Brandenburg
University of Technology
Cottbus - Senftenberg

Branch-and-Refine for Solving Time-Dependent Problems

Fabian Gnegel
Armin Fügenschuh

Cottbus Mathematical Preprints
COMP # 13(2020)

Branch-and-Refine for Solving Time-Dependent Problems

Fabian Gnegel Armin Fügenschuh

June 15, 2020

Abstract

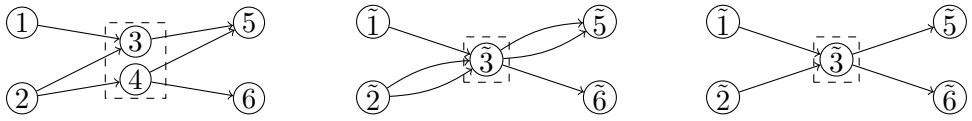
One of the standard approaches for solving time-dependent discrete optimization problems, such as the travelling salesman problem with time-windows or the shortest path problem with time-windows is to derive a so-called time-indexed formulation. If the problem has an underlying structure that can be described by a graph, the time-indexed formulation is usually based on a different, extended graph, commonly referred to as the time-expanded graph. The time-expanded graph can often be derived in such a way that all time constraints are incorporated in its topology, and therefore algorithms for the corresponding time-independent variant become applicable. The downside of this approach is, that the sets of vertices and arcs of the time-expanded graph are much larger than the ones of the original graph. In recent works, however, it has been shown that for many practical applications a partial graph expansion, that might contain time infeasible paths, often suffices to find a proven optimal solution. These approaches, instead, iteratively refine the original graph and solve a relaxation of the time-expanded formulation in each iteration. When the solution of the current relaxation is time feasible an optimal solution can be derived from it and the algorithm terminates. In this work we present new ideas, that allow for the propagation of information about the optimal solution of a coarser graph to a more refined graph and show how these can be used in algorithms, which are based on graph refinement. More precisely we present a new algorithm for solving Mixed Integer Linear Program (MILP) formulations of time-dependent problems that allows for the graph refinement to be carried out during the exploration of the branch-and-bound tree instead of restarting whenever the optimal solution was found to be infeasible. For demonstrating the practical relevance of this algorithm we present numerical results on its application to the shortest path problem with time-windows and the traveling salesman problem with time-windows.

Keywords: Graph Refinement, Branch-and-Bound, Shortest Path Problem with Time-Windows, Traveling Salesman Problem with Time-Windows.

1 Introduction

Many classical problems of Operations Research such as the shortest path problem (SPP), the traveling salesman problem (TSP) or the vehicle routing problem have an underlying structure that can be formulated as a mixed integer linear program (MILP) derived from a directed graph $G = (N, A)$. In routing problems the set of vertices N is used to represent locations and the set of arcs A is used to indicate whether there is a direct connection between two locations. In most cases a complete description of the problem also requires the specification of parameters for each vertex and each arc. For the vertices these can be for example time-windows, processing times, or demands/supplies of a certain commodity. For the set of arcs distances, traveling costs, traveling times, or capacities are often relevant instance parameters. In this work we restrict ourselves to including time constraints, i.e., time-windows and traveling times. One approach to include these constraints is to formulate the problem not on the graph G, but on a graph, often referred to as a time-expanded graph (see for example Skutella [26]). The nodes of the time-expanded graph

