . . . . . . . . . . . . 60

16 Hypervolume Growth by Search Progress . . . . . . . . . . . . . . . . . . 61

# List of Tables


1 Parameters of the Proposed Metaheuristic . . . . . . . . . . . . . . . . . 52

2 Default Parameter Values . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

3 Results for Benchmark Instances (10 Runs Each) . . . . . . . . . . . . . 55

4 Summary of Results on Benchmark Instances . . . . . . . . . . . . . . . 55

5 Modified Parameter Values for Extended Runs . . . . . . . . . . . . . . 57

6 Results for Larger Instances (10 Runs Each) . . . . . . . . . . . . . . . . 59

7 Summary of Results on Larger Instances . . . . . . . . . . . . . . . . . . 60

# List of Algorithms

1 Large Neighborhood Search . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2 Timing Update . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3 Slack Calculation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

# Chapter 1

# Introduction

*"So much to do, so little time."*

## 1.1 Motivation

Balancing professional obligations with one's private life can be challenging. Self-employed people in particular (e.g. event managers, photographers, consultants) tend to have busy and complex schedules with multiple projects and clients, but without a clear distinction between fixed working hours and leisure time. Although this flexibility can be an advantage, realizing its full potential may be far from simple.

An intelligent schedule optimizer with built-in routing functionality can support busy people in planning their work and getting the most out of their leisure time. Determining when, where, and in what order to get things done can be overwhelming for a person when their schedule is very flexible due to the large number of options and combinations. Yet more flexible schedules have more potential for time savings. A sophisticated optimizer can help users overcome this complexity by filtering out the large number of inefficient options and presenting only a few of the best streamlined suggestions.

The development of such an optimization application was one of the aims of a research project initiated by the Austrian Research Promotion Agency. Carried out in cooperation with the Austrian Institute of Technology, this thesis presents a mathematical model for a combined scheduling and routing problem that captures the real-life challenge faced by people with complex and flexible schedules, and proposes an algorithm to efficiently solve it. The optimization algorithm has been successfully implemented and embedded within a prototype mobile application for end-users.

## 1.2 Problem Description

As described above, the problem to be solved is motivated by the challenges faced by mobile self-employed entrepreneurs. These people have a variety of tasks which they may need to perform at various locations (e.g. meetings with different clients, project work at different venues). These tasks may be subject to timing constraints (e.g. arranged appointment times, opening hours), and they may have varying levels of importance or urgency (e.g. submitting a deliverable, versus cleaning the home office). The problem can therefore be modeled as an orienteering problem with time windows (OPTW), where the nodes are the tasks and their locations, the profits at the nodes represent the importance of those tasks, and the service times measure how much time the tasks require.

Several extensions to the classic OPTW are needed to model additional real-life aspects. First, tasks may have several possible locations to choose from. For example, art supplies may be bought at a number of different shops, eating out can be done at various restaurants, and packages can be sent from any post office. Second, efficient schedules should not be short-sighted and should be based on a planning horizon longer than just one day. As a result, tasks and locations may have multiple time windows during the planning horizon. For instance, a bank may be open only on weekdays, and its opening hours may be split due to a midday break. Similarly, tasks such as having lunch at noon may have their own time windows independent of their potential locations.

In addition, sets of tasks may be connected by precedence relations. For example, the subsequent stages of a project (e.g. music composition, practice, and recording), may be done individually at any time, but not in any order. Furthermore, it may be necessary to respect a certain time delay between related tasks. For example, if the person wishes to exercise three times per week, they may wish to allocate a day for rest between each session. This can be modeled by imposing a minimum time delay constraint between the individual sessions. Similarly, a maximum time delay constraint may be added to ensure that tasks are not too far apart.

Finally, the overall "quality" of a schedule depends not only on its score in terms of the tasks planned, but also on the efficiency of their timing and routing. These two aspects are conflicting, and people may also have varying preferences with regard to the trade-off between getting more tasks done and having more free time. As a result, there exists in most cases no single unique *best* solution which maximizes both objectives simultaneously. Nonetheless, a solution can still be *better* than another, for instance if both its profit and free time values are higher thanks to a more efficient routing plan.

In general, a solution is said to *dominate* another if it is not worse in any objective, and better in at least one. A solution which cannot not dominated by any other feasible solution is *Pareto-optimal*, and it is not possible to improve any of

its objective values without deteriorating another. Similarly, a solution is *efficient* or non-dominated with respect to a reference set if it is not dominated by any of the solutions in that set. Pareto-optimal solutions are always efficient, and the set of all Pareto-optimal solutions dominates all other solutions to the problem. From this perspective, the Pareto-optimal set can be considered the optimal "answer" to a multi-objective problem.

For these reasons, a bi-objective model is proposed so that both of the above-mentioned aspects of schedule quality can be taken into account. This also allows to present the decision maker with a set of different schedules which he or she may compare and choose from.

The classic OPTW is thus extended into a bi-objective OPTW with multiple time windows per task and location, multiple locations per task, precedence relations, as well as minimum and maximum time delays between related tasks. We propose to call this extension the *Personal Planning Problem* (PPP).

## 1.3 Related Research

A large body of research has been published on the OP and its variants. A recent review of this research, including the OPTW and variants of the team orienteering problem (TOP) is presented in [43]. The OP itself can also be formulated as a traveling salesman problem with profits (TSPP), of which a slightly older review is provided by [10]. The PPP is most similar to the OPTW, with some similarity to the TOP with time windows (TOPTW) due to multiple visits to the depot (i.e. home) location. As noted by [43], not much research has been published specifically on the OPTW, but the TOP with time windows (TOPTW) is a generalization that has been given noticeable attention.

An exact solution method to orienteering problems including the TOPTW is proposed in [4]. However, due to the problem's difficulty and real-life applications most research on the TOPTW has focused on heuristic approaches [43]. [42] propose a fast and deterministic iterated local search (ILS) to solve the problem in only a few seconds. New benchmark instances are also introduced. [23] propose an ant colony optimization (ACO) algorithm which finds superior results, but requires more computation time. An enhanced version of this ACO is proposed by the same authors in [24]. Motivated by a real industrial problem, [40] present a rich extension of the OPTW with multiple periods and multiple time windows per profit point (MuPOPTW). In this case the authors focus more on solution quality and combine a variable neighborhood search (VNS) with an exact route feasibility check. The VNS of [40] also produces high quality results on the (T)OPTW benchmark instances, though at the expense of additional run time.

A number of competing solution methods for the TOPTW have recently been

published. [19] combine a greedy randomized adaptive search procedure (GRASP) algorithm with variable neighborhood descent (VND). The procedure is shown to have a low average gap to the best known solutions, and some new best solutions are identified. [20] introduce a granular VNS (GVNS) in which the efficiency of neighborhood exploration is boosted by exploiting dual information from an LP formulation of a sub-problem. The GVNS of [20] further improves the set of best known benchmark solutions.

A relatively straightforward heuristic for the TOPTW is proposed by [22]. The authors randomly explore one of the common *swap*, *move*, and *2-opt* neighborhoods, and embed this local search within a simulated annealing (SA) framework. Fast and slow versions of the algorithm are tested by changing the stopping criterion, and both variants produce new best solutions. [11] extend the ILS of [42] with a clustering mechanism to mitigate some of the algorithm's weaknesses when applied to practical tourist trip design problems. An artificial bee colony (ABC) approach is presented in [8]. Very recently, [14] combine heuristic and exact methods: the authors propose a local search with perturbation elements, embedded in a SA framework, to generate a set of neighbor solutions to the incumbent, and then solve a set packing formulation of the TOPTW based on the pool of routes among the discovered neighbor solutions. [14] find 35 new best solutions and their algorithm outperforms previous methods in terms of average performance.

The multi-constraint TOP with multiple time windows (MCTOPMTW) introduced by [39] is similar to the PPP in some respects. In this formulation, the vertices may have multiple time windows (as in the PPP), and they may have several different attributes which are subject to knapsack constraints. [39] are motivated by a tourism application where the different attributes can for instance represent entry costs (to limit total spend) or point-of-interest categories (to accommodate "max-$n$-type constraints" such as only visiting at most $n$ museums). Such attribute data could be used in the PPP to model types of tasks which are to be done regularly a certain number of times, and to ensure that each task is planned at only one of its possible visits. However, the precedence constraints and the minimum/maximum time delay constraints still remain, and these are the most complicating factors in the PPP.

Despite this large number of publications, little attention has been given to *multi-objective* formulations of the OP or its variants. This is somewhat surprising since the TSPP and OP are characterized by an inherent conflict between the profit collected and the distance traveled. However, most researchers solve these problems in a single-objective way by considering the distance/cost objective as a constraint, or sequentially solving such a single-objective problem with different limits on this constraint [10].

Nonetheless, some true multi-objective approaches have recently been proposed. [35] present a multi-objective OP (MOOP) arising in the tourism sector. The objec-

tives refer in this case to the different categories of points of interest (e.g. culture, leisure, dining), with each such point offering different degrees of benefits for each category. An ant colony optimization (ACO) algorithm and a variable neighborhood search (VNS) are developed to generate non-dominated fronts of potential solutions. Both procedures are tested for the bi-objective OP (BOOP) on benchmark instances as well as on real-life instances from the cities of Vienna, Austria and Padua, Italy. An important point of difference between the BOOP presented by [35] and the bi-objective PPP is that the objective functions in the PPP are highly correlated.

[41] proposes a general algorithmic framework (multi-directional local search, MDLS) for solving multi-objective problems including the BOOP. The non-dominated set is iteratively improved using single-objective improvement algorithms for each objective. The framework is tested on the BOOP instances of [35] and improved results are obtained.

Also motivated by tourism applications, [30] present a formulation with three objectives: minimizing distance traveled, minimizing the cost of scheduled activities, and maximizing the utility gained from those activities (a fourth objective for the deviation of time spent on activities is also introduced, but treated as a constraint). Constraints include visiting hours, lunch and dinner breaks, and preference information on the types of activities. The problem is solved using an interactive tabu search (TS) metaheuristic in which the search is guided by the decision maker's choices.

Most recently, a generic bi-objective branch-and-bound algorithm is presented by [26] and applied specifically to the TOPTW. The authors use the MDLS of [41] to compute upper bound sets, and column generation for lower bound sets. Compared to the popular $\epsilon$-constraint method, the approach of [26] has the notable advantage of returning a much more evenly spaced and representative solution set if the search is terminated early or times out.

This thesis adds to the research on the OP by explicitly considering the timing aspect as a second optimization objective besides the collected score, and introduces several scheduling constraints not yet examined in the context of the OP. The thesis is structured as follows:

Chapter 2 presents a mathematical model for the PPP, and discusses the characteristics of the problem as they relate to developing a solution procedure. A large neighborhood search (LNS) framework was chosen for this purpose - Chapter 3 describes the LNS framework and reviews recent advances and applications. Chapter 4 describes in detail the search strategy and constituent elements of the implemented metaheuristic. Since there exist no benchmark instances for the PPP, the algorithm was tested on a set of new instances developed by the Austrian Institute of Technology - Chapter 5 describes the data sets and reports the results of computational experiments. Conclusions and closing remarks are made in Chapter 6.

# Chapter 2

# The Personal Planning Problem

As with many routing problems, the PPP can be modeled on a graph. Like in the standard OP and its extensions, the tasks are performed on the nodes of the graph, which in turn represent locations on the plane. The arcs between the nodes are then the connections (distance, travel time) between adjacent tasks/locations. This section presents the mathematical formulation of the PPP, and analyzes the defining characteristics of the problem with regard to choosing an appropriate solution method.

## 2.1 Input Data and Pre-Processing

The set of locations $L$ is given. Each location has at least one time window, and a distance/travel time matrix is provided. Note that the location time windows are independent of any tasks which might be performed at these locations. The set of tasks $T$ is also provided. Tasks may be performed at one of several possible locations, but note that they cannot be *split* between locations. Each task has a set of available time windows, a given profit, and a service time.

At this point, a significant difference between the PPP and the standard OP and its variants is clear. In the PPP it is possible that two or more tasks are performed at the same location, and that the same location is visited multiple times, possibly during two or more of the location's different time windows. This leads to some complications with the mathematical model and subsequent implementation.

In order to deal with the first extension (multiple locations per task), location nodes are duplicated so that every location node is associated with exactly one task. This results in location nodes with two sets of time windows - one set for the location itself, and one set for its associated task. In order to accommodate the second extension (multiple time windows per task and location), the overlaps between every location's two sets of time windows are determined. For each overlap that is equal

to or greater than the service time of the location's allocated task, a copy of the location is created with that overlap as its only time window. Figure 1 below provides a graphical representation for one task and one of its locations. All of the task's time windows are compared with the time windows of the selected location, and all overlaps which are at least as large as the task's service time result in separate visits:

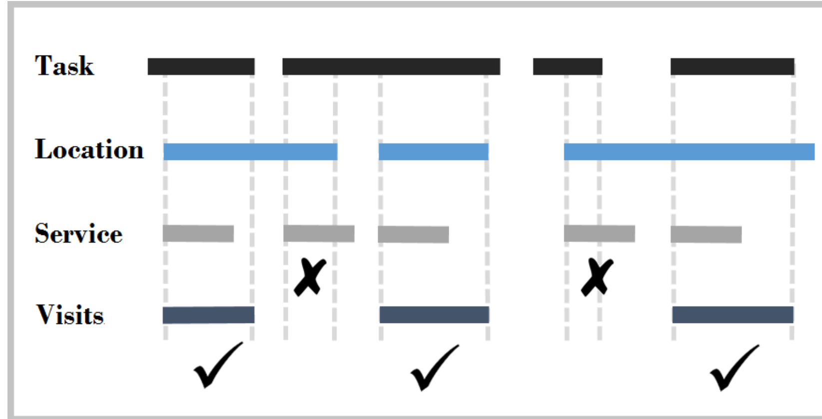


Figure 1: Processing Tasks and Locations into Visits

To avoid confusion with the original set of locations $L$, this processed set of nodes will be referred to as the set of feasible visits $V$. Since each visit maps to a unique and feasible combination of one task, one location, and one (combined) time window, it is possible to generate a distance matrix between visits and express the original problem on a graph of visits rather than locations. This leads to the following notation:

| | |
|---|---|
| $V_j$ | the set of available visits for task $j$ |
| $[o_i, c_i]$ | the (only) time window at visit $i$ |
| $p_i$ | the profit of the task associated with visit $i$ |
| $t_i$ | the service time of the task associated with visit $i$ |
| $d_{ij}$ | the travel time between the locations of visits $i$ and $j$ |

The resulting graph network significantly reduces redundancy as it contains (only) all the individually feasible combinations of tasks, locations, and time windows. Feasible routes through this network, i.e. feasible solutions to the problem, are then subject to further constraints.

The PPP introduces two more extensions: precedence relations between tasks, and minimum as well as maximum time delays between them. The following notation is added:

$\rho_i$ the set of precedents of task $i$
$\delta_i$ the set of dependents of task $i$
$\alpha_{ij}$ the minimum delay between tasks $i$ and $j$
$\beta_{ij}$ the maximum delay between tasks $i$ and $j$

The set of precedents $\rho_i$ specifies which tasks must be performed before task $i$ if task $i$ is performed. For each precedent $j \in \rho_i$, the minimum delay $\alpha_{ij}$ / maximum delay $\beta_{ij}$ specifies the minimum/maximum time that may be allotted between the end time of precedent $j$ and the start time of task $i$. With this information, it is possible to construct another set $\delta_i$ containing those tasks for which task $i$ is a precedent. These tasks are *dependent* on task $i$ being in the solution. Although not necessary for formulating the problem, this set is very useful for the solution procedure.

Finally, a set of mandatory tasks $\Omega \subset T$ is given, which holds those tasks that must be scheduled for the solution to be feasible (e.g. fixed appointments). In addition, a constant $\omega$ specifies the minimum time above which idle time (i.e. time in which no task and no travel time have been scheduled) counts as a free time slot for meaningful leisure time. Time is discretized and can take only integer values.

## 2.2 Mathematical Model

### 2.2.1 Decision Variables

In the following, it is assumed that the input data has been pre-processed as explained in the previous section. A solution to the PPP can then be specified using the following decision variables:

$y_i$ a binary variable taking the value 1 if task $i$ is scheduled, 0 otherwise
$v_i$ a binary variable taking the value 1 if visit $i$ is scheduled, 0 otherwise
$x_{ij}$ a binary variable taking the value 1 if the arc from visit $i$ to visit $j$ is used
$\sigma_i$ the time at which task $i$ is started
$\gamma_i$ the start of service at visit $i$
$w_i$ the idle time before $\gamma_i$ (does not include travel time from previous visit)
$b_i$ a binary variable taking the value 1 if there is a free time slot before visit $i$, 0 otherwise

### 2.2.2 Objective Functions

There is a clear trade-off between taking on additional tasks, and enjoying more free time. Keeping all other variables constant, every additional task added to an existing schedule reduces the total free time within that schedule, and vice versa. The PPP is

therefore a bi-objective optimization problem. Roughly speaking, the aim is to maximize the subjective "satisfaction" of squeezing in (and hopefully accomplishing) as many tasks as possible while maximizing the quality of the schedule's timing.

In more mathematical terms, the first objective corresponds to maximizing the sum of the profit scores of the scheduled tasks. This can easily be captured with the following objective function:

$$f_1 = \max \sum_{i \in T} p_i y_i \tag{1}$$

Measuring the quality of a schedule's timing is less straightforward. However, it appears reasonable to base this objective on some numerical measure relating to the schedule's periods of idle time, i.e periods during which neither service nor travel are planned. Admittedly, these periods could be seen as "free" time (positive) or as "waiting" time (negative), depending on the individual. For this reason, the following four objective functions are examined:

$$f_2 = \min \sum_{i \in V} b_i \tag{2}$$

$$f_3 = \max \sum_{i \in V} w_i \tag{3}$$

$$f_4 = \max(\min_{i \in V} w_i) \tag{4}$$

$$f_5 = \max(\max_{i \in V} w_i) \tag{5}$$

$f_2$ minimizes the number of free time slots in the schedule. Free time slots are defined as those periods of idle time $w_i$ greater than the time slot threshold $\omega$. The objective function $f_2$ is motivated by the notion that decision makers may likely prefer fewer, but longer and contiguous, periods of free time, rather than many smaller and separated breaks.

Given two identical solutions, it is possible to manipulate the total number of free time slots by shifting the visit start times forward or backward. When visits and visit order are kept constant, shifting the start times affects only the distribution of the available idle time, but not the overall total. In this way, it becomes possible to eliminate smaller periods of idle time by squeezing tasks into longer uninterrupted periods, and the freed up idle time is accumulated into fewer, but larger periods. Thus, minimizing the number of free time slots results in schedules with generally longer and uninterrupted periods of free time.

$f_3$ maximizes the total idle time available in the schedule. Unlike $f_2$, no distinction is made with regard to the distribution of this idle time. This objective function

makes no assumptions about the decision maker's preference for longer or shorter periods of free time, and in this way avoids the issue of generalizing such preferences.

$f_4$ maximizes the minimum duration among the free time slots in the schedule. The motive behind this objective function is to guarantee that if there are idle time breaks in the decision maker's schedule, they are all guaranteed to be at least of a particular duration, which is maximized by this function. Decision makers may prefer schedules with as many breaks of as long a duration as possible.

$f_5$ maximizes the maximum duration among the free time slots in the schedule. This objective function aims to ensure at least one free time slot that is as long as possible, which is in some way a stronger version of $f_2$. Like with $f_2$, extending the duration of an existing free time slot implies reducing the duration of another. However, unlike $f_2$, $f_5$ makes a distinction with regard to the distribution of the idle time among the free time slots. $f_5$ favors schedules in which one of the slots is as large as possible. Since this also minimizes the other time slots, $f_5$ brings the schedule as close as possible to having one less time slot, whereas $f_2$ only distinguishes whether this is possible at all or not for the given solution's visit order.

The proposed bi-objective model consists in general of $f_1$ as the first objective for the schedule's score, and one of $f_2$ to $f_5$ for the second objective of the schedule's timing quality. However, $f_3$ was eventually chosen as the second objective for the purposes of the practical application. In any case, the objective functions are optimized subject to the constraints presented below.