not only represent a location, but a location at a point of time and its arcs are only allowed to connect nodes, if the time layer associated with the head of the arc is later than the time layer associated the tail plus the traveling time of the arc. A major difficulty of applying this approach lies in the derivation of the time-expanded graph, more precisely, finding the pairs of location and time that have to be included in the graph. Time flows continuously, so it is impossible to include all of them in a finite graph. In order to derive a model with only finitely many nodes, a common approach is to consider only a finite time-span T and re-scale the traveling times and time-windows so that they are integers. If there is benefit to traveling slower than the traveling time and no cost for idle time, it suffices to use only time layers that are integers as well. Even with this requirement a completely time-expanded graph containing all feasible tours can have up to $T \cdot |N|$ many nodes. Now, a helpful observation is that the time-expanded graph does not have to include all nodes that are part of feasible tours, but only those that are part of optimal tours (actually it would be sufficient to find the arrival times of one optimal tour). Finding all of these nodes is as difficult as solving the original problem, but nevertheless for some applications, it is already helpful to determine some of the nodes that cannot be part of an optimal tour and remove them from the time-expanded graph in a preprocessing routine. Whereas in practise, preprocessing can reduce computation times significantly, some instances can remain notoriously difficult to solve. The approach we are going to use in this work tackles the problem of constructing the time-expanded graph in a different way and has recently been applied successfully to a variety of time-dependent problems.Instead of starting with a large graph and removing nodes, the idea is to start with a graph that underestimates the traveling times, but is much smaller than the time-expanded graph with correct traveling times. We do this by aggregating the nodes of the time-expanded graph to pairings of locations and time-intervals (instead of singular arrival times). While solutions of the MILPs derived from these graphs may be time-infeasible, i.e., violate some of the time-windows, they can be used to heuristically determine location-time pairings that are likely to be part of an optimal solution. By splitting the time-intervals of the nodes of the time-expanded graph at these points and adjusting the set of arcs, the graph can be expanded iteratively until the solution of the MILP formulation associated with it is time-feasible, i.e., fulfilling all time-window constraints. The graphs generated in this way only differ locally. In this work, we show that the similarity of the graphs transfers to the MILPs derived from them.



(a) (b) (c)

Figure 1: Illustration of a node contraction in a graph

Although conceptually we start with a coarse graph and refine it, the ideas in this work are better illustrated when going from a fine graph to a contracted graph. Consider the graphs (a),(b), and (c) given in Figure 1. Here, the graph in (c) can be obtained from the one in (a) by contracting the nodes 3 and 4 and (b) is an intermediate multigraph. In this example it is easy to see, that any path in (a) has a representative in (c). However, the path $\tilde{1} \to \tilde{3} \to \tilde{6}$ present in (c) has no representative in (a). For the constraints derived from these graphs this can imply that a formulation derived from (c) which has less variables and less constraints is a relaxation of the one derived from (a). For example flow conservation constraints for (a), with node 2 as a source and node 6 as a sink, are given by

$$\begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 & 0 & 0 \\ 1 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & -1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{(1,3)} \\ x_{(2,3)} \\ x_{(2,4)} \\ x_{(3,5)} \\ x_{(4,5)} \\ x_{(4,6)} \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{1}$$

for (b), respectively, by

$$\begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 & 0 & 0 \\ 1 & 1 & 1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{(\tilde{1},\tilde{3})} \\ x_{(\tilde{2},\tilde{3})_1} \\ x_{(\tilde{2},\tilde{3})_2} \\ x_{(\tilde{3},\tilde{5})_1} \\ x_{(\tilde{3},\tilde{5})_2} \\ x_{(\tilde{4},\tilde{6})} \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{2}$$

and for (c) by

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 1 & -1 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{(\tilde{1},\tilde{3})} \\ x_{(\tilde{2},\tilde{3})} \\ x_{(\tilde{3},\tilde{5})} \\ x_{(\tilde{3},\tilde{6})} \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{3}$$

Notice that (2) is just (1) with two aggregated rows (which can be expressed by multiplying its system matrix with some matrix from the left), and (3) is just (2) after deleting duplicate columns (which can be expressed by multiplying its system matrix by some matrix from the right). Vice versa, it is also possible to obtain (2) from (3) by duplicating the respective columns. So, in this example we can obtain a relaxed MILP formulation with less rows and less columns by multiplying the system matrix with suitable matrices. However, it is not possible to delete columns arbitrarily: If we deleted the column corresponding to the arc $(4, 6)$ instead of the duplicate ones, any path containing that arc would also be deleted, and the corresponding MILP formulation cannot be used as a relaxation. For our concept of aggregation, we therefore make use of the intermediate formulation (2) of (b), which we can obtain from (1) by aggregating two rows and also by disaggregating columns of (3). Based on this concept we present two algorithms, one very similar to an approach from the literature and one new branch-and-bound type algorithm, that allows for such a graph refinement during the exploration of the branch-and-bound tree. To test these algorithms we customize them for two suitable problems from the literature, the shortest path problem with time-windows (SPPTW) for the LP case and the traveling salesman problem with time-windows (TSPTW) for the MILP case.

2 Literature Review

In this work, we do not consider integer programming formulations that include time explicitly as continuous variables, but so-called time-indexed formulations. These can, for example, be derived from time-expanded graphs and are in many cases known to have better LP-relaxations than those formulations incorporating time as additional continuous variables (see for example Wang and Regan [28] or Dash et al. [9]). Many examples of time-indexed MILP formulations can be found in the literature, an early example has been given by Appelgren [1]. An early example for the use of time-expanded graphs in an MILP formulation is the work of Levin [19]. As mentioned before, one of the problems of using

these formulations is that the time-expanded graph has much more nodes and arcs than the original graph resulting in very large MILP formulations. Large MILP formulations are not necessarily only theoretically relevant. Many branch-and-cut algorithms, see Nemhauser and Wolsey [22], deal with a huge number of rows by including subroutines that separate violated constraints and add these to the LP relaxation 'on demand'. Column generation algorithms for MILP such as branch-and-price, are often used to solve problems, where the number of variables in the formulation scales exponentially with some input parameters, hence they include a subroutine to find those variables that have negative reduced cost (see Barnhart et al. [4] for an in depth explanation).

In the case of graph refinement algorithms, it is not the variables or constraints which are added on-demand, but the nodes and arcs of the time-expanded graph from which the MILP is derived. If the solution of the current graph is found to be time-infeasible and therefore additional nodes are required, they are added and the MILP formulation is adjusted. In contrast to branch-and-cut algorithms, in which constraints can be added during the exploration of the branch-and-bound tree, the graph refinement is performed in an outer loop. Boland et al. [7] suggest to make use of two different graphs, that are refined based on infeasible solutions. The first is underestimating traveling times and can, therefore, be used to find lower bounds and the second is overestimating traveling times and therefore defines upper bounds. If the optimal solution of the current MILP violates a time-window, the graph is refined and the MILP is adjusted to the new graph. These steps are repeated until either no time-windows are violated and an optimal solution is found, or the objective of the current MILP is equal to the upper bound found by the formulation from the graph overestimating traveling times, proving its optimality. Recently, this approach has been adapted successfully to a variety of problems, see He et al. [17], Vu et al. [27], Boland et al. [6], and Lagos et al. [18]. Riedler et al. [25] propose a similar algorithm, the iterative refinement algorithm, in which additionally nodes are added based on the current LP relaxation. Furthermore, they compare a variety of refinement strategies. Gnegel and Fügenschuh [16] propose another type of refinement algorithm for an airplane scheduling and routing problem. The difference to the previously mentioned approaches is that time is included as variables and the graph is expanded based on the number of times an airport is visited in the solution of the relaxation. We adopt the terminology 'iterative refinement' from Riedler et al. [25], instead of 'dynamic discretization discovery' from He et al. [17], because our refinement algorithms are not restricted to finding discretizations of time-windows and can be applied to more general MILPs.

In this work, in addition to a general iterative refinement algorithm fitting the previously mentioned examples, we suggest a new kind of refinement algorithm, which we call branch-and-refine. Similar to branch-and-cut algorithms, which add cuts during the exploration of the branch-and-bound tree, branch-and-refine includes the graph refinement as an additional step in a branch-and-bound algorithm. To the best of our knowledge this kind of concept has not been explored in the literature before.

The first problem we use to encourage the use of our algorithms is the SPPTW, a time-dependent variant of the SPP. While algorithms for the SPP with polynomial runtime in the number of vertices are well-known, e.g., Dijkstra's algorithm [10], if the arc weights are non-negative, or the Bellmann-Ford algorithm [13, 5], if there are no negative cycles, no such algorithm exists for the general SPPTW unless P equals NP [11]. In order to solve the SPPTW, we use the flow-based SPP formulation of Wolsey [29] on the time-expanded graph. Because strong duality holds for this formulation, it suffices to solve an LP instead of an MILP.