### 2.2.3 Constraints

A solution to the PPP can be modeled as a single giant tour spanning the entire planning horizon. In the following, visit 0 represents the start of the tour, and visit 1 the end of the tour. Artificial tasks and visits, e.g. sleeping time at the decision maker's home, may be added to the model and enforced by adding them to the set of mandatory tasks $\Omega$.

Constraint 6 ensures that tasks are marked as scheduled if one of their assigned visits is planned. Constraints 7 and 8 ensure that these visits are planned only if they are reached from a previous location, and then left to reach the next location, respectively.

$$\sum_{j \in V_i} v_j = y_i \qquad \forall i \in T \tag{6}$$

$$v_i = \sum_{j \in V \setminus \{1\}} x_{ji} \qquad \forall i \in V \setminus \{0\} \tag{7}$$

$$v_i = \sum_{j \in V \setminus \{0\}} x_{ij} \qquad \forall i \in V \setminus \{1\} \tag{8}$$

$$o_i v_i \leq \gamma_i \qquad \forall i \in V \tag{9}$$

$$\gamma_i + t_i \leq c_i + M(1 - z_i) \qquad \forall i \in V \tag{10}$$

$$\sigma_i = \gamma_j \qquad \forall i \in T, \forall j \in V_i \tag{11}$$

$$\gamma_i + t_i + d_{ij} x_{ij} \leq \gamma_j + M(1 - x_{ij}) \qquad \forall i, j \in V \tag{12}$$

$$w_j \leq \gamma_j - (\gamma_i + t_i + d_{ij} x_{ij}) + M(1 - x_{ij}) \qquad \forall i, j \in V \tag{13}$$

$$w_j \geq \gamma_j - (\gamma_i + t_i + d_{ij} x_{ij}) - M(1 - x_{ij}) \qquad \forall i, j \in V \tag{14}$$

$$w_j \leq z_j \cdot M \qquad \forall j \in V \tag{15}$$

$$b_j \cdot M \geq w_j - \omega \qquad \forall j \in V \setminus \{0\} \tag{16}$$

$$y_i < y_j \qquad \forall i \in T, \forall j \in \rho_i \tag{17}$$

$$\sigma_i \leq \sigma_j + t_j + \beta_{ij} + M(1 - y_i) \qquad \forall i \in T, \forall j \in \rho_i \tag{18}$$

$$\sigma_i \geq \sigma_j + t_j + \alpha_{ij} - M(1 - y_i) \qquad \forall i \in T, \forall j \in \rho_i \tag{19}$$

$$y_i = 1 \qquad \forall i \in \Omega \tag{20}$$

$$y_i \in \{0, 1\} \qquad \forall i \in T \tag{21}$$

$$z_i \in \{0, 1\} \qquad \forall i \in V \tag{22}$$

$$b_i \in \{0, 1\} \qquad \forall i \in V \tag{23}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in V \tag{24}$$

$$\sigma_i \geq 0 \qquad \forall i \in T \tag{25}$$

$$w_i \geq 0 \qquad \forall i \in V \tag{26}$$

Constraint 9 ensures that service at a scheduled visit starts only on or after the visit's opening time, and constraint 10 makes sure that the service is completed before the visit's closing time. Constraint 11 then synchronizes the start of service at the visit with the start of service of the corresponding task. Given these start times, constraint 12 maintains the time consistency so that consecutive tasks are begun only after the earlier task has been completed and enough travel time has been allotted to reach the next task on or before its planned start time. Constraints 13, 14, and 15 then set the idle times before start of service. Constraint 16, together with the minimizing objective function, handles the number of free time slots in the schedule.

Constraint 17 controls the start times of tasks linked by precedence relations such that all of a task's predecessors start before the task itself. In addition, constraints 18 and 19 enforce the maximum and minimum delays between related tasks. Finally, constraint 20 ensures that all of the mandatory tasks are scheduled, and the remaining constraints 21 to 26 define the domains of the decision variables.

## 2.3 Selecting an Appropriate Metaheuristic

At its core, the PPP is a form of a TSP with profits (TSPP). Within this broad range of problems, it falls more specifically into the category of orienteering problems (OP), in particular those with time windows (OPTW). A variety of solution methods have been proposed for the TSPP, a survey of which is provided by [10]. Likewise, different competing approaches have been developed for the OP and its variants, a recent overview of which is available in [43]. As a result, there is a wide range of seemingly promising options, and it is not immediately clear which solution strategy would be most appropriate.

This section first examines the defining characteristics for the difficulty of the PPP and the requirements relevant for the intended practical application. The suitability of some common solution methods is then evaluated based on these features in order to determine an appropriate strategy for tackling the PPP.

### 2.3.1 Defining Characteristics of the PPP

From the mathematical model presented in the previous section, it is clear that the PPP is generally a tightly constrained problem. In addition to the classic constraints on sub-tours and distances, feasibility is subject to time windows, precedence relations, minimum and maximum time delays, and a set of mandatory tasks, which themselves are also subject to these constraints. At the same time, the number of feasible visits and thus schedules may generally be larger than in other OPs, because tasks can be performed at one of several locations, and both the tasks and their locations offer several feasible time windows. PPP problem instances can thus be tightly constrained in terms of the number of constraints, but not necessarily *over*-constrained when it comes to the size of the feasible solution space.

In addition, the precedence and time delay constraints introduce a greater degree of interdependency between the elements of a PPP solution than is usual in other routing problems. For example, a 2-opt procedure at the start of a TSPP route does not affect the elements at the end of the route. Likewise, tours in a VRP or TOP can to a large extent be constructed and improved separately without affecting the feasibility or structure of the other routes in the solution. This is not the case in the PPP, where shifting a task with a tight time delay may require the shifting of a depen-

dent task either earlier and/or later in the route - and those shifts can themselves trigger further changes as well. As a result, the design of efficient feasibility evaluation of insertions, deletions, or shifts must consider the effect on the entire solution, rather than only on a local level.

The bi-objective formulation of the PPP is another decisive characteristic for the selection of a solution method. The planned commercial application of the problem is intended for a large number of unrelated decision makers. A priori information about the relative importance of the two objectives is therefore not available, making aggregate methods for transforming the problem into a single-objective formulation inappropriate. Furthermore, a varied and representative subset of efficient solutions should be available from a single run of the solution procedure. Consequently, the chosen solution method must be capable of yielding multiple efficient solutions that are more or less evenly distributed along the approximation front.

Finally, the intended application should produce these results within at most a few minutes of computation time. Although no rigorous formal proof has been presented, the PPP is a generalization of the OP, where the OP can be seen as a PPP with only single locations per task, no timing aspects, and no precedence constraints. The OP, in turn, is itself a generalization of the TSP, which is well-known to be NP-Hard. As a result, only heuristic methods are expected to provide the required trade-off between solution quality and computational time.

To conclude, the chosen solution procedure for the PPP should:

- account for a “rich” set of different types of constraints,
- handle the interdependencies between them,
- generate a varied and representative set of efficient solutions, and
- perform efficiently with limited computational resources.

### 2.3.2 Challenges for Local Search Methods

Local search methods such as Variable Neighborhood Search (VNS), Iterated Local Search (ILS), or Tabu Search (TS) are based on the notion of neighborhoods. The neighborhood $N(s)$ of a current solution $s$ is defined as all the solutions which can be created by applying a corresponding neighborhood operator on $s$. The neighborhood operator is generally designed to make minor changes to the solution so that $N(s)$ is relatively small and a complete search of the neighborhood remains computationally tractable. Examples of common neighborhood operators in the routing field include move (moves a customer to another position in the current or in a different route) and swap (exchanges the positions of two customers).

Some general guidelines for the selection and design of metaheuristics for combinatorial optimization problems are presented in [13]. The authors emphasize that methods based on local search should above all operate on a solution space *S* for which it is easy to generate feasible solutions. In particular, it should be easy to reach feasible neighbor solutions. Otherwise, the neighborhood *N(s)* will be filled with infeasible moves, and restoring feasibility to the unusable neighboring solutions would defeat the purpose of using small neighborhoods for quick evaluation.

With regard to the PPP, the presence of time windows means that simple operators modifying the order of the tasks are not likely to generate many feasible moves. This is further compounded by the precedence constraints, and the allowable time delays. Several tasks may be linked in order due to precedence relations between successive tasks in the sequence. Tasks that are part of such a precedence chain are even more restricted in where they may be moved, and the constraints on their ordering and timing place further limits on the timing options of other unrelated tasks. The more complex the underlying scheduling constraints, the unlikelier it is for basic operators to lead to admissible solutions [37]. Yet using more complex operators goes against the purpose of local search as a method for efficient neighborhood exploration.

In addition, for local search to be effective, optimal or near-optimal solutions to the problem must be reachable from most, if not all, of the solutions in *S* through paths defined by the neighborhoods [13]. This is unlikely to be possible or efficient if these neighborhoods are small and highly restricted, due to the large number of constraints as described above. It can be expected that such a local search would exhibit a strong tendency to stagnate in "isolated" local optima [17, 38]. To overcome this, penalty functions may be used to traverse the space of infeasible solutions, but as pointed out by [37], this often leads to solutions which are "slightly infeasible". This becomes more than a "slight" problem when restoring feasibility is not simple, as is the case in the PPP.

### 2.3.3 Challenges for Evolutionary Methods

Evolutionary methods, particularly genetic algorithms (GA)s, have been applied to a variety of multi-objective optimization problems, as reviewed by [45]. In general, these methods work by maintaining a population of solutions, evaluating their characteristics, and recombining their constituent elements to create new solutions. Poor solutions and their elements steadily disappear from the population, and the search converges so that the solutions are distributed around one or several optima. Since GAs operate by default on a set of solutions, they may be well-suited for also returning a set of solutions as their output [5, 45], as required in the proposed PPP.

With regard to an application for the PPP, the major problem of evolutionary

techniques would be in maintaining the feasibility of the generated solutions. GAs rely on the combination of elements from two or more solutions, and indeed, their effectiveness can be a result of combining very different solutions from a purposely diverse population [5, 45]. However, the more constrained the problem, the more care must be paid to prevent infeasibility, or to restore it if it appears. In a GA, maintaining feasibility would require a very careful design of the crossover operators. Given the number of constraints in the PPP, the operators would have to focus on foreseeing and minimizing infeasibility, while simultaneously trying to capture synergy between different solutions. Nonetheess, it is unlikely for infeasibility to be avoided completely given the number of constraints in the PPP.

Given an infeasible solution to the PPP, it is not necessarily obvious what the underlying cause is, or which fix would restore feasibility with minimal changes or at minimal cost to the given solution. For example, if a task's timing violates the time window constraint, possible fixes include scheduling it at another of its time windows, at another of its locations, deleting another task to make room in the schedule, or removing the task itself. Changing to another time window or location may simply propagate infeasibility, so removal of another task or the task itself would seem more promising in general. However, unlike in most other routing problems, even the removal itself may introduce further infeasibility in the PPP if the task to remove is mandatory, or part of a precedence chain. If it is part of a chain, then all of the subsequent tasks would have to be removed as well, and the solution begins to deviate further and further from the structure originally intended by the evolutionary algorithm's selection strategy.

In summary, an evolutionary implementation for the PPP could be expected to spend too much of its resources on ensuring or restoring feasibility, rather than actively optimizing the solution set.

# Chapter 3

# Large Neighborhood Search

After consideration of the issues discussed in Section 2.3, large neighborhood search (LNS) was selected as a promising optimization method capable of dealing with the particular challenges posed by the PPP. This chapter introduces the LNS framework, reviews some recent applications, and discusses the advantages of LNS for solving the PPP.

## 3.1 General Framework and Extensions

First introduced in 1998 by [38] and subsequently elaborated upon by [37] as *ruin and recreate*, LNS is a metaheuristic framework based on the idea of gradually improving an initial solution by destroying and subsequently repairing its constituent parts. A destroy operator deletes certain parts of the solution, as defined by the operator's selection strategy. The resulting partial solution is then reconstructed with a repair operator, which is essentially a construction algorithm. A neighborhood in an LNS is defined as a combination of a destroy and repair operator. How the solution is destroyed usually depends on some stochastic elements, and how it is reconstructed depends on which of its parts were destroyed. Consequently, the resulting neighborhood implicitly defined by the combination of the two operators is large, giving this metaheuristic its name [28].

Pseudocode for an elementary LNS implementation for a minimization problem is presented in Algorithm 1 (adapted from [28]). The LNS procedure starts with an initial feasible solution $x$. This initial solution may be created by a construction heuristic, possibly the same one used in the repair phase. Either way, line 2 sets the initial solution as the best found solution $x^{best}$. Line 3 begins the actual LNS search phase, the length of which depends on a pre-defined stopping condition. Examples for common stopping criteria are a set computation time, or a set number of iterations. An iteration of the LNS starts in line 4 with the destroy phase. The *destroy*

**Algorithm 1** Large Neighborhood Search

1: begin with a feasible solution $x$
2: $x^{best} \leftarrow x$
3: **while** *stopping criterion not met* **do**
4: $x' \leftarrow$ *destroy(x)*
5: $x' \leftarrow$ *repair(x')*
6: **if** *accept*$(x', x)$ **then**
7: $x \leftarrow x'$
8: **if** *cost(x') < cost($x^{best}$)* **then**
9: $x^{best} \leftarrow x'$
10: **return** $x^{best}$

operator is applied to the working solution $x$ to remove some elements and create a partial new solution $x'$. How much of the working solution $x$ is destroyed will usually depend on some intensity parameter. The partial new solution $x'$ is then repaired in line 5 by applying the *repair* operator. Lines 6 and 7 allow for the implementation of an acceptance or selection criterion for choosing the working solution $x$ for the next iteration. The simplest possibility here is to always accept the new solution $x'$ as the next working solution, but many other options exist. For instance, a greedy search might only continue subsequent iterations on improved solutions. In any case, the cost of the new solution $x'$ is compared with the best found solution so far $x^{best}$, and replaces the best found solution if the cost is lower. This marks the end of the iteration, and the next one begins again on line 4, and this pattern continues until the stopping criterion is reached. The best found solution $x^{best}$ is returned in Line 10 as the algorithm's output at the end of the search.

It is clear that the LNS concept is a very general metaheuristic, and that specific applications will depend greatly on the particular destroy and repair operators chosen for the problem to be solved. However, this general framework may still be extended further. One common extension is to define multiple destroy and/or repair operators, rather than just one of each. The operators should be designed so that they target different parts of the solution for destruction, and favor different reconstruction patterns for rebuilding the solution. For the repair operators, a distinction is also made between optimal and heuristic repair methods, where the optimal ones reconstruct the best possible complete solution with an exact algorithm, and the heuristic ones operate like construction heuristics [28].

In principle, LNS implementations with multiple destroy and repair operators may select the operators randomly at each iteration. However, this selection can be refined by keeping track of the success rates of individual operators or destroy/repair operator combinations, and selecting the operators in each iteration based on statistical measures of their past performance. Such LNS implementations are classified as Adaptive Large Neighborhood Search (ALNS) algorithms.

## 3.2 Recent Applications

LNS is a relatively new metaheuristic, especially compared to more established approaches like tabu search or genetic algorithms. However, a number of successful pioneering applications have recently been proposed, particularly in the routing and scheduling literature [28]. This section reviews those publications and their contributions.

### 3.2.1 LNS with Exact Methods

The seminal LNS paper by [38] proposes the application of various destroy operators (e.g. worst or related removal), combined with a single, exact repair operator based on constraint programming. [38] notes the synergy of moderate destroy operators with exact algorithms - partial solutions represent more tractable problem sizes for exact algorithms, and applying different destroy operators diversifies the search. Since then, a number of similar applications have appeared, hybridizing the LNS framework with an exact method.

[2] solve the VRPTW using a two-stage approach. The authors first minimize the number of routes using a simulated annealing heuristic, and then minimize total travel distance using an LNS combined with a heuristic branch-and-bound procedure for the repair operation. With regard to this problem class, [2] conclude that LNS is generally much better at minimizing the distance objective and relatively poor at minimizing the route objective, confirming the findings of [38]. A similar solution approach is proposed and similar conclusions are reported by the authors also for the pickup-and-delivery problem with time windows (PDPTW) [3].

[36] proposes a decomposition-based approach for solving a bus rapid transit route design problem. The problem is decomposed into the route design and route evaluation levels. The space of route designs is searched with LNS, while the route frequencies and passenger flows are determined using an LP model. Although the LNS of [36] is not adaptive as such, it uses the route characteristics of the optimal LP solutions for guiding the LNS operators. In addition, the exact approach is only applied to promising route designs, i.e. a lower bound is computed for the selected design before finding the optimal frequencies and flows.

[15] propose a general LNS combined with constraint programming for dial-a-ride problems (DARPs). A distinguishing feature of their approach is that the method does not search for the optimal reconstruction of the solution, but rather only for the first feasible one. There is thus no direct attempt at finding improving solutions with the constraint method. Instead, a simulated annealing acceptance criterion directs the search in promising directions while allowing for diversification. This general approach by [15] outperformed more specialized algorithms at the time of publication, and can be applied more easily to DARP variations, for which side-constraints

are often dependent on underlying real-life applications.

Most recently, [25] present an algorithm combining LNS with column generation, also or the DARP. Within this framework, the LNS considers the diversification and improvement of entire solutions, while the column generation component works to improve the solution at the route level. The generation of additional columns is guided by an embedded variable neighborhood search (VNS). Although this makes the framework proposed by [25] more complex than the approach by [15] above, it finds new best solutions for almost half of the benchmark instances, and at only a third of the running time as the state-of-the-art.

### 3.2.2 Purely Heuristic LNS

Unlike the approaches above, purely heuristic LNS algorithms generally rely on multiple repair operators. The application of these operators is often controlled by an adaptive layer which favors those operators which have led to improved solutions during the search. A relatively large number of ALNS applications have been proposed in the routing literature. The flexibility and general character of the LNS concept is reflected by the variety of problem classes and application-specific extensions considered by these contributions.

[31] present a rich heuristic framework capable of modeling a variety of VRPs with backhauls. The authors implement an ALNS for a rich pickup and delivery problem with time windows (RPDPTW) and apply the same algorithm to solve the standard VRP with backhauls (VRPB), mixed vehicles (MVRPB), multiple depots (MD-MVRPB), and time windows (VRPBTW, MVRPBTW), as well as the VRP with simultaneous pick ups and deliveries (VRPSPD). New best known solutions were found for two thirds of the benchmark instances.

[27] propose a general ALNS metaheuristic for tackling several related problem classes, namely the VRPTW, the capacitated VRP (CVRP), the multi-depot VRP (MD-VRP), the site-dependent VRP (SDVRP), and the open VRP (OVRP). As in [31], the authors transform these five problem types into a RPDPTW which is then solved with an ALNS. It is notable that this ALNS obtains competitive results for all examined classes except the CRVP, despite the lack of problem-specific tuning of the operators or of the adaptive layer parameters.

Other extensions of the VRP have also been considered. [12] develop an ALNS for the 2-Echelon VRP (2E-VRP). The problem consists in optimizing a two-level distribution system of goods from a central depot to a number of given satellite facilities, and from these facilities to the customers. Based on the hierarchical structure of the problem, the authors propose also a two-level approach for the destroy operators. Some destroy operators manipulate the solution structure at the satellite level (large impact), while others focus on changes at the customer level (small impact).

In addition, [12] incorporate a strong local search phase after the repair operation for promising solutions, by exploring the *split*, *move*, *swap*, *2-opt*, and *2-opt** neighborhoods with a first improvement strategy. New best solutions for more than half of the 2E-VRP benchmark instances are found.

A similar application is described by [1], who develop an ALNS for the VRPTW with multiple routes (VRPMTW). In this problem, each vehicle may complete multiple tours during the operational day. Similar to [12], the authors take advantage of the hierarchical nature of the VRPTW and propose a similar multi-level approach to the destroy operators, removing either workdays, routes, or individual customers, either randomly or according to a relatedness measure. Their results indicate that the multi-level scheme markedly improves the percentage of served customers in the solutions compared to only a customer-based approach.

[9] present an extension of the VRPTW focusing on green logistics, called the pollution routing problem (PRP). The objective is similar to the classical VRPTW, but the objective is a complex function of fuel consumption, emissions, and driver costs. The authors solve the problem in two phases, optimizing first the classical VRPTW, and then the PRP objective for the given VRPTW solution. Interestingly, this application combines an ALNS with an exact procedure - the ALNS solves the underlying VRPTW, and a polynomial time speed optimization algorithm improves the final objective function.

Like the VRPTW, the PDPTW is often optimized in the same two phases, by minimizing first the number of routes, and then the total distance. [32] apply an ALNS to this problem, but unlike the contributions described above, they apply the same ALNS to both stages of the search and obtain competitive results. The authors introduce additional diversification by adding noise to the repair operators' objective functions, and by using a simulated annealing acceptance criterion within their LNS.

Motivated by a real problem faced by a commercial air carrier, [29] consider an extension of the PDPTW where transhipment of loads from one vehicle to another is permitted (PDPT). Similar to other authors, [29] divide the search into two phases, using a GRASP procedure to generate a set of diverse and promising initial solutions, which are then improved with an ALNS. The authors extend the adaptive nature of their ALNS to the destroy operation as well: rather than choosing the degree of destruction randomly at each iteration, this factor is reset to a minimum every time an improved solution is found, and increases steadily until the next improvement. This has the effect of automatically intensifying the search when a promising solution is found, and diversifying otherwise.

[7] consider a pickup and delivery problem with multiple stacks arising in the distribution of pallets or containers of goods. Items picked up by the vehicle must be placed in one of a limited number of stacks in the vehicle, and delivery can be per-

formed only on a LIFO basis. The authors introduce stack-specific destroy and repair operators, including a regret-based insertion strategy for multiple stacks. Computational experiments show that the LNS of [7] scales better than a competing VNS algorithm when the instance size grows.

Highly competitive results have also been published recently by [18] for the team orienteering problem (TOP). Similar to [12], the authors apply a local search phase for further intensification after the LNS repair operation. Their LNS is not adaptive, but [18] explore the *move*, *swap*, *2-opt*, and *Or-opt* neighborhoods, as well as a *swap* neighborhood using the pool of unrouted stops. New best known solutions are found for all but one of the 387 benchmark instances, within comparable computation times.

Some researchers have used ALNS to handle very rich routing applications with a wide variety of constraints. [6] present an ALNS for the technician and task scheduling problem (TTSP). Their implementation handles a variety of different constraints, such as technician skill levels, precedent and successor relationships between the tasks, outsourcing options, and team building. [16] build on this work and consider the TTSP with routing aspects (STRSP), as well as with and without team building. A novel feature of the ALNS proposed by [16] is that the adaptive layer operates on pairs of destroy and repair operators, rather than individually. The addition of noise to objective value calculations is handled adaptively as well. Like [6], [16] also apply the proposed metaheuristic on real-life cases with additional constraints such as lunch breaks and working hours regulations, with promising results.

Another rich application, a rollon-rolloff waste collection VRPTW, was solved by [44]. The problem consists in routing containers from industrial and retail sites to disposal facilities, container storage yards, and depots. The authors consider a variety of complicating factors, such as multiple disposal facilities, multiple storage yards, different container types and sizes, drivers' lunch breaks, and different work schedules. Despite the many constraints, [44] find feasible solutions and generate considerable improvements compared to the industrial partner's previous practices.

### 3.2.3 Multi-Objective LNS

There are comparably fewer LNS contributions taking multiple objectives into account. Indeed, evolutionary algorithms appear to be one of the more popular heuristic multi-objective approaches. [45] provide a recent review of the state-of-the-art for this stream of multi-objective optimization.

Nonetheless, a general theoretical framework for a multi-objective LNS (MO-LNS) has recently been proposed by [34]. The authors combine LNS with constraint programming (CP), taking advantage of the synergies first pointed out in the seminal LNS paper by [38]. Solutions are selected from the archive of non-dominated solu-

tions, relaxed, and solved with a CP solver. In order to prevent clustering around one part of the approximation front, solutions are not selected randomly from a simple list of the archive. Instead, [34] propose to select a uniform random point on the hyperplane between the extreme solutions of the archive, and choose the archived solution closest to this point as the solution used in the next iteration.

In order to find a set of non-dominated solutions, the search is steered in different directions by dynamically changing a *filtering* constraint for each objective: *no* filter allows the objective to take any value, a *weak* filter updates the upper bound on the objective such that a new solution must have an equal or better value, and a *strong* filter updates the objective's upper bound so that a new solution must strictly improve this upper bound to be accepted. Various levels of intensification can then be achieved by changing the strength of the filters set for each objective. [34] apply diversification by turning off all filters and activating a *Pareto* constraint on the domain of each objective such that a non-dominated solution is produced by the constraint solver. Initial experiments on benchmarks for the multi-objective quadratic assignment (MO-QAP) problem, the multi-objective binary knapsack problem, and a bi-objective tank allocation problem yield competitive results.

When it comes to multi-objective LNS for real-life applications, an example can be found in [21]. The authors consider a DARP arising in healthcare logistics, where the interests of multiple stakeholders must be taken into account. [21] combine five different objective functions within a multi-criteria decision making framework, and solve the DARP with an ALNS. A distinguishing feature of the LNS by [21] is that three sets of destroy and repair operators are considered: basic ones from the literature, advanced ones exploiting some structure in the specific application, and parameterized ones specifically designed around the five objective functions to be optimized. Although no configuration is found to dominate any other, this can be a useful planning framework for designing meaningful destroy and repair operators in a multi-objective context.

## 3.3 Advantages of LNS for Solving the PPP

The LNS framework offers three significant advantages for tackling the PPP:

1. it requires only the most basic feasibility checks related to simple insertion steps,
2. it is able to conduct a broad search of the tightly constrained and potentially discontinuous PPP solution space, and
3. it provides a built-in mechanism for finding efficient solutions from different segments of the bi-objective approximation set.

### 3.3.1 Reduced Need for Complex Feasibility Evaluations

In order to better understand Points 1 and 2, it is necessary to take a closer look at the concept of infeasibility. For any problem, many kinds of constraint violations may exist. Let $V$ be the set of all such violations for a given problem. If a solution contains *any* violation $v \in V$, then this renders the solution *infeasible*.

Consider problems with at least one set of constraints stating that some, but not all, elements of the problem must be included in a solution for it to be feasible. When constructing an initial solution to such a problem, individual elements are added one at a time, such that a sequence of partial, *incomplete* solutions are created that lead up to the first feasible solution state. It is clear that all of the *incomplete* solutions are *infeasible*, because they violate some constraint on the mandatory elements. However, the inverse is generally not true: not all *infeasible* solutions are *incomplete*.

Hence, there exists a subset of violations $C \subseteq V$ which determine whether a given solution is *incomplete*, and its complement set $F$ of all other violations. If a solution contains violations *only* from $C$ and *none* from $F$, it will be referred to as *weakly infeasible*, and this type of infeasibility will be referred to as *weak infeasibility*. All other cases will be referred to as *strongly infeasible* with *strong infeasibility*. In this context, it is shown that the basic LNS framework enables, by its design, to set a clear limit on the degree and type of infeasibility encountered during the search process.

Consider first the destroy operator. At the most basic level, it can only *remove* elements. Since the search begins with a feasible solution, applying a destroy operator can therefore introduce *only* weak infeasibility into the working solution, if any at all. The actual degree of any potential infeasibility is further limited by the destroy operator's intensity parameter, which can be set as desired.

In the specific context of the PPP, only two constraint violations qualify for weak feasibility: the lack of a mandatory task from the set $\Omega$, and the lack of a required precedent from a scheduled task $i$'s set of precedents $\rho_i$. In both cases, and assuming that solutions are restored to feasibility immediately upon being destroyed, at least one feasible reinsertion is always guaranteed to exist after the destroy operation, namely the position before the operation. Thus, correcting weak infeasibility is simply a matter of reinserting the missing tasks just like any others, and thus does not require any specialized infeasibility handling. At the same time, it may be possible to restore the removed mandatory elements to more efficient positions in the current state of the partial solution. This leads us to the repair operator.

In principle, a repair operator should only *insert* elements into the solution. Repair operators are essentially construction procedures, and as such they require only the minimum feasibility evaluations required for performing feasible insertions. In contrast, local search moves such as *move* or *swap* already require either more so-

phisticated feasibility checks (even a simple *move* operator is already equivalent to a combination of one destroy and one repair operation), or special feasibility restoration procedures.

Since the initial solution is feasible and the destroy operator can introduce only weak infeasibility as detailed above, the repair operator can be the only source of strong infeasibility in an LNS optimization procedure. Whether it is introduced or not depends wholly on the intended search strategy implemented by the algorithm designer. Merely inserting additional elements cannot correct strong infeasibility if it is present, as this type of violation is caused by a conflict between two or more existing elements, rather than by their lack. Hence, any strong infeasibility permitted at any point will require some dedicated feasibility restoration procedure elsewhere. However, allowing a search through the infeasible solution space is likely to be a redundant strategy when using an LNS (elaborated in the next section).

As a result, there is little motivation for deviating from pure construction-based repair operators performing only feasible insertions. An LNS thus requires only the elemental feasibility evaluations required for the simplest insertion operations, as stated in Point 1. This is clearly an advantage for tightly constrained problems such as the PPP, in which more involved feasibility checks or corrections may be too ambiguous, more time-consuming, or even more complicated than just generating a new solution. In fact, no infeasibility correction procedures are required unless the implementation explicitly allows and plans to explore the infeasible solution space.

### 3.3.2 Navigation Across Disconnected Solution Spaces

Complex problems with many constraints like the PPP are often characterized by a "disconnected" solution space or "discontinuous, uneven" objective function landscapes. In such cases, local search procedures limited to only feasible moves are likely to be trapped in "isolated" sections of the solution space, leading to premature convergence at a possibly weak local optimum [17, 37]. Exploration of the infeasible solution space is a possible strategy to overcome these difficulties. In this context, limiting the degree of allowable infeasibility, as described in the previous section, could very well be counterproductive.

Fortunately, the ability of an LNS algorithm to navigate such "problematic" solution spaces is one of the framework's most commonly recognized and exploited advantages [17, 28, 37, 38].

The smaller neighborhoods applied in most local search heuristics are explicitly defined in terms of basic operators performing only small steps. These algorithms tend to have trouble with discontinuous landscapes because as the number of constraints increases, it becomes less and less likely that there exists a sequence composed of many such small steps that links two isolated solution spaces [17]. Even if

such a sequence exists, it is likely that the objective function does not improve along the entire path, and therefore the search might be steered away from this direction unless a sufficient, possibly large number of deteriorating moves are permitted.

In contrast, an LNS can perform such jumps in one iteration, and without the need to handle the infeasible search space. The destroy operator reduces the solution to a state from which different parts of the solution space may still be reached, since the solution is only partially complete. The repair operator then determines in which direction to reconstruct the solution. The combination of these two operators means that even a single iteration can move the search to distant parts of the solution space. The degree of this behavior can also be dynamically controlled by the destroy intensity parameter. It is also possible to define several destroy and repair operators so that applying different combinations of them will also have the effect of searching different distant parts of the solution space.

It is likely that the number and types of constraints in the PPP could result in the kind of disconnected or uneven solution spaces described above. As a result, the LNS framework appears to be a promising option for achieving a sufficiently diverse exploration of the PPP solution space in a simple and modular way.

### 3.3.3 Synergy with Generating an Approximation Front

Finally, the LNS framework happens to be very conducive to filling out the approximation set for the PPP. In this particular problem, the relationship between profit and available free time is strongly related to how "empty" or "full" a solution is: the more profit is collected, the less free time there is, and vice versa. As a result, solutions with low profit/more free time will tend to be "emptier" in the sense of having fewer tasks, while solutions at the other end of the front with high profit/less free time will tend to be "fuller".

From this perspective, the LNS operations of destroy and repair generate a natural progression of solutions along the efficient front. A destroy operator makes the solution "emptier", and a repair operator gradually makes it "fuller" again. This can be taken advantage of by evaluating the efficiency of solutions after every insertion step. In this way, large parts of the objective space can be explored every time a solution is destroyed and repaired.

It should be noted, however, that this is not a general advantage of the LNS framework, but rather happens to work for the specific objective functions in the PPP, and because an actual solution set is required as the output for the particular application motivating the present work. If a problem's objective functions are not directly or indirectly related to how "full" or "empty" the solution is, then the destroy/repair concept does not necessarily offer any special advantages for the construction of an efficient set.

# Chapter 4

# The Proposed Metaheuristic

The aim of the proposed algorithm is to find a set of non-dominated solutions which approximates the true Pareto frontier of the problem. In the bi-objective case of the PPP in which both objectives are to be maximized, the approximation front will generally take on a form similar to the one shown in Figure 2, where each point represents one solution. For the purposes of this narrative, the area with lower profit and more free time will be referred to as the “upper” region of the front, and the area with higher profit and less free time as the “lower” region of the front. It is important to note that the upper region is composed of “emptier” solutions in the sense that they contain fewer tasks - this yields high free time values, and low profit. The opposite is true for the lower region - more tasks allow for a higher total profit, but each additional task decreases free time. As a consequence, the upper regions are generally easier to optimize for the selected objective functions, because there are fewer potential permutations of tasks and routes for solutions with a lower limit on their total profit.

Particular focus is placed on exploring both the upper and lower regions of the approximation set in order to generate a diverse and representative set of schedules. This is important because *a priori* preference information is not available from the decision maker, and he or she may also wish to compare different schedules without re-running the algorithm.

Another practical consideration for the intended application is that it should be possible for the decision maker to terminate the search at any time. This means that the algorithm should generate a representative and front of reasonably high quality solutions even after a very brief search. The available computational time is therefore treated as a variable, making elaborate initialization procedures unsuitable for the application. The chosen search strategy reflects these considerations.

Section 4.1 explains the overarching search strategy in which the actual LNS has been embedded. In particular, the different phases of the search are explained. Sec-

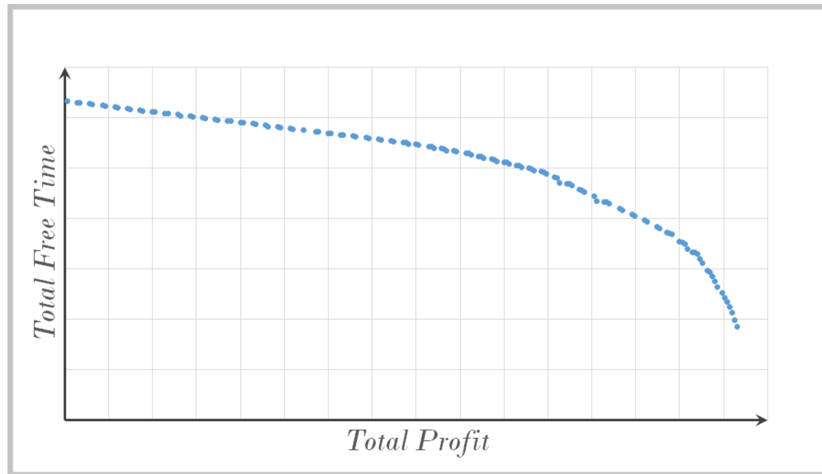


Figure 2: A Typical Approximation Set for a PPP Instance

tion 4.2 then continues with a closer look at the specific elements of the LNS itself, namely the destroy and repair operators, as well as the efficiency evaluation procedure. Section 4.3 concludes the chapter with an explanation of how some common procedures for routing problems can be adapted in order to deal with the specific constraints imposed by the PPP.

## 4.1 High-Level Layer

The proposed search strategy can be divided into three phases:

| | | |
|---|---|---|
| **A** | **exploration** | *an initial approximation set is constructed* |
| **B** | **consolidation** | *formerly efficient solutions are re-examined for local optima* |
| **C** | **refinement** | *individual elements of the approximation set are improved* |

### 4.1.1 Exploration Phase

The aim of this first phase is to explore a varied set of solutions with different underlying structure. For this purpose, the heuristic starts each iteration of this phase with the empty solution. This empty solution is developed in as many directions as possible by applying many different repair operators (presented in more detail in Section 4.2.2). Construction ends once no more insertions are possible. In this and all other phases, an iteration of the proposed algorithm starts with some solution state, and ends once no further insertions are possible.

The empty solution is first made feasible through the insertion of all the mandatory tasks. These tasks constitute to some extent a backbone to all potential solu-

tions since every feasible solution must contain them. How they are sequenced, and the visit (i.e. location and time window) chosen for each of them, can have a significant impact on the options available for the further insertion of all remaining tasks. As a result, it is important to explore as many combinations as possible of these tasks' available visits, using different repair operators. This is the easiest and most flexible point in the search to do so, since the solution is still empty.

Finding an initial feasible solution may not be easy if the set of mandatory tasks $\Omega$ is large, and if it includes many precedence constraints. In fact, it is not guaranteed that a feasible initial solution even exists for the input data specified by the decision maker. In cases when no feasible initial solution can be found within a certain number of attempts, the input data is relaxed by transforming all mandatory tasks in $\Omega$ into optional ones, and setting their profit scores to a very large value to ensure that as many as possible are included during later insertion operations.

Once the empty solution has taken on a feasible initial state, a randomly selected repair operator is iteratively applied to build the solution one insertion at a time. The efficiency of the current solution is evaluated *after every insertion*, and the approximation set is updated accordingly if the new solution is efficient. If the solution is in fact efficient, it is also copied to a stack for re-examination during the next phase. This stack keeps track of all the solutions which were efficient at some point, even if they become dominated by other solutions in subsequent iterations.

The exploration phase is concluded after a predetermined number of iterations.

### 4.1.2 Consolidation Phase

The aim of the consolidation phase is to more thoroughly examine all the solutions saved in the stack and see if any improvements can still be made from them. Recall that all of these solutions must have been efficient at some point in the search in order to be added to the stack. It is thus possible that they may still possess some good features, even if they have since become dominated. The consolidation phase applies the LNS to these solutions to search for local optima around them.

Each iteration consists of taking a solution from the stack, applying a destroy operator, and then rebuilding it with a repair operator until no further insertions are possible. As before, the efficiency of intermediate solutions is examined after every insertion. Newly discovered efficient solutions are still added to the stack as before. As a result, promising solutions are immediately explored further in the next iteration since they will have been placed on top of the stack.

In principle, all solutions from the stack may be considered. However, experiments showed that it becomes increasingly unlikely and time-consuming for solutions lower in the stack to yield new efficient solutions. Hence, in addition to the LIFO policy used for stacks, a stopping criterion is also proposed which breaks the

search of the stack after a predefined number of solutions have not produced any improvements of the approximation set. Either way, the stack is empty at the end of this phase.

If the exploration phase was thorough, then the consolidation phase does not make many improvements, and as a result does not take much time, because no new solutions get added to the stack. On the other hand, if the exploration phase was weak, this phase provides the chance to make improvements by considering the previously efficient solutions one more time. In this way, the consolidation phase ensures that the approximation set is sufficiently diverse before the search focuses more on intensification.

### 4.1.3 Refinement Phase

This is the final phase of the search, and it aims at intensifying the search around the specific solutions in the approximation set and fine-tuning the routing or timing. At this point, only the solutions in the approximation set are considered for further improvement, and for this reason it is important to develop a representative front during the previous phases.

The working solution at the start of each iteration is taken from the approximation set. The solutions of the approximation set are considered in order, sweeping from one end of the front to the other, examining each solution along the way. The neighborhood around each solution is explored by applying a destroy operator and then reconstructing with the repair operators as before.

The solution stack is empty at the start of this phase. However, new efficient solutions are still added to it whenever discovered. The selection of the next working solution at the start of each iteration gives priority to any solutions which may have been added to the stack. As in the consolidation phase, this has the effect of intensifying the search around promising parts of the front before moving on to the next solution.

Once one end of the front is reached, the search restarts again at the other end. This sweep procedure is repeated until the search is terminated manually or another stopping condition is reached. For the practical application, a limit on the total number of sweeps as well as a limit on the run time were used as stopping conditions, but other criteria, e.g. a certain number of iterations without improvement, can also be implemented. Finally, the solutions of the approximation set are returned as the output of the algorithm. Since some approximation sets may be very large (over 100 solutions), a post-processing procedure may be applied to reduce the number of solutions in the output. For the practical application, the front is divided into $n$ regions each with an equal number of solutions, and from each region the solution with the highest profit is returned.