The second problem we consider in this work is the TSPTW. The TSP without additional constraints is already an NP hard problem and was used by Dantzig [8] to demonstrate the relevance of the simplex algorithm for the solution of combinatorial problems. It is one of

the classical examples of integer programming problems. In the literature, many different solution approaches for the TSPTW can be found. Exact algorithms are for example the branch-and-cut algorithms of Ascheuer et al. [2] or Dash et al. [9], and the constraint logic programming based approach by Pesant et al. [24]. The best-known feasible solutions on very large instances, however, are often found by heuristics. Examples of well-performing heuristics are the hybrid algorithm of López-Ibáñez and Blum [20] combining techniques of ant colony algorithms and beam search, the simulated annealing approach by Ohlmann and Thomas [23], and the insertion heuristic of Gendreau et al. [15].

3 Constraint and Variable Aggregations of Mixed-Integer Linear Programs

We start by introducing some general terms and notions, which allow us to formally express the variable and constraint aggregations outlined in the introduction.

For the set of non-negative rational numbers, we use $\mathbb{Q}_{\geq}$, and for the set of natural numbers including 0 we use $\mathbb{N}$. Then, for given $n \in \mathbb{N}$ we use $\mathbf{I}_n \in \mathbb{Q}^{n\times n}$ for the $n \times n$-identity matrix and $\mathbf{e}_i^n \in \mathbb{Q}^n$ for the i-th unit column vector.

For the theoretical results, we use the following standardized form of MILPs, making use of inequality constraints only. In later sections, however, we will introduce models that include equality constraints. For these models, we implicitly assume that they are first transformed into the standard form so that the theoretical results can be applied.

Definition 1. *An MILP (in standard form) M with n variables, of which p are continuous, and m constraints is an optimization problem given by*

$$\begin{aligned} \min_{x\in\mathbb{Q}_{\geq}^{p}\times\mathbb{N}^{n-p}} \quad & c^\top x \\ s.t. \quad & Ax \geq b, \end{aligned} \tag{M}$$

where $A \in \mathbb{Q}^{m\times n}, c \in \mathbb{Q}^n, b \in \mathbb{Q}^m$. If $p = n$ then M is called a linear program (LP). The MILP obtained by replacing p with n in the optimization problem above is called the linear relaxation of M and is denoted by L. If M is an LP, then we call the optimization problem given by

$$\begin{aligned} \max_{y\in\mathbb{Q}_{\geq}^{m}} \quad & b^\top y \\ s.t. \quad & A^\top y \leq c, \end{aligned} \tag{L^{d}}$$

the dual linear program.

Whenever we introduce a general MILP M_j^i with subscript i and superscript j, we use $A_j^i, b_j^i, c_j^i, m_j^i, n_j^i, p_j^i$ in the same way as A, b, c, m, n, p are used in Definition 1 for M without explicitly introducing them[1].

Definition 2. *Given an MILP M and a point $x \in \mathbb{Q}_{\geq}^p \times \mathbb{N}^{n-p}$, we say x is feasible if $Ax \geq b$ and call $c^\top x$ objective value of x. The set of all feasible x for an MILP M is denoted by* $\mathrm{feas}(M)$.

We call $\mathrm{obj}(M) := \inf\{c^\top x \mid x \in \mathrm{feas}(M)\}$ *the optimal value of M and say x is an optimal solution if $c^\top x = \mathrm{opt}(M)$. The set of all optimal solutions is denoted by* $\mathrm{opt}(M)$.

Given a matrix $A \in \mathbb{Q}^{m\times n}$, we write $A(\cdot) \in \mathcal{L}(\mathbb{Q}^n, \mathbb{Q}^m)$ for the respective linear mapping. For $B \subseteq \mathbb{Q}^n$, we denote the image of B under the mapping $A(\cdot)$ by $A(B)$.

With these notations, we now give the definition of a relation between MILPs that is essential for the remainder of this work.

[1]Here, i and j do not necessarily have to be numbers or letters, and might be missing entirely.

Definition 3. *Let M_1 and M_2 be two MILPs such that $m_2 \leq m_1$ and $n_2 \leq n_1$. Further let $C \in \mathbb{Q}_{\geq}^{m_2 \times m_1}$ with $\operatorname{rank}(C) = m_2$ and $V \in \mathbb{Q}_{\geq}^{n_2 \times n_1}$ with $\operatorname{rank}(V) = n_2$ be two matrices. Additionally, let V have a block structure*

$$V = \begin{pmatrix} V' & 0 \\ 0 & V'' \end{pmatrix}$$

with $V' \in \mathbb{Q}^{p_2 \times p_1}$ and $V'' \in \mathbb{N}^{n_2 - p_2 \times n_1 - p_1}$.