## 4.2 Medium-Level Layer

This section goes into detail on the proposed destroy and repair operators, as well as on the procedure for evaluating solution efficiency in the bi-objective case.

One factor influencing the design of the embedded LNS was the desire to minimize the number and influence of parameters. For this reason, the destroy and repair operators were tentatively set to be selected randomly and with equal probability. As computational tests suggested this to be a sufficient configuration, further adaptive elements were not added.

However, the framework presented here for determining the particular characteristics of individual operators was specifically designed for a comprehensive statistical evaluation and adaptive implementation. It is a fairly novel framework in that it does not require explicitly predefining every possible operator, but rather only a set of combinable characteristics. The set of actual operators is then implicit as the set of combinations of these characteristics.

### 4.2.1 Destroy Operators

One thing all destroy operators have in common is that they result in the removal of some parts of the solution. They differ from one another in terms of what kind of parts they remove, why specific parts are chosen over others, and how much of the solution is destroyed. A specific instance of a destroy operator can be sufficiently defined by specifying these three characteristics. The destroy operators used in the proposed metaheuristic are thus determined by the following properties:

| | | |
|---|---|---|
| **A** | **component** | *definition of what constitutes a "part" of the solution* |
| **B** | **criterion** | *on what basis these parts are evaluated* |
| **C** | **quantity** | *how much of the solution is to be destroyed* |

A destroy operator is built from a combination of these three properties. As stated above, in the specific practical application motivating this work, these properties are randomly combined with equal probability for all combinations of their possible states. However, the framework provides potential for a structured examination of the effectiveness of particular components at a more detailed level than that of whole destroy operators. This can be incorporated into a dedicated ALNS framework, especially if data are available over a longer period with a large number of instances, and particularly for problems characterized by a significant degree of variability in instance structure (which is where an ALNS shines).

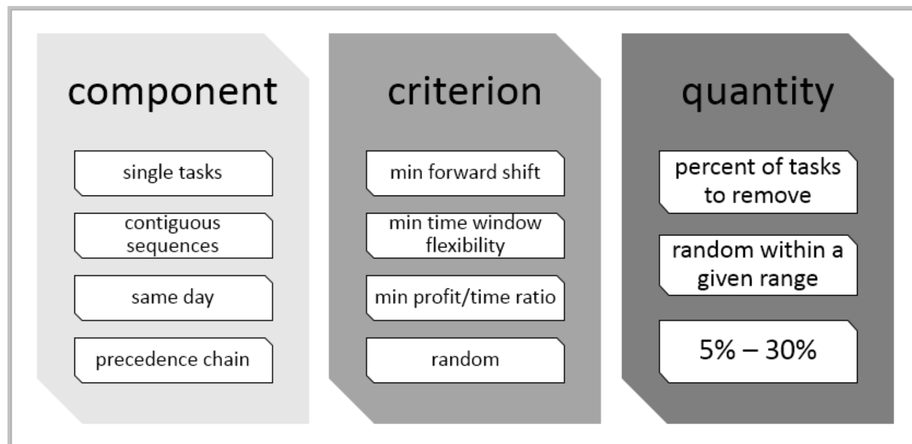


Figure 3: Overview of Destroy Operator Characteristics

**Solution Components**

As pointed out by [28], the destroy method as a whole should be capable of reaching the entire solution space, or at least those parts which are likely to include the global optimum. Consequently, it is a good idea to not always remove the same kind of parts from the solution. This of course depends on how the notion of "part" is defined. [38] proposes to select parts based on "relatedness", and the present work uses that idea as well.

The basic unit of construction in the PPP is the visit, i.e. a unique combination of task, location, task time window, and location time window. The following components, i.e. definitions of what constitutes a "part" of the solution, are proposed:

| | | |
|---|---|---|
| **A.1** | **single** | *each visit by itself* |
| **A.2** | **consecutive** | *consecutive sequence of visits* |
| **A.3** | **same-day** | *all visits on a particular day* |
| **A.4** | **chain** | *visits connected by precedence relations* |

The *single* component type is of course the most basic. It allows for the most fine-grained selection of parts to remove, and is in this sense better for small and selective perturbations. On the other hand, it offers no concept of relatedness, so that the visits removed with this definition may be from completely different and unrelated parts of the solution. This definition is also the best for a purely random destroy operator.

The *consecutive* component type defines a "part" as a sequence of $x$ consecutive visits, scheduled one immediately after the other in the solution. This provides relatedness both in terms of routing (the locations are visited one after another) as well as timing (the time windows either overlap or progress from earlier to later). Smaller

values of the component length parameter $x$ allow for a more targeted selection, while larger values focus more on the notion of relatedness. If one such consecutive part is selected for removal, *all* of its component visits are removed. A distinction can be made whether the parts defined in this way are allowed to overlap or not. For example, if a solution consists of the tasks $\{A, B, C, D\}$ and the consecutive component length is 2, then the available parts may be either $\{(A, B), (C, D)\}$ if overlaps are not considered, or $\{(A, B), (B, C), (C, D)\}$ if they are. The proposed algorithm implements the latter understanding, which allows for a finer selection of the tasks to be removed.

As its name suggests, the *same-day* component type defines a "part" as those tasks which are scheduled on the same day. Although this definition is somewhat specific to the practical application, in which separate days are clearly defined and divided, a more general definition could use consecutive, non-overlapping time windows of some specific length (which could also be subject to some variance). This component type is intended to introduce a greater degree of perturbation while ensuring that all the removed tasks are still related.

Finally, the *chain* component type targets tasks which may not necessarily be related spatially or temporally, but which are nonetheless connected through precedence relations, possibly also with time delays. Recall that in the PPP, tasks connected by precedence relations may also be subject to timing restrictions in the forms of minimum and maximum time gaps between the consecutive tasks of the chain. Such chains of visits can place a significant degree of restriction on the solution and further insertions. Although individual parts of a chain may be removed as elements of components defined with the previous definitions, this definition allows for the removal and subsequent reinsertion of an entire chain and no other tasks.

#### Removal Criteria

Provided that the potential parts for removal have been identified, the decision still has to be made on which of them to actually remove. Various evaluation or removal criteria may be defined depending on the particular factors relevant for the solution quality of the given problem. For the PPP, the following removal criteria are proposed:

| | | |
|---|---|---|
| **B.1** | **min forward shift** | *how much the visits can be shifted now* |
| **B.2** | **min time window flex.** | *how much the visits can ever be shifted* |
| **B.3** | **min profit/time ratio** | *how much the visits are worth their time* |
| **B.4** | **random** | *a purely random selection* |

The *forward shift* criterion measures how much a visit (and subsequent ones) can be pushed forward without affecting the feasibility of the solution as a whole.

When applied to an aggregate solution component, the average is calculated as an approximation. All visits are, by default, scheduled as early as possible within their position in the schedule order. As a result, a lower forward shift indicates a more constrained part of the solution, which generally points to a poor scheduling or routing plan, especially in solutions that are not that full to begin with. Although for those solutions where many tasks are scheduled a low forward shift may indicate that the time windows have been used up as much as possible, it could just as well mean that there is a bottleneck somewhere which prevents the insertion of further tasks.

The *time window flexibility* criterion refers to the degree a visit can potentially be shifted back or forward if some others are removed. It is defined as the size of the visit's time window minus the task's service time. Although this figure is constant for each visit, recall that each task has a number of visits to choose from. This criterion evaluates the visits actually chosen in the current solution. As before, the average is calculated in the case of aggregate components. Visits or segments with lower values indicate potential (or possibly already existing) bottlenecks, but from a different perspective than above. The motivation for removing them is similar as above, namely to increase flexibility for the insertion of further visits.

The *profit/time ratio* criterion tries to capture the relative value of a component compared to how it has been scheduled. The profit of course represents the score awarded by the task(s), while the time in the denominator refers to the travel time to/from these tasks, as well as their service times. Free time within a segment does not count toward the total time. Short segments with high profit score the best, whereas long segments with low profit score the worst. It should be noted that an appropriate way to deal with mandatory tasks is necessary, depending on how their profit scores are modeled: if they are set to 0, then segments with these tasks will tend to be chosen more often, whereas if the profits are set to a high number, they will rarely be chosen. For the practical application, non-mandatory tasks could have profits of either 100, 500, or 1000, so mandatory tasks were scored with 500 for the purposes of this criterion. Other policies are possible, e.g. taking the median or average profit of the tasks in the current instance. In any case, this criterion is designed to also consider the profit of the tasks and not only their timing (the focus of the previous criteria).

Finally, the *random* criterion simply assigns a random score between 0 and 100 to each solution component. As pointed out by [38], a random selection of the parts to destroy results in a diversification effect, while removing the worst components (according to the previous criteria) facilitates intensification.

#### Destroy Quantity

This last feature of the destroy operator determines how much of the solution will actually be destroyed. This parameter should generally be a variable number in a range defined by a preset minimum intensity and maximum intensity. Since different solutions will have different numbers of visits scheduled, and different destroy operators may define "parts" of different sizes for removal, this parameter refers to the percentage of *visits* to remove from those *currently scheduled*. This ensures that the relative effect is more or less the same regardless of instance size and of how empty or full the solution is.

The destroy quantity and its range play a central role in an LNS, and should be chosen carefully. If it is too low, then the search is more likely to become trapped in local optima, but if it is too high, then the LNS devolves into a random search. One possibility is to vary the possible range depending on the stage of the search, e.g. setting higher minimum and maximum values during early stages for a greater emphasis on diversification, but reducing them to a lower range later on for more intensification. However, this introduces further parameters and requires additional tuning. The proposed metaheuristic forgoes a variable destroy quantity as it was found to perform well with a fixed range of 5% to 30%.

### 4.2.2 Repair Operators

Given an empty or partial solution, the task of the repair operator is to reconstruct it in a way that improves the solution's objective values as much as possible. To this end, two elements are necessary: a feasibility restoration procedure, and a reconstruction procedure.

The issues surrounding feasibility in the PPP have been discussed in more detail in Chapter 2.3 and Section 3.3.1. Recall that if a solution is feasible, then a destroy operator can introduce at most only *weak* infeasibility, i.e. the kind resulting from the lack of some mandatory elements in the solution. If such infeasibility is handled immediately after destroying the solution, then no elaborate procedures are necessary. The missing mandatory elements, in this case visits, can be simply reinserted according to some random or greedy criteria. The reason for this is that if no non-mandatory elements are inserted beforehand, then it is guaranteed that at least one set of feasible reinsertion positions exists, namely the positions held by the tasks immediately prior to the destruction of the solution. However, other positions and other visits (i.e. locations and time windows) may still be explored, allowing for diversification among these tasks even if they are always present in every solution.

The repair procedure therefore always begins by first restoring feasibility to the solution if necessary. This procedure is identical to the general reconstruction procedure detailed below, the only difference being that only mandatory tasks are con-

sidered for reinsertion, and all are required to be inserted. If the chosen insertion order leads to infeasibility, then a new insertion order is generated with the task that could not be inserted previously at the start of the ordering. If this policy does not find a solution after a set amount of attempts, then the tasks are reinserted at their previous positions, which are guaranteed to be feasible and open.

The actual repair operators applied for reconstructing the partial solution are generated according to the same framework used for defining the destroy operators, i.e. through a combination of fundamental properties. One thing common to all repair operators is that they insert a missing element, and in the case of the PPP this is always a visit node, i.e. a preset combination of task, location, and time windows. However, distinctions between repair operators can be made based on the following properties:

| | | |
|---|---|---|
| **A** | **scope** | *the pool of tasks which are considered for insertion* |
| **B** | **order** | *the order in which the visits are inserted* |
| **C** | **criterion** | *which insertion positions are selected* |

The proposed repair operators behave like construction heuristics and are not designed to specifically find only the optimal reconstruction from a given partial solution. Optimal repair operators can be useful, particularly with small degrees of destruction to limit the potential search space and thus computational effort of the reconstruction. The original proposal for the LNS framework by [38] was in fact based around a constraint-programming method for optimal reconstruction. However, this type of approach did not seem appropriate given the limited expected computation time for the practical application.

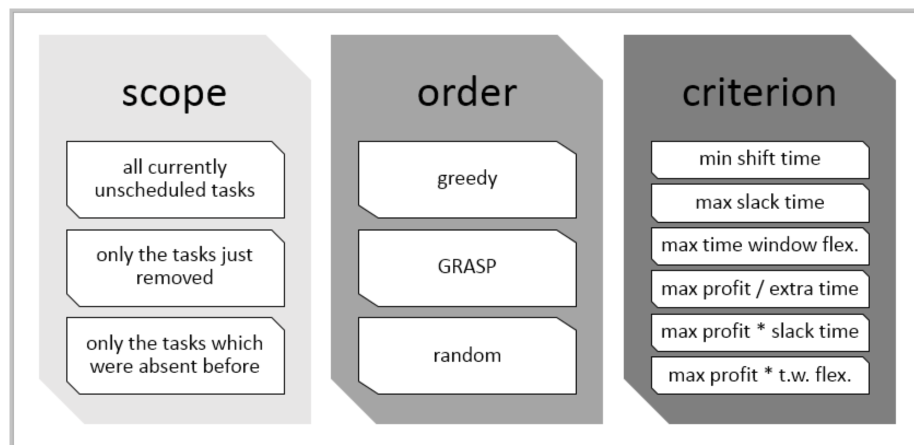


Figure 4: Overview of Repair Operator Characteristics

#### Insertion Scope

The *scope* characteristic determines the set of tasks which are eligible for insertion by the operator. One of three options may be selected:

| | | |
|---|---|---|
| **A.1** | **all** | *all currently unscheduled tasks* |
| **A.2** | **removed** | *only the tasks removed by the destroy operator* |
| **A.3** | **neglected** | *only the tasks which were previously unscheduled* |

These three options reflect varying degrees of intensification. Selecting from the pool of *removed* tasks offers the greatest degree of intensification, while the pool of *neglected* tasks imposes considerable diversification. The pool of *all* currently unscheduled tasks represents a middle ground with opportunities for both. The *removed* pool is generally the smallest, however even if it consists of only a single task, note that this does not imply that it will be reinserted at the same position or with the same visit.

#### Insertion Order

As its name implies, the *order* property determines in which order visits are inserted into the partial solution. Three options are proposed:

| | | |
|---|---|---|
| **B.1** | **greedy** | *the visit with the best best-insertion is inserted first* |
| **B.2** | **GRASP** | *one of the best best-insertions is randomly selected* |
| **B.3** | **random** | *a purely stochastic ordering* |

For the *greedy* option, the best insertion according to the selected insertion criterion (detailed in the next section) is calculated for every unscheduled task, and the best of these best insertions is performed. It should be noted that the greedy ordering must be recalculated after every insertion, because the best insertions for the remaining tasks are subject to change as the solution changes. As a result, this option may be considerably more time-consuming if many new tasks are to be scheduled. The *greedy* option may also lead to premature convergence at a relatively poor local optimum. However, if the insertion pool is relatively small, then a greedy insertion order could produce near-optimal reconstructions (not to be confused with near-optimal *solutions*).

The *GRASP* option also requires evaluating the best insertion for each unscheduled visit, but it is not always the best of these best insertions which is selected. Instead, a restricted candidate list of some predefined size is considered, and one of the insertions from this list is randomly performed. This is done in very much the same way as in the more general GRASP heuristic. A selection based on a roulette wheel implementation is proposed here so that within the restricted candidate list,

insertions with relatively better costs/benefit ratios have a greater probability of being chosen.

The *random* option is mostly self-explanatory. A list of all the unscheduled tasks is created and shuffled, and the tasks are then best-inserted in this order until all tasks have been examined. This option provides a degree of diversification not obtainable with the other options above. Although it is unpredictable, it is also noticeably faster per iteration since its computational complexity is considerably lower (each visit's best insertion is calculated only once, compared to once for every insertion step as in the previous two options).

In all cases, it should be noted that this property has no direct influence on which visits are chosen for each task, or on the positions at which those visits are inserted. Those decisions are determined by the last repair element:

#### Insertion Criterion

The insertion *criterion* defines the measure used to evaluate and compare the many feasible insertions for each visit. The following three are proposed:

| | | |
|---|---|---|
| **C.1** | **min shift time** | *the extra travel and service time required* |
| **C.2** | **max total slack** | *the potential slack time at the chosen position* |
| **C.3** | **max time w. flex.** | *time window size compared to the service time* |

The *shift time* criterion measures how much subsequent tasks must be shifted in order to accommodate the insertion. It is the sum of the visit's service time and the travel time to and from this visit at the proposed insertion position, minus the current travel time between the visits before and after the insertion position. Smaller scores are favored as they reflect a better routing and a less time-consuming task. Since all the other criteria are to be maximized, the reciprocal of the extra time is taken instead, with a large numerator to prevent rounding inaccuracies.

The *total slack* criterion is an estimation of how flexible the inserted visit could be at the selected position. The calculation of the exact slack time is somewhat more involved due to the presence of the minimum/maximum time delays between some tasks, so that only an approximation is used here. The score for this criterion is the sum of the necessary waiting time before starting service, and the time between the end of service and the visit's closing time. Insertions with lower total slack potential are more likely to become bottlenecks because they cannot be shifted as easily. In addition, including the waiting time in this measure encourages time windows to be chosen in such a way that subsequent visits in the schedule do not have overlapping time windows (if they do, then there is only travel time between them). The visits with large or similar time windows are then generally left for later steps of the construction, and can then still be more easily inserted thanks to these overlaps,

whether as replacements or as additional visits. Overall, the *total slack* criterion attempts to capture flexible positions in the solution's current timing structure, since the waiting time and slack time are highly dependent on the current state of the schedule.

The *time window flexibility* criterion, on the other hand, does not depend on the current state of the schedule. Defined as the visit's time window length minus its service time, it is a static measure for each visit. However, as with the equivalent destroy criterion, each task in the PPP generally has many possible visits, as determined by all of its time windows, all of its locations, and all of those locations' time windows. As a result, this criterion seeks to filter these possibilities for those which likely offer greater temporal flexibility in the long run of the construction. Visits with larger leeway in their time windows will generally have shorter service time, longer time windows, or both. Inserting such visits is unlikely to trap the solution in a local optimum too early, since these kinds of visits can more easily be shifted as needed during subsequent insertions.

All three of these criteria focus exclusively on the timing aspects of the problem. However, the visit's profit can easily be incorporated into the insertion decision. To achieve this, the insertion scores are weighted according to the profit of the visit, by multiplying the score and the profit. This yields a total of six insertion criteria.

### 4.2.3 Efficiency Evaluation

Efficiency is examined not only for every repaired solution, but also for every intermediate solution. Therefore, it is important that the efficiency evaluation is sufficiently efficient itself, especially when the approximation sets become large. Compared to a naive approach, the following procedure allows to reduce the average number of comparisons necessary to evaluate the efficiency of a new solution and determine all the old solutions which have possibly been dominated.

Storing the efficient solutions in order of increasing profit ensures that they are also listed in order of decreasing free time, and vice versa. When a new solution is found, it is compared to the solutions in the efficient set in sequence. If it is dominated by any single solution then the search is immediately ended. If the solution is incomparable because one objective is better while the other is worse, then the search is continued until a dominating solution is encountered, or until the relationship between the objectives is reversed - this happens precisely at the new solution's position in the efficient set if it is efficient. If the new solution dominates any existing solutions, then they will all be encountered immediately prior to this position in the list.

## 4.3 Low-Level Layer

This section explains how some common elements from standard routing applications can be adapted to deal with the additional constraints in the PPP. Specifically, the precedence constraints and their associated minimum and maximum time gaps lead to complications with evaluating the feasibility of insertions, updating the timing data after an insertion, and finding plans in which precedence chains are efficiently scheduled.

### 4.3.1 Timing Update

After an insertion has been performed, it is necessary to update the timing data. Procedures for standard routing problems with time windows have been suggested in the literature (e.g. by [42] for the TOPTW). However, the presence of minimum and maximum time delays between tasks introduces some complicating factors. This section details how the timing update after an insertion can be handled in the PPP.

For each visit in the schedule, the following timing data are stored:

| | | |
|---|---|---|
| **Wait** | $w$ | *the waiting time before starting the task* |
| **Begin** | $b$ | *the start of the task at the chosen location* |
| **End** | $e$ | *the end of the task at the chosen location* |
| **Slack** | $s$ | *the maximum feasible postponement of the task* |

It is assumed that all visits are scheduled at the earliest feasible time within their selected time windows. Note that this does not mean the earliest time window. The schedules returned at the end of the search may still be post-processed to shift visits to later times as desired.

The wait variable is subject to some additional considerations in the PPP. If there exists a minimum time delay between two visits, then even if they are scheduled consecutively and have overlapping time windows, there must still be a waiting time planned before the later visit in order to satisfy the delay. Similarly, if there is a maximum time delay and the visits are scheduled far apart, it may be necessary to add extra waiting time before the earlier visit. When multiple time gap relations exist for a given visit, then it is necessary to find a waiting time which satisfies all of them in order for the insertion to be feasible, and this waiting time should be minimized so that the start time is at the earliest time within the selected time window (as noted above).

The slack time also requires additional checks. In standard routing problems with time windows, the slack time for a visit $j$ measures the amount of time it may be delayed without causing any time window infeasibility at *subsequent* visits. It is then simply the sum of the waiting and slack time at the next visit $j+1$, or the time

until the end of the time window at visit $j$, whichever is lower [42]. However, in the PPP, minimum and maximum time delay constraints from both earlier and later visits may shorten this slack time further. Determining by how much can become rather involved when multiple time delay constraints are present, and multiple, unrelated precedence chains are involved. This makes it inefficient and error-prone to update the slack time together with the other variables, so this step is only done once the other three timing variables have already been updated for the whole schedule. The update of the wait, begin, and end times is described below, and the slack time update is detailed in the next section.

Inserting a selected visit $j$ between visits $i$ and $k$ may shift $k$ and subsequent visits. Provided that the triangle inequality is satisfied in the distance matrix, the total *shift* starting at the position of visit $k$ is determined according to the formula below:

$$shift = d_{ij} + w_j + t_j + d_{jk} - d_{ik}$$

Figure 5 shows the insertion of a visit $j$ in a standard routing problem with time windows. The x-axis represents the time at that point in the schedule, visit $j$ is inserted at time 13. The y-axis corresponds to the remaining shift at a particular position in the schedule. The step-wise decreasing line plots the decreasing shift as it is resolved at subsequent visits of the schedule. Once the shift is reduced to 0, no further changes occur. In the example, only visits below the step-wise line are affected by the shift caused by inserting visit $j$.

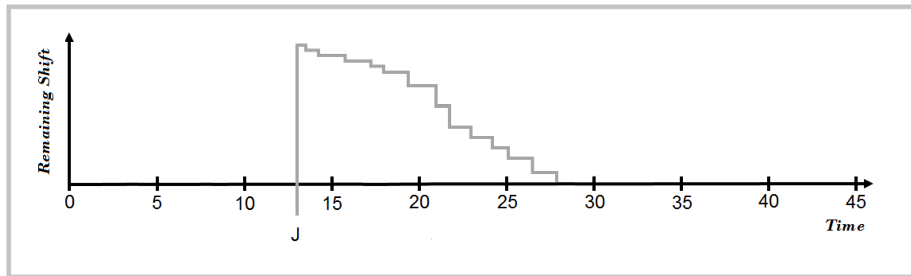


Figure 5: Example Timing Update (1/4)

At this point, the presence of minimum and maximum time delays introduces several complications to this procedure in the PPP. Some examples are now considered. For the sake of simplicity, service times and travel times are ignored.

Suppose there is a precedence chain of visits $x$ and $y$, scheduled at times 21 and 36, respectively, and that the minimum delay between these two visits is 15 time units. When the timing update reaches visit $x$, the shift has not yet been reduced to 0, and visit $x$ is shifted by some time units. However, this leads to a violation of

the minimum time delay to visit $y$. Clearly, visit $y$ must be shifted as well, and in this example by the same amount as visit $x$. Notice that visit $y$ can be shifted only *at most* as much as its predecessor visit $x$ - if the delay between the visits were larger than the time delay, for example 20 time units, then the shift at visit $y$ would be reduced by (20 - 15) = 5 time units. Either way, some of the shift is propagated to a later part of the schedule, even though that part would otherwise not have been reached by the shift from task $j$. This is shown in Figure 6.

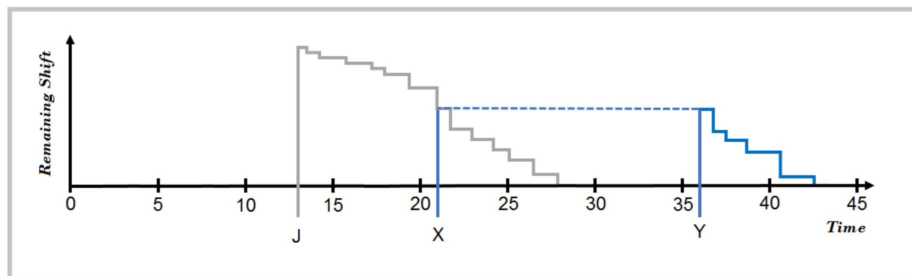


Figure 6: Example Timing Update (2/4)

Consider now the case of a maximum delay. In the example above, even if there is a maximum delay between visits $x$ and $y$, it can never be violated by shifting visits to later parts of the schedule, since visit $y$ can never be shifted more than visit $x$. However, suppose there is a precedence chain of visits $a$ and $b$, scheduled at times 3 and 18, with a maximum delay of 20 time units. This delay is not a tight constraint at this point, as the visits are only (18 - 3) = 15 time units apart. Suppose now that when the timing update reaches visit $b$, and the remaining shift is 10 time units. After visit $b$ is shifted to time 28, the gap to visit $a$ has increased to 25, so that visit $a$ must now be shifted (25 - 20) = 5 time units. As before, visit $a$ can be shifted no more than visit $b$, but unlike in the previous example, now the shift can be propagated even to parts of the schedule *before* the original insertion position of visit $j$. This stands in stark contrast to classic routing problems. Figure 7 shows the effect of maximum delays in the PPP.

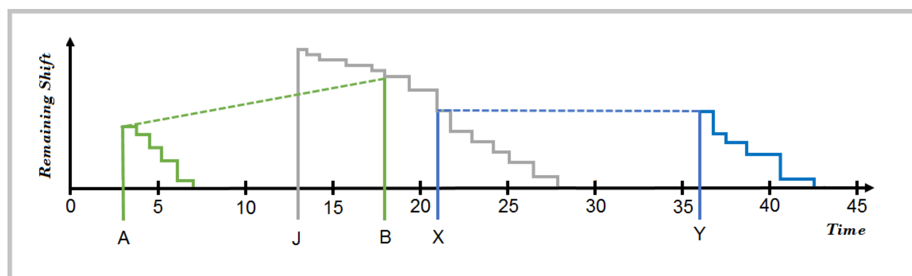


Figure 7: Example Timing Update (3/4)

In the two examples above, the shift is propagated to positions before or after the reach of the original shift (marked in grey) caused by visit $j$. In both cases, it does not matter in what order the propagations are resolved. In general however, propagations of a shift should be resolved immediately. This prevents difficulties in cases when two or more visits of a precedent chain fall within the reach of a single shift.

Consider the example shown in Figure 8. Suppose there is another chain of visits $p$ and $q$, scheduled at times 23 and 26, respectively, and with a minimum delay of 3 time units. When the timing update reaches visit $p$, visit $q$ must be shifted in the same manner as visit $y$, but before the remainder of the initial shift is resolved. In this way, when the original shift caused by visit $j$ reaches visit $q$, the waiting time before visit $q$ will have already been updated to absorb the entire remaining shift. Otherwise, visit $q$ could be incorrectly shifted a second time.

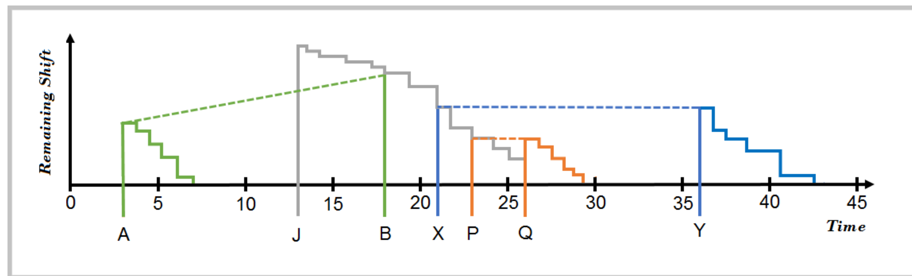


Figure 8: Example Timing Update (4/4)

Note that only minimum delay constraints can cause propagations to later parts of the solution (as in Figure 6). Likewise, only maximum delay constraints can lead to propagations to earlier parts of the solution (as in Figure 7). Importantly, every propagation can itself lead to further propagations, each of which may take the form of any of the cases described above. With this in mind, Algorithm 2 presents pseudocode for a recursive implementation handling the timing update in the PPP:

**Algorithm 2** Timing Update

1: $j \leftarrow$ the visit which was originally inserted
2: $n \leftarrow$ the visit to be shifted
3: $shift \leftarrow$ the amount of time visit $n$ must be postponed
4: **while** $shift > 0$ **do**
5: $\quad shift \leftarrow max[0, (shift - w_n)]$
6: $\quad b_n \leftarrow b_n + shift$
7: $\quad e_n \leftarrow e_n + shift$
8: $\quad w_n \leftarrow b_n - (e_m + d_{mn})$ (where $m$ is the visit immediately before $n$)
9: $\quad$ **for** all scheduled predecessors $p$ in $\rho_n$ **do**
10: $\quad\quad v \leftarrow (b_n - e_p) - \beta_{nd}$
11: $\quad\quad$ **if** $v > 0$ **then**
12: $\quad\quad\quad$ timingUpdate$(j, p, (v + w_p))$
13: $\quad$ **for** all scheduled dependent visits $d$ in $\delta_n$ **do**
14: $\quad\quad v \leftarrow \alpha_{nd} - (b_d - e_n)$
15: $\quad\quad$ **if** $v > 0$ **then**
16: $\quad\quad\quad$ timingUpdate$(j, p, (v + w_d))$
17: $\quad$ **if** $n = j$ **then**
18: $\quad\quad shift \leftarrow d_{ij} + w_j + t_j + d_{jk} - d_{ik}$
19: $\quad n \leftarrow n + 1$ (the next visit after $n$)
20: **return**

The algorithm takes three parameters: the visit $j$ which was originally inserted, the visit $n$ which is to be shifted, and the *shift* amount. Due to the recursive nature of the algorithm, the first call is made with values of ($j$, $j$, 1), and beforehand $w_n$ is set to 1, and $b_j$ and $e_j$ are reduced by 1. It is assumed that the solution is feasible prior to the insertion.

Line 4 opens the while loop which continues the update until the *shift* has been reduced to 0. Line 5 reduces the *shift* by the waiting time prior to the current visit, and ensures it remains non-negative in case all the shift has been absorbed. Lines 6 to 8 then update the begin, end, and waiting times of the current visit.

Line 9 examines all of the current visit $n$'s scheduled predecessors, if any. For each one, the difference between the time delay to the predecessor, and the maximum time delay, is computed in line 10. If this difference is greater than 0, then the maximum delay has been violated (linen 11), and the predecessor must be shifted by the difference. The algorithm is therefore called again with $j$, $n$, and the difference $v$ in line 12. An equivalent procedure is followed for all of visit $n$'s scheduled dependent visits in lines 13 to 16.

If this is the first call to the algorithm and the first iteration of the loop, then visit

$n$ will in fact be visit $j$. Line 18 then sets the *shift* to its proper value for all subsequent visits. The dummy parameters for the first call to the algorithm are necessary to ensure that also visit $j$'s predecessors and dependent tasks are shifted accordingly, since visit $j$ would otherwise never actually be shifted (but rather only inserted). Finally, line 19 sets the current visit to the next one in the schedule. The procedure then loops from line 4 until no more visits must be shifted.

### 4.3.2 Slack Update

In classical standard routing problems with time windows, the slack time can be rather easily updated along with the other timing variables. [33] and [42] propose such procedures, based on the *shift* time of an insertion, and backwards recursion.

In the PPP however, minimum and maximum time delays introduce some complicating factors. In the algorithms of [33] and [42], once the *shift* has been reduced to 0, no later visits need to be updated. This is generally not the case in the PPP. Consider the example in Figure 6 from the previous section: there could be a visit $z$ near the end of the schedule with a maximum time delay to visit $y$. Even if $z$ is not shifted, the shift of $y$ can nonetheless extend the slack time of $z$. The additional slack time at $z$ may then affect the visits before $z$, even if they are not related to $y$ or $z$ in any way by time delay constraints. Similarly, if there is a minimum time delay between $y$ and $z$, then removing $z$ may increase the slack time of $y$ by more than simply the freed up time. The complexity of these interactions increases when multiple unrelated sets of such connected visits are considered, especially when these sets overlap (for example visits $p$ and $q$ in Figure 6 are between visits $x$ and $y$, but are not directly related).

One consequence of the above is that potentially every visit's slack time may need to be updated, regardless of whether or not it was actually shifted itself (like visit $z$ and those before $z$ in the example above). Looking at the highly simplified example in Figure 8, there are already four non-connected parts of the solution which are not directly affected by any of the shifts. Rather than keeping track of all of them and dealing with the many potential precedence chains and the complex interactions between them *during* the timing update, it is conceptually simpler and less error-prone to update the slack times of all visits afterward, in a single run through the schedule. Such an implementation is proposed for the PPP in Algorithm 3 below:

**Algorithm 3** Slack Calculation

```
1: for all visits j in the schedule do
2:     s_j ← c_j − e_j
3: for all visits j except the last, from latest to earliest do
4:     if s_j > w_{j+1} + s_{j+1} then
5:         s_j ← w_{j+1} + s_{j+1}
6:     for all scheduled precedent visits i in ρ_j do
7:         u ← e_i + s_i + β_{ij}
8:         b_max ← b_j + s_j
9:         if u < b_max then
10:            s_j ← s_j − (b_max − u)
11:        b_max ← b_j + s_j
12:        l ← e_i + s_i + α_{ij}
13:        if b_max < l then
14:            s_i ← s_i − (l − b_max)
15: return
```

The main idea behind the algorithm is to recalculate all the slack times by reducing them from their upper bounds until all of them are feasible. It is similar to the procedures of [33] and [42] in that it also relies on backwards recursion. The difference is that the updated timing variables (waiting, start, end) must already be known for all visits in the schedule, rather than only the current one. That is why Algorithm 2 from the previous section must be run first.

Lines 1 and 2 start the algorithm by setting the slack time of every scheduled visit $j$ to its upper bound, namely the difference between the visit's closing time $c_j$ and its end time $e_j$. This is necessary so that the procedure can also be used when visits are removed from the schedule, which leads to an unknown *increase* in the slack times, rather than a decrease. Line 3 then begins the updating procedure from the next to last visit $j$. Line 4 checks whether the current visit $j$'s slack time $s_j$ is larger than the sum of the waiting time $w_{j+1}$ and the slack time $s_{j+1}$ of the next visit $j+1$ in the schedule. If this sum is larger, then visit $j$'s slack is infeasible and must be reduced to that sum in line 5. This is identical to the backwards recursion applied by [33] and [42].

In order to consider the additional effects on time delay constraints, line 6 examines all scheduled predecessors $i$ of the current visit $j$. Only tasks connected by precedence relations can have associated time delays in the proposed PPP formulation, but an extension to any combination is straightforward by simply replacing $\rho_j$ with the set of visits connected to visit $j$ by time delays. Lines 7 to 10 consider the effect of a maximum delay between visit $j$ and its predecessor $i$. Line 7 calculates

an *upper* bound $u$ on the latest feasible start time of visit $j$ when predecessor $i$ is shifted as late as possible. This is the sum of the predecessor's end time $e_i$, the predecessor's slack time $s_i$, and the maximum time delay $\beta_{ij}$ between the predecessor $i$ and current visit $j$. Line 8 calculates the latest feasible start time $b_{max}$ of visit $j$ with regard to its own slack time. This is simply the start time $b_j$ plus the slack time $s_j$. If $b_{max}$ is greater than $u$ (line 9), then that implies that visit $j$'s slack has been overestimated, and $s_j$ is then reduced by the difference of $b_{max}$ and $u$ in line 10.

Similar calculations are then performed in lines 11 to 14 to examine the effect of a minimum time delay between visits $i$ and $j$. $b_{max}$ is first recomputed in line 11 using the updated $s_j$. Line 12 then calculates a *lower* bound $l$ on $j$'s start time for the case that its predecessor $i$ is shifted as much as possible. This is the sum of $i$'s end time $e_i$, its slack time $s_i$, and the minimum time delay $\alpha_{ij}$ between $i$ and $j$. If the lower bound $l$ is greater than the actual latest start time $b_{max}$ (line 13), then that means the slack time of the predecessor has been overestimated, and $s_i$ is then reduced accordingly by the difference of $l$ and $b_{max}$ in line 14.

The for-loop started in line 6 is then continued for the next scheduled predecessor of current visit $j$. Note that dependent tasks scheduled later than the current visit $j$ need not be considered explicitly - they will have been considered before reaching visit $j$ since the algorithm begins at the end of the schedule. Once all predecessors of $j$ have been examined, the algorithm continues from line 3 with the next visit $j-1$, and terminates on line 15 once the outer for-loop has been resolved.

Pre-processing the slack time allows for a more efficient feasibility evaluation of insertions. For the problems considered by [33] and [42], the computational complexity of evaluating insertion feasibility can be reduced from $O(n)$ to $O(1)$ constant time on average in this way. In the PPP, the maximum and minimum time delays add some overhead. The algorithm described above runs in $O(n \cdot B)$ time, where $B$ is greater than $\rho_j$ for all visits $j$. As a result, the average complexity of checking an insertion's feasibility becomes $O(B)$. Although $B$ can be as large as $n-1$, where $n$ is the total number of tasks, precedence chains in practical instances are generally much shorter and rarely longer than 5 tasks. The overhead is therefore mostly negligible in practice.

### 4.3.3 Insertion Feasibility Evaluation

In the previous sections, it is assumed that an insertion has already been selected and performed. However, before this can be done, the feasibility of the selected insertion must be evaluated. In standard routing problems with time windows, it is sufficient to examine the slack time of the visit immediately after the intended insertion position. If the shift caused by the intended insertion is less than or equal to this slack time, then the insertion is at least feasible.

In the PPP, the slack time for each visit must take into account all of the precedence and time delay constraints between all visits in the schedule. If the slack times are updated as outlined in the previous section, then this condition is satisfied. However, there exist specific cases in the PPP where examining only the slack time is not sufficient.

Consider the example shown in Figure 9. The boxes represent task service time, and the thick lines beneath represent the time window of the selected visit. Assume all travel times are 1 and that the schedule begins with visit $a$. Looking at the initial state, visit $x$ has a slack time of 3 time units, with a waiting time of 1. Visit $a$ has 5 time units left until the end of its time window, so the slack time of visit $a$ is $min$[5, (3+1)] = 4 time units.

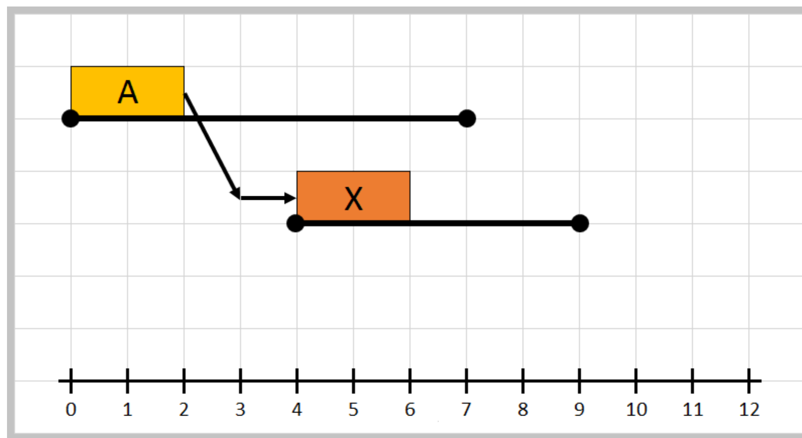


Figure 9: Example Insertion (1/3)

Visit $a$ is part of a precedence chain, and visit $b$ is to be inserted as shown in Figure 10. Suppose that the maximum time gap between visits $a$ and $b$ is 4 time units. For the given visit $b$, the time window starts at time 8. Subtracting the time gap 4 from this means that visit $a$ must end at time 4 or later. It currently ends at time 2, meaning it would have to be shifted 2 time units. If the shift of earlier visits is not considered, then this particular insertion will be regarded as infeasible.