M_2 is called a (C, V)-aggregation (or simply aggregation) of M_1 if

$$CA_1 = A_2V,\ Cb_1 = b_2,\ c_1^\top = c_2^\top V. \tag{4}$$

An aggregation is called proper if either $n_2 < n_1$ or $m_2 < m_1$. If M_2 is a (proper) aggregation of M_1, then, conversely, M_1 is called a (proper) refinement of M_2. Furthermore, if M_2 is a (C, V)-aggregation of M_1 we call $y \in \operatorname{feas}(M_2)$ expandable if there exists $x \in \operatorname{feas}(M_1)$ with $Vx = y$.

With this definition, we can now formally introduce the variable and constraint aggregations mentioned in the introduction in general terms.

Example 1. *1. Let M_1 be an MILP, $i, j \in \{1, \ldots, m\}$ with $i < j$, and $\alpha \in \mathbb{Q}_{\geq}$. With $V = \mathbf{I}_n$ and*

$$C = \begin{pmatrix} \mathbf{e}_1^m, & \ldots, & \mathbf{e}_{i-1}^m, & \mathbf{e}_i^m + \alpha \mathbf{e}_j, & \mathbf{e}_{i+1}^m, & \ldots, & \mathbf{e}_{j-1}^m, & \mathbf{e}_{j+1}^m, & \ldots, & \mathbf{e}_m^m \end{pmatrix}, \tag{5}$$

we obtain an aggregation M_2 of M_1, with $A^* = CA$, $b^* = Cb$, $c^* = c$, $n^* = n$, $p^* = p$ and $m^* = m - 1$. We call M_2 a constraint aggregation of M_1 and additionally the special case $\alpha = 0$ a constraint removal of M.

2. Let M_2 be an MILP, $i, j \in \{p+1, \ldots, n\}$ and $\alpha \in \mathbb{N}_{\geq}$. With $C = \mathbf{I}_m$ and

$$V = \begin{pmatrix} \mathbf{e}_1^n, & \ldots, & \mathbf{e}_{j-1}^n, & \alpha \mathbf{e}_i^n, & \mathbf{e}_j^n, & \ldots, & \mathbf{e}_n^n \end{pmatrix}, \tag{6}$$

we can obtain a refinement M_1 of M_2 with $A^* = AV$, $b^* = b$, $c^* = V^\top c$, $p^* = p$, $n^* = n+1$ and $m^* = m$. We call M_1 (and also the analogous case for two continuous variables, i.e., $i, j \in \{1, \ldots, p\}$ and $p^* = p+1$) a variable refinement of M_2, and M_2 a variable aggregation of M_1.

3. The previous examples can be generalized to MILPs obtained by aggregating multiple subsets of constraints or variables.

4. Consider the MILPs

$$\begin{aligned} \min_{x \in \mathbb{N}^2} \quad & x_1 + x_2 \\ s.t. \quad & x_1 \geq 1, \\ & x_2 \geq 1 \end{aligned} \tag{M_1}$$

and

$$\begin{aligned} \min_{y \in \mathbb{N}^2} \quad & y_1 + y_2 \\ & y_2 \geq 1, \end{aligned} \tag{M_2}$$

where the second can be obtained from the first by removing the first constraint, so it is an aggregation with $V = \mathbf{I}_2$ and $C = \begin{pmatrix} 1 & 0 \end{pmatrix}$. In this case $y = \begin{pmatrix} 0 & 1 \end{pmatrix}$ is not expandable, but $y = \begin{pmatrix} 1 & 1 \end{pmatrix}$ is.

Although the concept of aggregations is much more general, the algorithms of the later sections are only making use of those in Example 1 with $\alpha \in \{0, 1\}$. It would be interesting to identify other, maybe not even graph based problems, where other types of aggregations appear.

The following lemma indicates, that there are several properties of MILPs and their solutions which can be derived from their aggregations. It also illustrates that aggregating an MILP can be useful for solving it.