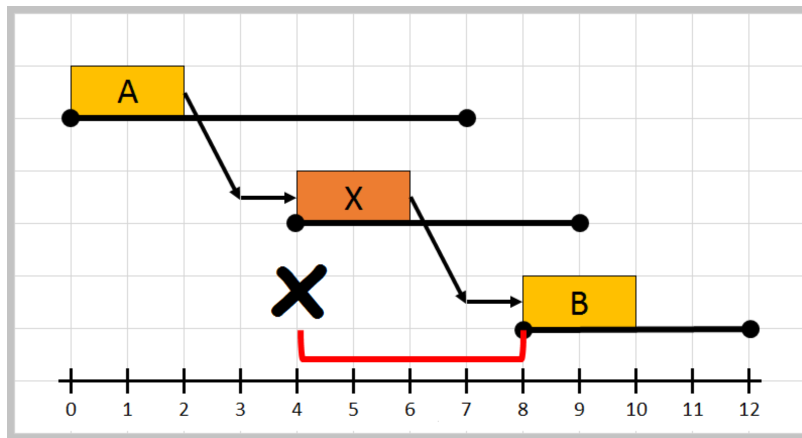


Figure 10: Example Insertion (2/3)

However, it may in fact be a highly efficient position, and in this example it is feasible. As was shown in Section 4.3.1, in the PPP it is possible that insertions cause shifts also *before* the insertion position. The resulting schedule is shown in Figure 11.

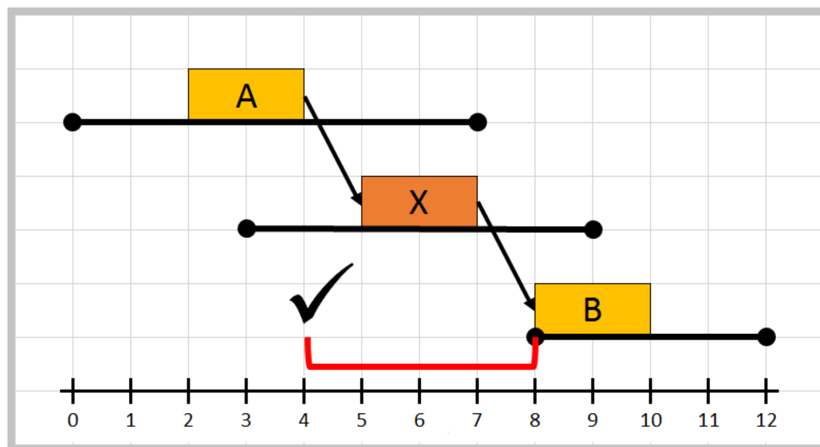


Figure 11: Example Insertion (3/3)

Consider now the case where the maximum time gap is 3 units instead of 4. Now visit $a$ has to be shifted by 3 time units, and this also appears to be possible since this is lower than the visit's slack of 4. However, shifting visit $a$ by 2 units ends up pushing visit $b$ forward by 1 unit due to visit $x$ and the travel times. This in turn requires shifting visit $a$ again, leading to an infinite loop. It can be seen in Figure 11 that the time difference between visits $a$ and $b$ can be no smaller than 4 time units due to visit $x$ and the travel times, so that the intended insertion can never be feasible.

Clearly, the slack alone is not sufficient for evaluating insertion feasibility under the presence of time gap constraints. The problem lies in the fact that the slack time can take into account only time delay constraints which are actually binding in the current schedule. Hence visit $a$'s slack cannot take into consideration the maximum time delay to visit $b$, because visit $b$ is not in the solution when this slack time is calculated.

This kind of error can only happen after the insertion of a visit with a maximum delay to an earlier precedent, and even in these cases, it is rare. It depends on which visits are scheduled in between, and what the exact travel times are. In order to handle this case, it is sufficient to perform these kinds of insertions on a tentative copy of the schedule, and accept it only if no such loops occur. An insertion is infeasible in this regard if any visit is ever shifted more than once - if this occurs then the insertion can be discarded and another one may be taken instead. In most cases, the original visit to insert may simply be inserted at the next available position.

# Chapter 5

# Computational Experiments

This chapter reports the results of running the proposed metaheuristic on a number of test instances supplied by the partnering research institution. The first section describes these instances and the data sets in more detail. This is followed by an explanation of the parameter settings used for the experiments. Finally, summaries of the test results are presented. An exact solution method for the PPP was implemented by the partnering research institution, so for a subset of the instances, optimal reference sets are available as benchmarks. Approximate reference sets were used as benchmarks for the remaining instances.

## 5.1 Data Sets

The available test instances can be divided into six groups (A to F), each representing one of six different fictional personas with a different type of schedule pattern. Their schedules differ in the number of tasks, ranging from 30 to over 80, and the relative number of precedence relations among them, ranging from approx. 15% to 88% (meaning this share of the tasks are subject in some way to precedence constraints). Five versions exist for each schedule, representing different weeks of a longer, partitioned planning horizon.

With regard to the routing aspect of the problem, existing data sets of real locations in the city of Vienna, Austria were used. Appropriate subsets (e.g. supermarkets, restaurants, specialty stores) were used as the available locations for different types of tasks. One location designates the decision-maker's "home", which is always the start and end point of the schedule. Time window data on opening and closing times was either supplied from a database or estimated. The underlying distance matrix for these locations was determined according to the mode of transportation used in each instance. Five modes of transportation (lowercase a to e) were modeled: personal automobile (a), car-sharing service such as Car2Go (b),

public transportation (c), cycling (d), and walking (e).

Each instance is constructed using one of the schedules (persona/week), a set of available locations, and one mode of transportation. This is reflected in the instance IDs, e.g. ID “B2a” corresponds to persona B’s second planning week, using a car.

Two instances were created for each schedule (persona/week), using different modes of transportation, and every resulting instance was scaled using different sets of available locations, ranging in size from 50 to nearly 3000 locations. The results reported in this section are based on the largest available instance for each schedule and transportation mode combination.

## 5.2 Parameter Settings

Although in principle a mostly parameter-less implementation can be constructed, several parameters are used to better control the progression from one search phase to another in order to improve the quality/run time performance. The table below summarizes the parameters of the algorithm:

| **Symbol** | **Description** |
|---|---|
| $l_{exp}$ | the limit on the number of iterations in the exploration phase |
| $l_{con}$ | the limit on the maximum consecutive iterations without improvement in the consolidation phase |
| $l_{ref}$ | the limit on the number of sweeps over the whole efficient set in the refinement phase |
| $f_{ref}$ | a parameter which skews the focus of the search in the refinement phase |
| $t_{max}$ | the maximum run time before the algorithm terminates |
| $d_{min}$ | the minimum destroy intensity, as a percentage of visits scheduled |
| $d_{max}$ | the maximum destroy intensity, as a percentage of visits scheduled |
| $s_{min}$ | the minimum segment length for the *consecutive* destroy operator |
| $s_{min}$ | the maximum segment length for the *consecutive* destroy operator |
| $RCL$ | the size of the restricted candidate list for the *GRASP* repair operator |

Table 1: Parameters of the Proposed Metaheuristic

It is possible to skip the exploration phase entirely and start the consolidation phase with only the empty solution in the stack. However, initial experiments showed that this approach takes much longer to converge to an approximation set of similar quality. If the first and only iteration starting with the empty solution results in a poor construction, then the consolidation phase spends much of its time iteratively improving these poor intermediate solutions. As a result, $l_{exp}$ was set to 20 so that the consolidation phase can start with a reasonably good approximation set.

In the consolidation phase, it is possible to omit the parameter entirely by ending the phase only once the solution stack is empty. However, experiments showed

that the solutions near the bottom of the stack only rarely lead to improvements in the efficient set. By the time the consolidation phase starts, these early solutions are too inferior to be of much help. This is the reason why a LIFO policy was used for the stack, but also the reason why a parameter $l_{con}$ is necessary, especially for large-scale instances in which the stack may be very large. Tests on the largest instances showed that an $l_{con}$ value of 20 was sufficient for exploring the most promising solutions in the stack and deleting the rest.

Being the final part of the search, the refinement phase needs some sort of stopping condition. A natural choice is to limit the number of times the approximation set is refined, i.e. the number of destroy/repair sweeps over the entire set. Examining the most difficult instances, it was observed that improvements to the efficient set were at best only marginal after two or three sweeps. As a result, a sweep count limit $l_{ref}$ of 3 was chosen. In addition, an upper limit on the run time, $r_{max}$, was defined and set to 300 seconds.

An additional parameter $f_{ref}$ was introduced in order to skew the focus of the refinement phase. As mentioned at the start of Section 4, choosing total free time as the second objective results in "emptier" solutions with fewer tasks at the upper end of the front, and "fuller" solutions at the lower end. One consequence is that it is generally much easier to find highly efficient solutions for the upper part of the front, because finding a very good or optimal combination and arrangement of a small number of tasks is generally easier than for a larger number. This intuition was confirmed in initial experiments, where it was observed that most of the improvements in the refinement phase were made in the lower region of the front, and by using solutions from the lower region as the working solutions.

In order to focus more of the search on the lower part while still sweeping over all solutions in the approximation set, a solution $s_i$ with index $i$ in the approximation set $A$ is considered $n$ times per sweep, where:

$$n = \lceil (i/|A|) * f_{ref} \rceil$$

Provided that $A$ is ordered as proposed from lowest to highest profit, this selection rule allows solutions from the lower part to be selected for refinement up to $f_{ref}$ more frequently than the solutions from the upper part. Setting $f_{ref}$ to 1 results in an unbiased sweep over all solutions equally. Note that in the case of multiple sweeps, it is not equivalent to perform a single sweep using the sum of the $n$ attempts per solution, because the solutions in the set are subject to change during each sweep and the newer ones are reselected during the next. $f_{ref}$ was set to a value of 3 for all experiments.

With regard to the destroy and repair operators, the destroy intensity range was fixed for the entire search, with $d_{min}$ set to 5 and $d_{max}$ to 30. The *consecutive* operator was set to evaluate sequences with a minimum length $s_{min}$ of at least 2 visits,

and with an $s_{max}$ of 5. The size $RCL$ of the restricted candidate list for the repair operator's *GRASP* insertion order was fixed at 5. Finally, the available characteristics of the repair operator were slightly adjusted depending on the search phase. In particular, the insertion scope during the exploration phase must always be *all* since every iteration starts from an empty solution with no previous state. In addition, only the *greedy* and *GRASP* insertion orders were active during refinement, in order to focus explicitly on intensification.

Finally, all computations were conducted on a personal computer with an Intel Core i7 2.20 GHz processor with 8GB of RAM. The table below summarizes the default parameter settings used in all experiments (unless noted otherwise):

| Parameter | Value | Parameter | Value | Parameter | Value |
|---|---|---|---|---|---|
| $l_{exp}$ | 20 | $d_{min}$ | 5 | $s_{min}$ | 2 |
| $l_{con}$ | 20 | $d_{max}$ | 30 | $s_{min}$ | 5 |
| $l_{ref}$ | 3 | $t_{max}$ | 300 | $RCL$ | 5 |
| $f_{ref}$ | 3 | | | | |

Table 2: Default Parameter Values

## 5.3 Instances with Exact Reference Sets

A set of smaller instances solvable to optimality was made available by the partnering research institution as a benchmark for the proposed metaheuristic. These instances are based on a shorter planning horizon of 3 as opposed to 7 days, and contain fewer tasks and considerably fewer locations. Each instance was solved 10 times with the parameter settings outlined in the previous section. Table 3 reports the results for each instance, and Table 4 summarizes the main indicator values.

The results show that the algorithm performs very well and generates a near-optimal set of solutions for almost all of the benchmark instances. The average hypervolume attained over all instances and all runs is above 99% of the optimal value, and the average epsilon indicator over all instances and all runs is 1.019. In terms of worst-case performance, even the worst runs attain on average over 98% of the optimal hypervolume, and an average epsilon indicator of 1.045. These results indicate that the algorithm is generally reliable.

| Instance | Tasks | Chain % | Locations | Visits | Average Front Size | Average Run Time | Hypervolume (%) | | | Mult. Unary Epsilon | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Worst | Ave | Best | Worst | Ave | Best |
| A1a | 21 | 66.67 | 19 | 102 | 14 | 1 | 99.10 | 99.43 | 99.79 | 1.125 | 1.045 | 1.013 |
| A1d | 21 | 66.67 | 19 | 102 | 14 | 1 | 95.39 | 97.42 | 99.38 | 1.077 | 1.039 | 1.011 |
| A2a | 21 | 52.38 | 20 | 100 | 16 | 1 | 99.41 | 99.66 | 99.88 | 1.053 | 1.035 | 1.017 |
| A2d | 21 | 52.38 | 20 | 100 | 17 | 1 | 99.13 | 99.52 | 99.78 | 1.078 | 1.038 | 1.012 |
| A3a | 24 | 75.00 | 20 | 111 | 15 | 1 | 98.90 | 99.57 | 99.89 | 1.355 | 1.110 | 1.007 |
| A3d | 24 | 75.00 | 20 | 111 | 17 | 2 | 98.89 | 99.74 | 99.90 | 1.299 | 1.039 | 1.006 |
| A4a | 22 | 50.00 | 20 | 107 | 20 | 1 | 99.34 | 99.84 | 100.00 | 1.035 | 1.013 | 1.001 |
| A4d | 22 | 50.00 | 20 | 107 | 19 | 1 | 95.14 | 97.81 | 99.77 | 1.061 | 1.034 | 1.010 |
| A5a | 21 | 61.90 | 17 | 88 | 22 | 1 | 99.36 | 99.61 | 99.79 | 1.068 | 1.028 | 1.017 |
| A5d | 21 | 61.90 | 17 | 88 | 21 | 1 | 99.44 | 99.73 | 99.92 | 1.052 | 1.022 | 1.003 |
| B1a | 19 | 21.05 | 29 | 86 | 19 | 0 | 93.30 | 98.50 | 99.95 | 1.100 | 1.040 | 1.007 |
| B1d | 19 | 21.05 | 29 | 86 | 18 | 0 | 92.03 | 98.09 | 99.97 | 1.106 | 1.041 | 1.004 |
| B2a | 19 | 36.84 | 27 | 107 | 13 | 0 | 99.71 | 99.93 | 99.99 | 1.016 | 1.004 | 1.000 |
| B2d | 19 | 36.84 | 27 | 107 | 14 | 0 | 99.88 | 99.95 | 100.00 | 1.006 | 1.004 | 1.000 |
| B3a | 22 | 36.36 | 28 | 91 | 11 | 0 | 97.43 | 97.91 | 99.58 | 1.100 | 1.052 | 1.017 |
| B3d | 22 | 36.36 | 28 | 91 | 11 | 0 | 94.35 | 97.16 | 98.03 | 1.100 | 1.043 | 1.009 |
| B4a | 16 | 37.50 | 29 | 49 | 13 | 0 | 99.88 | 99.93 | 99.99 | 1.003 | 1.002 | 1.001 |
| B4d | 16 | 37.50 | 29 | 49 | 14 | 0 | 99.89 | 99.96 | 100.00 | 1.005 | 1.003 | 1.001 |
| B5a | 17 | 29.41 | 30 | 71 | 25 | 0 | 99.97 | 99.98 | 100.00 | 1.016 | 1.006 | 1.001 |
| B5d | 17 | 29.41 | 30 | 71 | 25 | 0 | 99.98 | 99.99 | 100.00 | 1.012 | 1.006 | 1.001 |
| C1a | 24 | 58.33 | 24 | 64 | 19 | 1 | 97.86 | 98.50 | 100.00 | 1.006 | 1.001 | 1.000 |
| C1d | 24 | 58.33 | 24 | 64 | 15 | 0 | 98.70 | 99.83 | 100.00 | 1.037 | 1.007 | 1.000 |
| C2a | 24 | 41.67 | 20 | 64 | 28 | 1 | 99.67 | 99.84 | 99.97 | 1.028 | 1.017 | 1.009 |
| C2d | 24 | 41.67 | 20 | 64 | 26 | 1 | 99.27 | 99.65 | 99.94 | 1.039 | 1.025 | 1.009 |
| C3a | 22 | 54.55 | 20 | 74 | 24 | 1 | 96.83 | 99.25 | 99.98 | 1.034 | 1.020 | 1.007 |
| C3d | 22 | 54.55 | 20 | 74 | 21 | 1 | 97.50 | 99.11 | 99.96 | 1.037 | 1.024 | 1.005 |
| C4a | 22 | 36.36 | 28 | 78 | 39 | 1 | 99.80 | 99.89 | 99.97 | 1.014 | 1.008 | 1.004 |
| C4d | 22 | 36.36 | 28 | 78 | 42 | 2 | 99.83 | 99.92 | 99.99 | 1.022 | 1.008 | 1.003 |
| C5a | 22 | 45.45 | 24 | 70 | 30 | 1 | 99.68 | 99.85 | 99.95 | 1.022 | 1.012 | 1.008 |
| C5d | 22 | 45.45 | 24 | 70 | 32 | 1 | 99.62 | 99.79 | 99.90 | 1.027 | 1.020 | 1.009 |

Table 3: Results for Benchmark Instances (10 Runs Each)

| | Average Front Size | Average Run Time | Hypervolume (%) | | | Mult. Unary Epsilon | | |
|---|---|---|---|---|---|---|---|---|
| | | | Worst | Ave | Best | Worst | Ave | Best |
| Minimum | 11 | 0 | 92.03 | 97.08 | 98.03 | 1.355 | 1.110 | 1.017 |
| Maximum | 90 | 301 | 99.98 | 99.99 | 100.00 | 1.003 | 1.001 | 1.000 |
| **Average** | **35** | **71** | **98.32** | **99.24** | **99.80** | **1.045** | **1.019** | **1.006** |

Table 4: Summary of Results on Benchmark Instances

Looking at the best and worst runs of each instance, it is clear that there is more variance among the worst runs. The best runs for each instance all fall within a range of 98% to 100% of the optimal hypervolume, whereas the weakest runs generate anywhere between 92% and just under 100%. Nonetheless, the results are promising since even the single worst run still manages to attain over 92% of the optimal hypervolume. Figure 12 shows the normalized approximation set and the Pareto-optimal set of the run with the worst hypervolume.

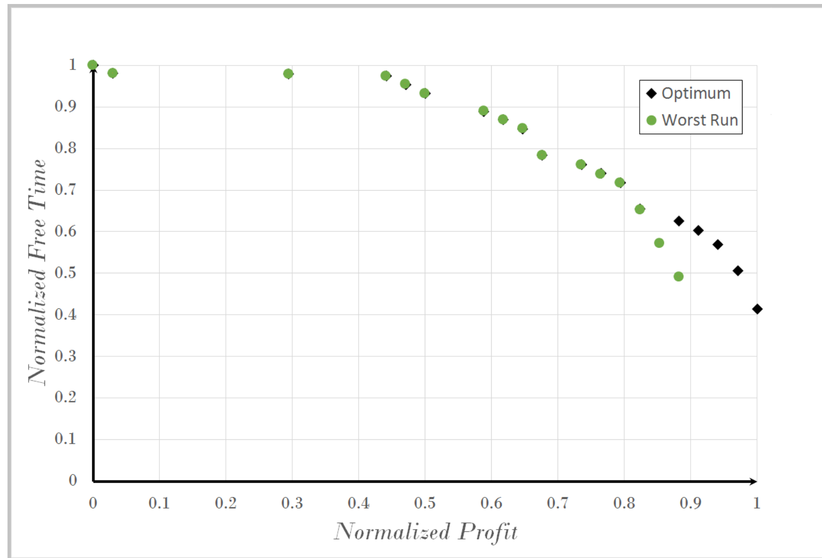


Figure 12: Approximation Set of Run with Worst Hypervolume %

It can be seen that even for this worst run, the algorithm finds the optimal solutions at the upper end of the front (high free time, low profit). This supports the notion that the upper area of the approximation set is generally easier to optimize for the PPP. At the other end of the front, the solutions are further from the optimum, and no solutions are found for some of the highest profit scores. However, looking at the other extreme, the best run on this instance (B1d) finds 99.97% of the optimum hypervolume, and the average is 98.09%, so it seems that finding a very good approximation set is more likely for this instance.