Lemma 1. *Let M_1 and M_2 be MILPs, and $V \in \mathbb{Q}_{\geq}^{n_2 \times n_1}$ and $C \in \mathbb{Q}_{\geq}^{m_2 \times m_1}$ be matrices such that M_2 is a (C, V)-aggregation of M_1. Then:*

(i) *L_2 is a (C, V)-aggregation of L_1,*

(ii) *$V(\text{feas}(M_1)) \subseteq \text{feas}(M_2)$,*

(iii) *if $\text{feas}(M_2) = \emptyset$, then $\text{feas}(M_1) = \emptyset$,*

(iv) *$\text{opt}(M_1) = \inf\{c_2^\top y | y \in V(\text{feas}(M_1))\}$,*

(v) *$\text{opt}(M_2) \leq \text{opt}(M_1)$,*

(vi) *if $x \in \text{feas}(M_1)$ with $Vx \in \text{opt}(M_2)$, then $x \in \text{opt}(M_1)$.*

Proof.

(i) Replace p_1 by n_1 and p_2 by n_2 in Definition 3.

(ii) Let $x \in \text{feas}(L_1)$, then $A_1 x \geq b_1$ and hence, $CA_1 x \geq Cb_1$. By substituting property (4) of the definition of (C, V)-aggregation, this is equivalent to $A_2 Vx \geq b_2$. Furthermore, the block structure of V guarantees that Vx fulfills the integrality constraints of M_2, if x fulfills the integrality constraints of M_1. So, we can conclude $Vx \in \text{feas}(M_2)$.

(iii) This is a direct consequence of (ii).

(iv) Since $c_1 = V^\top c_2$, this statements can be shown by the change of variables:

$$\text{opt}(M_1) = \inf_{x \in \text{feas}(M_1)} c_1^\top x = \inf_{x \in \text{feas}(M_1)} c_2^\top Vx = \inf_{y \in V(\text{feas}(M_1))} c_2^\top y.$$

(v) Note that by (ii) holds

$$\inf_{y \in \text{feas}(M_2)} c_2^\top y \leq \inf_{x \in \text{feas}(M_1)} c_2^\top Vx$$

and by assumption, also holds $c_2^\top V = c_1^\top$. So, we can conclude that

$$\text{opt}(M_2) = \inf_{y \in \text{feas}(M_2)} c_2^\top y \leq \inf_{x \in \text{feas}(M_1)} c_2^\top Vx = \inf_{x \in \text{feas}(M_1)} c_1^\top x = \text{opt}(M_1).$$

(vi) Let $x \in \text{feas}(M_1)$ and $Vx \in \text{opt}(M_2)$, then

$$\text{opt}(M_2) = c_2^\top Vx = c_1^\top x \geq \inf_{x' \in \text{feas}(M_1)} c_1^\top x' = \text{opt}(M_1).$$

With the estimation in property (v), we obtain $\text{opt}(M_1) = \text{opt}(M_2)$ and, therefore, $x \in \text{opt}(M_1)$.

□

From a practical point of view, the results of Lemma 1 are especially relevant, if M_2 has much less constraints or variables than M_1. In this case, solving M_2 can sometimes take only a fraction of the time it takes to solve M_1. If we determine that $\text{feas}(M_2) = \emptyset$, then we can directly conclude that also $\text{feas}(M_1) = \emptyset$, without investing the time necessary to solve M_1. Furthermore, if we find a solution $y \in \text{opt}(M_2)$, by (v) the objective value of y gives a proven lower bound for the optimal value of M_1. This can be very effective in combination with a heuristic that finds upper bounds for M_1 (in practise these heuristics usually try to find feasible points for M_1, whose objective then defines an upper bound on $\text{opt}(M_1)$). If, in addition we have an effective algorithm to check whether there exists an $x \in \text{feas}(M_1)$ such that $Vx = y$ (i.e., y is expandable), we can apply (vi) to conclude that x is a proven optimal solution of M_1.

The following result shows, that aggregation is also a transitive relation.