There appears to be more variance among the instances and runs when considering the worst epsilon indicator, which is rather high at 1.355. However, the unary epsilon indicator is based on the quality of the worst solution in an approximation set. Figure 13 shows the approximation set with the worst epsilon indicator.

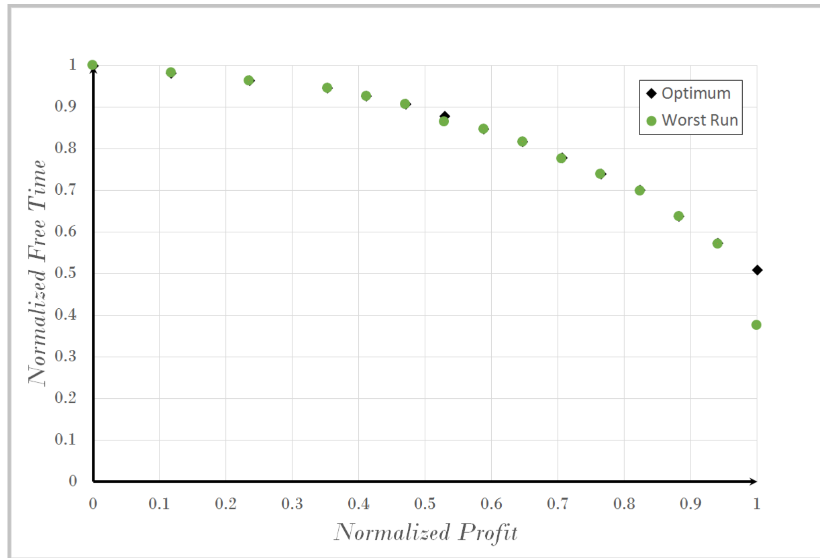


Figure 13: Approximation Set of Run with Worst Epsilon Indicator

Looking at the approximation set, it is clear that this "worst" set is in fact very efficient, except for the single highest profit value. This is an acceptable compromise, especially since the solutions with the highest profit are not only difficult to optimize, but in some cases difficult to attain at all, as in the previous instance.

Overall, there do not appear to be too many significant differences in performance between the schedules or transportation modes of the benchmark instances. Due to the small size of these instances, the run times are also all negligible.

## 5.4 Instances with Approximate Reference Sets

This section reports the results on test instances for which no exact optimal reference sets are known. Each instance was solved 10 times. In order to evaluate the potential effect of having limited the search with the parameter settings outlined in section 5.2, each instance was also solved once with much more generous settings:

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| $l_{exp}$ | 50 | $l_{ref}$ | $\infty$ |
| $l_{con}$ | $\infty$ | $r_{max}$ | 600 |

Table 5: Modified Parameter Values for Extended Runs

Although for most instances these extended runs produced slightly better solution sets, the final reference set used for these tests was the non-dominated set of solutions found over all 10 regular runs and the extended run. This reference set was used as the benchmark for evaluating the relative hypervolume and multiplicative unary epsilon indicators for the larger instances. Table 6 reports the results of the 10 regular runs for each instance, followed by a summary of all indicators in Table 7.

The experimental results indicate that the performance of the proposed metaheuristic is robust and consistent for much larger instances as well. Averaged over all instances, the gap between the best and worst average hypervolume is only about 1.4%, and never more than 7% for any single instance. The average hypervolume over all instances stands at over 99% of the respective instances' best known values. Likewise, the average epsilon indicator over all instances is 1.012, just slightly above the optimal value of 1.0. Even the worst of the worst runs manages to attain 93% of its respective instance's best found hypervolume, and the worst epsilon indicator is still only 1.062. Together, these indicators suggest that the solutions found by the proposed metaheuristic may be highly efficient and also well-distributed across the efficient frontier, with very little variance in quality from one run to the next. Unfortunately, it is not possible to draw strong conclusions about the *absolute* quality of the solutions without the optimal fronts as a reference.

Although the differences are small, a closer look at the data suggests that the instances from the A, B, D, and F groups may be somewhat easier to optimize. Their hypervolume indicators are consistently within 1% to 2% of the best known, and the average always above 99%. In contrast, instances from the C and E schedule groups result in somewhat more variable solution quality, particularly set E.

A similar split among the schedule groups is discernible in the average sizes of their approximation fronts. It can be seen in Figure 14 that instance sets C and E tend to have larger approximation fronts, with most of these instances having *at least* 60. On the other hand, it is rare for any instances from the other sets A, B, D, and F to have *more* than 60 solutions in an approximation set on average.

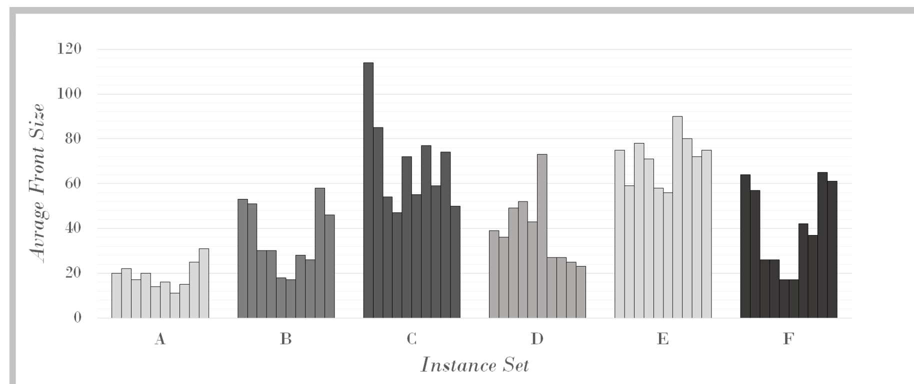


Figure 14: Average Front Size over 10 Runs

| Instance | Tasks | Chain % | Locations | Visits | Average Front Size | Average Run Time | Hypervolume (%) | | | Mult. Unary Epsilon | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Worst | Ave | Best | Worst | Ave | Best |
| A1a | 32 | 71.88 | 2244 | 17458 | 20 | 86 | 99.80 | 99.87 | 99.97 | 1.021 | 1.009 | 1.002 |
| A1b | 31 | 74.19 | 2244 | 17457 | 22 | 58 | 98.64 | 99.54 | 99.86 | 1.062 | 1.023 | 1.016 |
| A2a | 36 | 75.00 | 2117 | 19267 | 17 | 168 | 99.83 | 99.94 | 99.98 | 1.007 | 1.003 | 1.001 |
| A2b | 36 | 75.00 | 2117 | 19267 | 20 | 141 | 98.80 | 99.24 | 99.35 | 1.036 | 1.025 | 1.023 |
| A3a | 32 | 87.50 | 1930 | 15935 | 14 | 78 | 99.76 | 99.88 | 99.94 | 1.017 | 1.011 | 1.002 |
| A3b | 32 | 87.50 | 1930 | 15935 | 16 | 51 | 99.51 | 99.60 | 99.70 | 1.023 | 1.015 | 1.009 |
| A4a | 26 | 69.23 | 1931 | 15861 | 11 | 97 | 99.92 | 99.97 | 99.99 | 1.005 | 1.001 | 1.000 |
| A4b | 26 | 69.23 | 1931 | 15861 | 15 | 41 | 99.60 | 99.77 | 99.94 | 1.026 | 1.019 | 1.008 |
| A5a | 30 | 76.67 | 222 | 3909 | 25 | 60 | 99.91 | 99.95 | 99.98 | 1.006 | 1.003 | 1.001 |
| A5b | 29 | 79.31 | 222 | 3908 | 31 | 59 | 99.54 | 99.65 | 99.84 | 1.035 | 1.015 | 1.005 |
| B1a | 27 | 14.81 | 109 | 1935 | 53 | 6 | 99.97 | 99.99 | 99.99 | 1.002 | 1.001 | 1.001 |
| B1c | 26 | 15.38 | 109 | 1934 | 51 | 5 | 99.92 | 99.95 | 99.97 | 1.006 | 1.004 | 1.003 |
| B2a | 28 | 14.29 | 231 | 3869 | 30 | 7 | 95.51 | 99.17 | 99.94 | 1.029 | 1.010 | 1.002 |
| B2c | 27 | 14.81 | 231 | 3868 | 30 | 3 | 95.24 | 95.73 | 99.91 | 1.015 | 1.012 | 1.006 |
| B3a | 28 | 17.86 | 2184 | 8513 | 18 | 11 | 95.69 | 99.51 | 99.98 | 1.030 | 1.007 | 1.001 |
| B3c | 27 | 18.52 | 2184 | 8512 | 17 | 4 | 95.02 | 97.87 | 99.92 | 1.057 | 1.020 | 1.004 |
| B4a | 27 | 14.81 | 2185 | 6575 | 28 | 5 | 99.77 | 99.87 | 99.95 | 1.025 | 1.021 | 1.006 |
| B4c | 27 | 14.81 | 2185 | 6575 | 26 | 4 | 99.80 | 99.89 | 99.97 | 1.031 | 1.013 | 1.002 |
| B5a | 26 | 19.23 | 1889 | 5844 | 58 | 12 | 99.94 | 99.95 | 99.96 | 1.004 | 1.003 | 1.002 |
| B5c | 25 | 20.00 | 1889 | 5843 | 46 | 8 | 99.68 | 99.77 | 99.87 | 1.032 | 1.018 | 1.010 |
| C1a | 62 | 67.74 | 1106 | 4227 | 114 | 300 | 99.54 | 99.81 | 99.94 | 1.019 | 1.010 | 1.005 |
| C1e | 61 | 68.85 | 1106 | 4226 | 85 | 148 | 98.74 | 99.18 | 99.72 | 1.017 | 1.012 | 1.007 |
| C2a | 63 | 65.08 | 1315 | 6150 | 54 | 298 | 99.60 | 99.89 | 99.95 | 1.012 | 1.007 | 1.003 |
| C2e | 63 | 65.08 | 1315 | 6150 | 47 | 114 | 97.96 | 98.71 | 99.88 | 1.020 | 1.014 | 1.010 |
| C3a | 63 | 66.67 | 722 | 3034 | 72 | 252 | 98.94 | 99.07 | 99.64 | 1.013 | 1.009 | 1.005 |
| C3e | 62 | 67.74 | 722 | 3033 | 55 | 95 | 97.81 | 99.17 | 99.91 | 1.017 | 1.014 | 1.010 |
| C4a | 60 | 63.33 | 1100 | 5246 | 77 | 300 | 99.61 | 99.86 | 99.96 | 1.016 | 1.008 | 1.005 |
| C4e | 60 | 63.33 | 1100 | 5246 | 59 | 113 | 97.12 | 98.15 | 99.75 | 1.023 | 1.015 | 1.010 |
| C5a | 62 | 64.52 | 437 | 2358 | 74 | 291 | 97.92 | 98.86 | 99.93 | 1.019 | 1.016 | 1.008 |
| C5e | 62 | 64.52 | 437 | 2358 | 50 | 92 | 96.37 | 97.21 | 99.85 | 1.024 | 1.016 | 1.010 |
| D1d | 35 | 25.71 | 2598 | 12761 | 39 | 192 | 99.91 | 99.92 | 99.94 | 1.004 | 1.003 | 1.002 |
| D1e | 35 | 25.71 | 2598 | 12761 | 36 | 95 | 99.77 | 99.86 | 99.91 | 1.012 | 1.005 | 1.002 |
| D2d | 40 | 30.00 | 2439 | 26853 | 49 | 300 | 99.52 | 99.69 | 99.81 | 1.022 | 1.013 | 1.007 |
| D2e | 40 | 30.00 | 2439 | 26853 | 52 | 300 | 99.38 | 99.54 | 99.77 | 1.028 | 1.022 | 1.010 |
| D3d | 42 | 30.95 | 2035 | 65619 | 43 | 300 | 99.87 | 99.92 | 99.96 | 1.006 | 1.003 | 1.002 |
| D3e | 42 | 30.95 | 2035 | 65619 | 73 | 300 | 99.13 | 99.50 | 99.75 | 1.024 | 1.015 | 1.011 |
| D4d | 42 | 30.95 | 2033 | 43324 | 27 | 188 | 98.92 | 99.27 | 99.97 | 1.014 | 1.012 | 1.004 |
| D4e | 42 | 30.95 | 2033 | 43324 | 27 | 119 | 98.91 | 99.11 | 99.97 | 1.026 | 1.018 | 1.004 |
| D5d | 40 | 25.00 | 2048 | 29676 | 25 | 166 | 93.66 | 96.76 | 99.88 | 1.020 | 1.016 | 1.002 |
| D5e | 40 | 25.00 | 2048 | 29676 | 23 | 82 | 93.18 | 98.51 | 99.71 | 1.032 | 1.019 | 1.008 |
| E1d | 65 | 58.46 | 808 | 4253 | 75 | 300 | 98.46 | 99.26 | 99.91 | 1.017 | 1.010 | 1.006 |
| E1e | 65 | 58.46 | 808 | 4253 | 59 | 106 | 97.20 | 98.36 | 99.75 | 1.022 | 1.014 | 1.006 |
| E2d | 71 | 66.20 | 765 | 4082 | 78 | 300 | 98.69 | 99.36 | 99.77 | 1.021 | 1.011 | 1.008 |
| E2e | 71 | 66.20 | 765 | 4082 | 71 | 288 | 97.40 | 98.76 | 99.68 | 1.020 | 1.015 | 1.011 |
| E3d | 72 | 61.11 | 725 | 4640 | 58 | 300 | 95.99 | 98.12 | 99.48 | 1.027 | 1.015 | 1.007 |
| E3e | 71 | 61.97 | 725 | 4639 | 56 | 300 | 93.83 | 97.08 | 99.21 | 1.047 | 1.026 | 1.012 |
| E4d | 74 | 64.86 | 1002 | 5342 | 90 | 300 | 97.10 | 98.20 | 99.25 | 1.024 | 1.017 | 1.013 |
| E4e | 73 | 65.75 | 1002 | 5341 | 80 | 300 | 96.93 | 98.55 | 99.65 | 1.021 | 1.015 | 1.011 |
| E5d | 70 | 61.43 | 755 | 4831 | 72 | 300 | 97.47 | 98.55 | 99.75 | 1.016 | 1.012 | 1.008 |
| E5e | 70 | 61.43 | 755 | 4831 | 75 | 298 | 96.05 | 98.13 | 99.62 | 1.026 | 1.015 | 1.006 |
| F1d | 39 | 28.21 | 2947 | 13184 | 64 | 228 | 99.80 | 99.85 | 99.89 | 1.014 | 1.009 | 1.004 |
| F1e | 39 | 28.21 | 2947 | 13184 | 57 | 148 | 99.62 | 99.72 | 99.83 | 1.021 | 1.016 | 1.009 |
| F2d | 30 | 23.33 | 2487 | 12983 | 26 | 40 | 99.95 | 99.96 | 99.97 | 1.003 | 1.002 | 1.001 |
| F2e | 30 | 23.33 | 2487 | 12983 | 26 | 27 | 99.83 | 99.90 | 99.92 | 1.009 | 1.006 | 1.004 |
| F3d | 29 | 27.59 | 99 | 1933 | 17 | 3 | 99.88 | 99.93 | 99.97 | 1.004 | 1.002 | 1.001 |
| F3e | 29 | 27.59 | 99 | 1933 | 17 | 3 | 99.52 | 99.66 | 99.75 | 1.012 | 1.011 | 1.006 |
| F4d | 35 | 17.14 | 2629 | 14762 | 42 | 110 | 99.84 | 99.90 | 99.96 | 1.005 | 1.003 | 1.002 |
| F4e | 35 | 17.14 | 2629 | 14762 | 37 | 64 | 99.68 | 99.81 | 99.90 | 1.017 | 1.011 | 1.003 |
| F5d | 32 | 28.13 | 2480 | 12405 | 65 | 74 | 99.83 | 99.88 | 99.92 | 1.006 | 1.005 | 1.003 |
| F5e | 32 | 28.13 | 2480 | 12405 | 61 | 54 | 99.68 | 99.75 | 99.83 | 1.013 | 1.009 | 1.003 |

Table 6: Results for Larger Instances (10 Runs Each)

|  | Average Front Size | Average Run Time | Hypervolume (%) | | | Mult. Unary Epsilon | | |
|---|---|---|---|---|---|---|---|---|
|  |  |  | Worst | Ave | Best | Worst | Ave | Best |
| Minimum | 11 | 3 | 93.18 | 95.73 | 99.21 | 1.062 | 1.026 | 1.023 |
| Maximum | 114 | 300 | 99.97 | 99.99 | 99.99 | 1.002 | 1.001 | 1.000 |
| **Average** | **46** | **142** | **98.47** | **99.23** | **99.84** | **1.020** | **1.012** | **1.006** |

Table 7: Summary of Results on Larger Instances

There are also clear differences in the run times for these instance classes (Figure 15). Here too, schedule groups A, B, and F fall into a similar category, with most of their average run times being below 100 seconds, i.e. less than two thirds of the limit. On the other hand, schedule groups C, D, and E barely have any instances with run times *below* 100 seconds, and in fact, about half of them time out at the 300 second limit. Interestingly, instances with faster transportation modes (e.g. car (a) vs. public transportation (c), or cycling (d) vs. walking (e)) are usually solved more quickly.

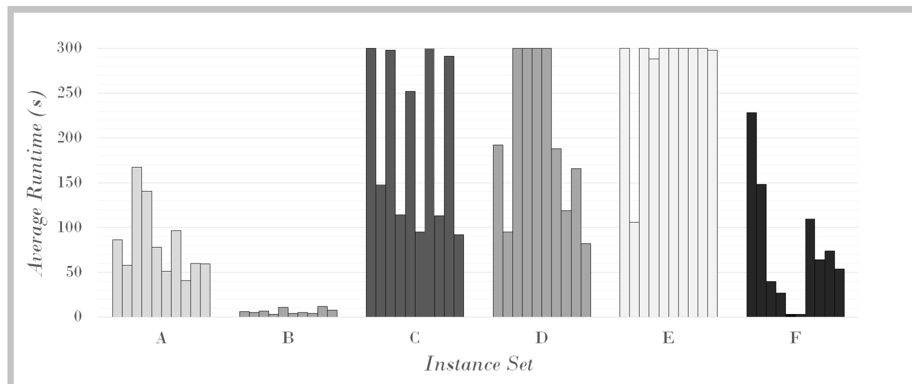


Figure 15: Average Run Time over 10 Runs, in Seconds

Putting this data together, it appears that schedule sets C and E are more difficult than the others. Their hypervolume and run time are generally worse than for the instances in the A, B, D, and F groups. Schedule set D is somewhat of an exception as the qualitative results are as good as those in the easier group, but the run time is clearly much higher and similar to the more difficult instances.

An closer examination of the instance characteristics reveals some insights into instance difficulty. Schedule sets C and E have a markedly higher number of tasks to be planned, ranging from 60 to over 70. In contrast, the instances from the A, B, D, and F sets generally have around half as many tasks, from 25 to 35 in most cases. In addition, schedule sets C and E also have considerably higher shares of tasks connected by precedence and time gap constraints, between approx. 58% and 69%. Most of the easier instance groups have no more than about 30% of their tasks connected by such complicating factors. Group A is an exception to this trend.

However, it is noteworthy that although sets C and E have the most tasks and precedence relations, they have by far the fewest locations - most C and E instances have well below 1000 locations total, whereas almost all of the other instance sets have well over 2000.

One counter-intuitive result is that the more difficult sets C and E actually have the fewest feasible visits, with just over 6000 for the largest instance, compared to well over 10000 for most of the others. It was expected that the number of visits would be the determining factor for difficulty, since this figure determines the size of the search space. However, it appears that the number of visits is a better indicator for the run time. In particular, it explains why schedule set D has such high run times despite being relatively easier to solve - instances of group D have by far the most visits, including some with 26000, 43000, and even 65000, compared to no more than 20000 for the largest of the remaining instances. This puts a significant computational burden on every insertion operation.