Proposition 1. *Let M_1, M_2 and M_3 be MILPs, and $V_1 \in \mathbb{Q}_{\geq}^{n_2 \times n_1}$, $C_1 \in \mathbb{Q}_{\geq}^{m_2 \times m_1}$, $V_2 \in \mathbb{Q}_{\geq}^{n_3 \times n_2}$ and $C_2 \in \mathbb{Q}_{\geq}^{m_3 \times m_2}$ be matrices such that M_3 is a (C_2, V_2)-aggregation of M_2 and M_2 is a (C_1, V_1)-aggregation of M_1. Then M_3 is a (C_2C_1, V_2V_1)-aggregation of M_1.*

Proof. The assumptions imply that $C_1A_1 = A_2V_1$ and $C_2A_2 = A_3V_2$. The second equality implies that $C_2A_2V_1 = A_3V_2V_1$, and a substitution of the first yields $C_2C_1A_1 = A_3V_2V_1$. Similarly, we get $c_1^\top = c_2^\top V_1 = c_3^\top V_2V_1$ and $b_3 = C_2b_2 = C_2C_1b_1$. Furthermore, it holds $V_1(\mathbb{Q}^{p_1} \times \mathbb{N}^{n_1-p_1}) \subseteq \mathbb{Q}^{p_2} \times \mathbb{N}^{n_2-p_2}$ and $V_2(\mathbb{Q}^{p_2} \times \mathbb{N}^{n_2-p_2}) \subseteq \mathbb{Q}^{p_3} \times \mathbb{N}^{n_3-p_3}$, and hence $V_2(V_1(\mathbb{Q}^{p_2} \times \mathbb{N}^{n_2-p_2})) \subseteq \mathbb{Q}^{p_3} \times \mathbb{N}^{n_3-p_3}$. The multiplication of two matrices with non-negative entries always results in a matrix with non-negative entries and $\text{rank}(V_2V_1) = n_3$ and $\text{rank}(C_2C_1) = m_3$ (by Sylvester's rank inequality or Sylvester's law of nullity, see for example [21]), so we can conclude that M_3 is a (C_2C_1, V_2V_1)-aggregation of M_1. □

The next result shows that we can preserve the aggregation relation between two MILPs when new constraints are added.

Lemma 2. *Let M_1 and M_2 be MILPs, and $V \in \mathbb{Q}_{\geq}^{n_2 \times n_1}$ and $C \in \mathbb{Q}_{\geq}^{m_2 \times m_1}$ be matrices such that M_2 is a (C, V)-aggregation of M_1. Additionally let $l \in \mathbb{N}$, $D \in \mathbb{Q}^{l \times n_2}$, $d \in \mathbb{Q}^l$, and M_1^*, M_2^* be the MILPs obtained by setting*

$$A_1^* = \begin{pmatrix} A_1 \\ DV \end{pmatrix}, b_1^* = \begin{pmatrix} b_1 \\ d \end{pmatrix}, c_1^* = c_1, n_1^* = n_1, m_1^* = m_1 + l, p_1^* = p_1$$

and

$$A_2^* = \begin{pmatrix} A_2 \\ D \end{pmatrix}, b_2^* = \begin{pmatrix} b_2 \\ d \end{pmatrix}, c_2^* = c_2, n_2^* = n_2, m_2^* = m_2 + l, p_2^* = p_2$$

then

(i) M_2^* is a (C^*, V)-aggregation of M_1^*, where $C^* = \begin{pmatrix} C & 0 \\ 0 & \mathbf{I}_l \end{pmatrix}$.

(ii) For all $x \in \text{feas}(M_1)$ with $Vx \in \text{feas}(M_2^*)$ it holds that $x \in \text{feas}(M_1^*)$.

Proof.

(i) We check all conditions from Definition 3. The objective functions did not change, so ${c_1^*}^\top = {c_2^*}^\top V$. Furthermore,

$$C^*A_1^* = \begin{pmatrix} CA_1 \\ DV \end{pmatrix} = \begin{pmatrix} A_2V \\ DV \end{pmatrix} = A_2^*V$$

and

$$C^*b_1^* = \begin{pmatrix} Cb_1 \\ d \end{pmatrix} = \begin{pmatrix} b_2 \\ d \end{pmatrix} = b_2^*.$$

If C has only non-negative entries and full row rank so does C^*.

(ii) If $x \in \text{feas}(M_1)$, then $A_1 x \geq b$ and if $Vx \in \text{feas}(M_2^*)$, then $DVx \geq d$ and therefore also $x \in \text{feas}(M_1^*)$.