Despite all these differences, the algorithm converges quickly to the respective solution sets. This was a major objective for the practical application, as the search could be stopped at any time by the decision-maker. Figure 16 plots the hypervolume as a percentage of the final found hypervolume per instance, relative to the progress of the search. The two curves represent the average and the worst growth over all instances. It can be seen that, on average, the algorithm quickly finds more than 95% of the final hypervolume within about 10% of the search, and still around 90% in the worst case. The rest of the search focuses on the final 5% to 10% of the solution quality. This suggests that even for the most time-consuming instances which timed out at 300 seconds, the quality of the returned solution would have been only slightly inferior if the search had been interrupted after only 30 seconds.

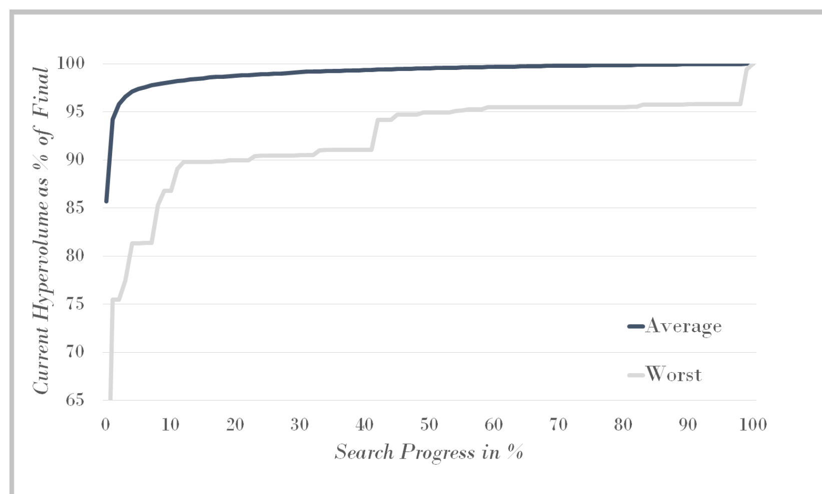


Figure 16: Hypervolume Growth by Search Progress

# Chapter 6

# Conclusion

The present work introduces the Personal Planning Problem (PPP), an extension of the classic OPTW. Motivated by the planning challenges faced by people with complex and flexible schedules, the PPP extends the routing and scheduling aspects of the OPTW with a number of real-life constraints, including multiple time windows, precedence relations between tasks, and minimum/maximum time delays between activities. A bi-objective formulation using total profit and total free time captures the inherent trade-off between scheduling more tasks, and having more leisure time.

A metaheuristic based on Large Neighborhood Search is proposed. The algorithm explores the search space in many different directions, maintains a set of non-dominated solutions, and iteratively refines this set to improve the solutions' objective values. Large parts of the objective space are explored at every iteration by evaluating the efficiency of all intermediate solutions. A modular framework for creating a variety of destroy and repair operators is presented, based on combining their defining characteristics. Classic procedures used for updating the timing variables in routing problems are adapted to handle the additional constraints of the PPP.

Computational experiments show that the algorithm can efficiently handle the many side-constraints, and produces highly efficient and representative solution sets in a consistent and reliable way. Benchmark instances are solved nearly to optimality in less than 2 seconds on average. Tests on larger instances confirm the consistency of the solution quality, with an average deviation of around only 1.5% from the best known hypervolume over 60 instances and 600 runs. Likewise, the very small range of the multiplicative unary epsilon values indicates that the solutions in the approximation sets are all of similar quality compared to the best known solutions, i.e. the algorithm does not favor the optimization of one objective over the other. Results for instance sizes ranging from 2000 to over 65000 feasible visits indicate that the algorithm scales well. Despite differences in total computation

time, the algorithm converges quickly to the optimal or best known solution sets, so that high quality solutions are available even if the search is terminated early by the decision maker.

Although the PPP has many more constraints than the OPTW, the feasible solution space is not necessarily smaller. The combinations of multiple locations per task and multiple time windows per task and per location introduce many possibilities. An analysis of instance characteristics and obtained solution quality reveals that the difficulty of PPP instances may depend primarily on the number of tasks and on how many of them are linked by precedence and time delay constraints. The number of locations or the total number of feasible visits appears to have more influence on the run time than on solution quality. This suggests that the scheduling aspect of the PPP may be more difficult to optimize than the routing aspect, or that the former determines to some extent the difficulty of the latter.

Since the computational results are very satisfactory with the basic LNS framework, additional adaptive elements are forgone in the practical application in favor of reducing the number of parameters and the required tuning. However, there is potential for synergy between the modular implementation proposed for the destroy and repair operators and an adaptive LNS framework. This may be a promising avenue for further research.

# Bibliography

[1] Azi N., Gendreau M., Potvin J-Y. (2014): An adaptive large neighborhood search for a vehicle routing problem with multiple routes, *Computers & Operations Research*, 41, 167-173.

[2] Bent R., Van Hentenryck P. (2004): A two-stage hybrid local search for the vehicle routing problem with time windows, *Transportation Science*, 38(4), 515-530.

[3] Bent R., Van Hentenryck P. (2006): A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows, *Computers & Operations Research*, 33, 875-893.

[4] Boussier S., Feillet D., Gendreau M. (2007): An exact algorithm for team orienteering problems, *4OR*, 5, 211-230.

[5] Coello Coello C.A. (1999): A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques, *Knowledge and Information Systems*, 1, 269-308.

[6] Cordeau J-F., Laporte G., Pasin F., Ropke S. (2010): Scheduling technicians and tasks in a telecommunications company, *Journal of Scheduling*, 13, 393-409.

[7] Côté J-F., Gendreau M., Potvin J-Y. (2012): Large Neighborhood Search for the Pickup and Delivery Traveling Salesman Problem with Multiple Stacks, *Networks*, 60(1), 19-30.

[8] Cura T. (2014): An artificial bee colony algorithm approach for the team orienteering problem with time windows, *Computers & Industrial Engineering*, 74, 270-290.

[9] Demir E., Bektas T., Laporte G. (2012): An adaptive large neighborhood search heuristic for the Pollution-Routing Problem, *European Journal of Operational Research*, 212, 346-359.

[10] Feillet D., Dejax P., Gendreau M. (2005): Traveling Salesman Problems with Profits, *Transportation Science*, 39(2), 188-205.

[11] Gavalas D., Konstantopoulos Ch., Mastakas K., Pantziou G., Tasoulas Y. (2013): Cluster-Based Heuristics for the Team Orienteering Problem with Time Windows, In: *Lecture Notes in Computer Science*, 7933, 390-401.

[12] Hemmelmayr V., Cordeau J-F., Crainic T.G. (2012): An adaptive large neighborhood search heuristic for Two-Echelon Vehicle Routing Problems arising in city logistics, *Computers & Operations Research*, 39, 3215-3228.

[13] Hertz A., Widmer M. (2003): Guidelines for the Use of Meta-Heuristics in Combinatorial Optimization, *European Journal of Operational Research*, 151, 247-252.

[14] Hu Q., Lim A. (2014): An iterative three-component heuristic for the team orienteering problem with time windows, *European Journal of Operational Research*, 232, 276-286.

[15] Jain S., Van Hentenryck P. (2011): Large Neighborhood Search for Dial-a-Ride Problems, *Principles and Practice of Constraint Programming - Proceedings of the 17th International Conference, CP 2011*, 400-413.

[16] Kovacs A.A., Parragh S.N., Doerner K.F., Hartl R.F. (2012): Adaptive large neighborhood search for service technician routing and scheduling problems, *Journal of Scheduling*, 15, 579-600.

[17] Kilby P., Prosser P., Shaw P. (2000): A Comparison of Traditional and Constraint-based Heuristic Methods on Vehicle Routing Problems with Side Constraints, *Constraints*, 5, 389-414.

[18] Kim B.-I., Li H., Johnson A.L. (2013): An augmented large neighborhood search method for solving the team orienteering problem, *Expert Systems with Applications*, 40, 3065-3072.

[19] Labadie N., Melechovský J., Wolfler Calvo R. (2011): Hybridized evolutionary local search algorithm for the team orienteering problem with time windows, *Journal of Heuristics*, 17, 729-753.

[20] Labadie N., Mansini R., Melechovský J., Wolfler Calvo R. (2012): The Team Orienteering Problem with Time Windows: An LP-based Granular Variable Neighborhood Search, *European Journal of Operational Research*, 220, 15-27.

[21] Lehuédé F., Masson R., Parragh S.N., Péton O., Tricoire F. (2014): A multi-criteria large neighbourhood search for the transportation of disabled people, *Journal of the Operational Research Society*, 65, 94-107.

[22] Lin S-W., Yu V.F. (2012): A simulated annealing heuristic for the team orienteering problem with time windows, *European Journal of Operational Research*, 217, 15-27.

[23] Montemanni R., Gambardella L. (2009): Ant colony system for team orienteering problems with time windows, *Foundations of Computing and Decision Sciences*, 34(4), 287-306.

[24] Montemanni R., Weyland D., Gambardella L. (2009): An Enhanced Ant Colony System for the Team Orienteering Problem with Time Windows, In: *2011 International Symposium on Computer Science and Society (ISCCS), IEEE*, 381-384.

[25] Parragh S.N., Schmid V. (2013): Hybrid column generation and large neighborhood search for the dial-a-ride problem, *Computers & Operations Research*, 40, 490-497.

[26] Parragh S.N., Tricoire F. (2014): Branch-and-bound for bi-objective optimization, Retrieved from: `http://www.optimization-online.org/DB_HTML/2014/07/4444.html`.

[27] Pisinger D., Ropke S. (2007): A general heuristic for vehicle routing problems, *Computers & Operations Research*, 34, 2403-2435.

[28] Pisinger D., Ropke S. (2010): Large Neighborhood Search, In: *Handbook of Metaheuristics, Second Edition*, Gendreau M., Potvin J.-Y. (Eds.), Springer, 399-419.

[29] Qu Y., Bard J.F. (2012): A GRASP with adaptive large neighborhood search for pickup and delivery problems with transshipment, *Computers & Operations Research*, 39, 2439-2456.

[30] Rodríguez B., Molina J., Pérez F., Caballero R. (2012): Interactive design of personalized tourism routes, *Tourism Management*, 33, 926-940.

[31] Ropke S., Pisinger D. (2006): A unified heuristic for a large class of Vehicle Routing Problems with Backhauls, *European Journal of Operational Research*, 171, 750-775.

[32] Ropke S., Pisinger D. (2006): An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows, *Transportation Science*, 40(4), 455-472.

[33] Savelsbergh M.V.P. (1992): The Vehicle Routing Problem with Time Windows: Minimizing Route Duration, *ORSA Journal on Computing*, 4(2), 146-154.

[34] Schaus P., Hartert R. (2013): Multi-objective Large Neighborhood Search, In: emphLecture Notes in Computer Science, 8124, 611-627.

[35] Schilde M., Doerner K.F., Hartl R.F., Kiechle G. (2009): Metaheuristics for the bi-objective orienteering problem, *Swarm Intelligence*, 3, 179-201.

[36] Schmid V. (2014): Hybrid large neighborhood search for the bus rapid transit route design problem, *European Journal of Operational Research*, 238, 427-437.

[37] Schrimpf G., Schneider J., Stamm-Wilbrandt H., Dueck G. (2000): Record Breaking Optimization Results Using the Ruin and Recreate Principle, *Journal of Computational Physics*, 159, 139-171.

[38] Shaw P. (1998): Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems, In: *Fourth International Conference on Principles and Practice of Constraint Programming (CP'98)*, 417-431.

[39] Souffriau W., Vansteenwegen P., Vanden Berghe G., Van Oudheusden D. (2013): The Multiconstraint Team Orienteering Problem with Multiple Time Windows, *Transportation Science*, 47(1), 53-63.

[40] Tricoire F., Romauch M., Doerner K.F., Hartl R.F. (2010): Heuristics for the multi-period orienteering problem with time windows, *Computers & Operations Research*, 37, 351-67.

[41] Tricoire F. (2012): Multi-directional local search, *Computers & Operations Research*, 39, 3089-3101.

[42] Vansteenwegen P., Souffriau W., Berghe G.V., Van Oudheusden D. (2009): Iterated local search for the team orienteering problem with time windows, *Computers & Operations Research*, 36, 3281-3290.

[43] Vansteenwegen P., Souffriau W., Van Oudheusden D. (2011): The Orienteering Problem: A Survey, *European Journal of Operational Research*, 209, 1-10.

[44] Wy J., Kim B-I., Kim S. (2013): The rollon-rolloff waste collection vehicle routing problem with time windows, *European Journal of Operational Research*, 224, 466-476.

[45] Zhou A., Qu B-Y., Li H., Zhao S-Z., Suganthan P.N., Zhang Q. (2011): Multiobjective Evolutionary Algorithms: A survey of the state of the art, *Swarm and Evolutionary Computation*, 1, 32-49.

## Abstract

People with complex schedules, such as self-employed people with multiple projects and clients, have to plan their time wisely to strike a balance between their professional activities and their private leisure time. An automated schedule optimizer with built-in routing functionality can provide support by offering good suggestions for when, where, and in what order to get tasks done. In doing so, such a planner should consider the individual user preferences with regard to the trade-off between professional activities and leisure.

The development of such an optimization application was one of the aims of a research project initiated by the Austrian Research Promotion Agency. Carried out in cooperation with the Austrian Institute of Technology (AIT), this thesis introduces and solves the *Personal Planning Problem* (PPP), a combined scheduling and routing model that captures the real-life challenge faced by people with complex and flexible schedules.

The PPP extends the *Orienteering Problem with Time Windows* (OPTW), where the nodes represent the tasks to be scheduled and their locations. In contrast to the OPTW, the PPP formulation allows tasks to be done at one of several possible locations and during one of several possible time windows. In addition, precedence relations between tasks, as well as minimum and maximum time delays between related tasks, are taken into account. In order to capture the trade-off between scheduling more tasks and enjoying more leisure time, a bi-objective model is formulated so that a diverse set of efficient and varied schedules can be presented to the decision maker.

A metaheuristic based on *Large Neighborhood Search* is proposed for solving the PPP. The solution space is explored by iteratively destroying different parts of a solution, and then reconstructing it in various ways. Several destroy and repair operators are used to diversify and intensify the search. Some common procedures from the literature are adapted to handle the specific constraints of the PPP.

The proposed metaheuristic is implemented and tested on a set of benchmark instances provided by the Austrian Institute of Technology. Computational experiments show that the algorithm is effective, reliable, and scales well with increasing instance size. Near-optimal sets of schedules are consistently found for the instances for which optimal solutions are known, and the solution quality is consistent for larger instances as well. Since the algorithm generates a representative set of non-dominated solutions, the decision maker can compare different schedules without re-running the optimization, and the various preferences of different decision makers can be taken into account.

# Kurzfassung

Menschen mit komplexen Tagesabläufen, wie beispielsweise mobile Selbstständige, müssen sich ihre Zeit sinnvoll einteilen, um alle beruflichen und privaten Termine sowie Freizeitaktivitäten wahrnehmen zu können. Ein automatischer Tagesplaner, der die effiziente Planung von Terminen und Aktivitäten mit dazugehöriger Routenplanung übernimmt, kann diese Zielgruppe unterstützen indem Vorschläge für einen effizienten Wochenplan gemacht werden. Wichtig ist hier, dass der Planer die nach Benutzer individuelle gewünschte Balance zwischen beruflichen Aktivitäten und Freizeit berücksichtigt.

Ein Ziel im Rahmen eines Forschungsprojektes der Österreichischen Forschungsförderungsgesellschaft war die Entwicklung einer solchen Optimierungssoftware. Die vorliegende Arbeit, verfasst im Rahmen einer Zusammenarbeit mit dem Austrian Institute of Technology (AIT), präsentiert und löst das Personal Planning Problem (PPP) - ein mathematisches Modell welches sowohl die zeitlichen als auch die örtlichen Aspekte einer realistischen Terminplanung berücksichtigt.

Das PPP erweitert das bekannte *Orienteering Problem with Time Windows* (OPTW). Die Knoten repräsentieren dabei die zu planende Aufgaben und die Orte an denen sie ausgeführt werden. Im Unterschied zum OPTW, ermöglicht die PPP Formulierung, dass eine Aufgabe an mehreren Standorten und zu verschiedenen Zeitfenstern geplant werden kann. Weiters werden Reihenfolgebedingungen und zeitliche Mindest- und/oder Maximalabstände zwischen den Aufgaben berücksichtigt. Um die Balance zwischen mehr Aktivitäten und mehr Freizeit zu berücksichtigen, wird eine bikriterielle Formulierung des Problems gelöst. So kann eine repräsentative Menge von effizienten Zeitplänen erstellt und dem Entscheidungsträger präsentiert werden.

Das vorgestellte Lösungsverfahren basiert auf *Large Neighborhood Search.* Der Lösungsraum wird erkundet indem verschiedene Teile eines Plans iterativ zerstört und anschließend wieder rekonstruiert werden. Der Algorithmus setzt verschiedene Zerstörungs- und Reparaturoperatoren ein um die Suche zu diversifizieren bzw. zu verstärken. Bekannte Verfahren aus der Literatur wurden an die problemspezifischen Nebenbedingungen des PPP angepasst.

Die präsentierte Metaheuristik wurde implementiert und auf problemspezifischen, vom AIT entworfenen, Vergleichsinstanzen getestet. Der Algorithmus erweist sich als effektiv und zuverlässig für alle untersuchten Problemgrößen. Für die kleinen Testinstanzen, für welche die exakte Pareto-Menge bekannt ist, findet die Metaheuristik nahezu immer die exakten Lösungen und für größere Instanzen ist die Qualität der Ergebnisse konsistent. Der Algorithmus erzeugt stets repräsentative Fronten von Pareto-effizienten Lösungen - dadurch können diverse Terminpläne ohne einen Neustart des Verfahrens verglichen, und die unterschiedlichen Präferenzen von verschiedenen Entscheidungsträgern berücksichtigt werden.

# Peter Matl

*Curriculum Vitae*

✉ *peter.l.matl@gmail.com*

## Education

2012–2014 **Master of Science**, *University of Vienna*.
International Business Administration
Operations Research
Transportation Logistics

2011 **International Exchange Semester**, *National University of Singapore (NUS)*.

2008–2012 **Bachelor of Business Studies**, *Vienna University of Economics and Business*.
International Business Track
Cross-Functional Management

2007–2008 **Translation Studies**, *University of Vienna*.
Translation & Interpretation
English, German, Polish

2006 **Cambridge CELTA**, *International House Cracow*.
Certificate in English Language Teaching to Adults

2002–2006 **International Baccalaureate Diploma**.
Stuyvesant High School, New York City, USA
Liceum V, Wroclaw, Poland

## Master's Thesis

Title *A Large Neighborhood Search Metaheuristic for the Personal Planning Problem*

Supervisors O.Univ.Prof. Dipl-Ing. Dr. Richard F. Hartl, Dr. Fabien Tricoire

Written as part of a collaborative project with the Austrian Institute of Technology with the aim of developing an intelligent schedule planner with built-in routing functionality. The thesis work focused on the design and implementation of an optimization algorithm for a prototype mobile application.

## Bachelor's Thesis

Title *Intercultural Differences in New Product Development: A Comparative Review*

Supervisor Dr. Elisabeth Götze

A meta-analysis of the success factors for new product innovation and development in an international context. The aim of the thesis was to survey the relatively fragmented empirical literature on the subject, find commonalities, and draw conclusions from the reported evidence.

## Awards

2014 Performance Scholarship of the University of Vienna for the 2012/13 academic year

## Work Experience

2007–Present **Freelance Translator**, *German to English*.
Various business and economic topics:

- location consulting
- shopping center optimization
- market analysis
- brand management
- demographics
- consumer expenditures
- press releases
- white papers

## Computer skills

Data Proc. LaTeX, Microsoft Office
Programming JAVA, C++
Other IBM CPLEX, SPSS

## Languages

English **Native speaker**
Polish **Native speaker**
German **Advanced (C2)**
Spanish **Intermediate (B2)**
French **Basic (A2)**