□

Adding constraints to an MILP is for example done in the branching step of branch-and-bound algorithms or when adding cuts in branch-and-cut algorithms. Now, if we only know a refinement of the original MILP M_1, but need to find a refinement of the MILP with additional inequalities, it is possible to apply Lemma 2 to find it.

The last result on aggregations we present in this section is about finding feasible points of the dual LP by solving an aggregation.

Proposition 2. *Let L_1 and L_2 be LPs, and $V \in \mathbb{Q}_{\geq}^{n_2 \times n_1}$ and $C \in \mathbb{Q}_{\geq}^{m_2 \times m_1}$ be matrices such that L_2 is a (C,V)-aggregation of L_1. For $y \in \text{feas}(L_2^{\text{d}})$ holds $C^\top y \in \text{feas}(L_1^{\text{d}})$.*

Proof. Let $y \in \text{feas}(L_2^{\text{d}})$ then by definition $A_2^\top y \leq c_2$. Multiplication by $V^\top$ from the left, which has only non-negative entries, yields $(A_2 V)^\top y \leq V^\top c_2$. Substituting equation (4) from Definition 3 implies $A_1^\top C^\top y \leq c_1$, and since $C^\top y \geq 0$ we get $C^\top y \in \text{feas}(L_1^{\text{d}})$. □

In the computational experiments we will make use of this result to speed up the algorithms. The basic idea is: if the linear relaxation of an MILP has already been solved, then a solution of the dual LP (which can be easily computed by the revised simplex or dual simplex algorithm) can be easily transformed to obtain a feasible starting point for solving the linear relaxation of its refinements by the dual simplex algorithm.

4 Refinement Algorithms

In the previous section we introduced the concept of aggregations and refinements. In particular, the results in Lemma 1 show that it is possible to obtain important insights about an MILP by solving some aggregations of it. A closer look, however, also reveals that there is no guarantee that solving an aggregation is actually useful.

Assume, for example, that we want to solve an MILP M for which holds $\text{feas}(M) = \emptyset$. Now, if we solve an aggregation M_{agg} of M, there are two possibilities: If $\text{feas}(M_{\text{agg}}) = \emptyset$, then by Lemma 1 (iii), we can conclude correctly that $\text{feas}(M) = \emptyset$, but if $\text{feas}(M_{\text{agg}}) \neq \emptyset$, there are no implications for M. A similar situation also appears if M has a solution. If we find a solution of M_{agg} and we can show that it is expandable, then by Lemma 1 (vi) we can calculate $\text{opt}(M)$. However, if it is not expandable (or we cannot show that it is), solving M_{agg} is of little use. In addition, the lower bound we obtain by applying Lemma 1 (v) does not have to be tight.

Finding aggregations of an MILP that are easier to solve is a trivial task, but finding those that are actually helpful for finding proven optimal solutions is not. In this section, we propose two algorithms for finding aggregations with the desired properties. For ease of notation we assume, that all MILPs used in the input of the algorithms have an optimal value strictly greater than $-\infty$.

4.1 General Iterative Refinement Algorithm

The first algorithm designed to find useful aggregations is outlined in Algorithm 1. We call it the General Iterative Refinement Algorithm. For the input of Algorithm 1 one should chose MILPs M, M_{agg} such that it is much easier to solve M_{agg} than M. Then, within a loop, M_{agg} is replaced by another MILP, that is both, an aggregation of M and a proper refinement of M_{agg}. The loop is exited, when the current M_{agg} has no solution (line 2)

Algorithm 1: General Iterative Refinement Algorithm

```
1 Input: An MILP M and an aggregation M_agg of M.
2 while feas(M_agg) ≠ ∅ do
3     Find y ∈ opt(M_agg);
4     if y is expandable then
5         return c_agg^⊤ y;
6     else
7         (Refine:) Find a proper refinement M_new of M_agg, such that M_new is an
           aggregation of M;
8         M_agg ← M_new;
9 return +∞;
```

or an expandable solution (line 5). Note that, because a proper refinement is required in line 7, with each iteration M_{agg} becomes more and more likely to have an expandable solution or no solution (if M has no solution). Although difficult in principle, in specific applications checking whether y is expandable in line 4 can be easy. In the applications of the later sections this is the case and because of this, it is not even necessary to explicitly store M in the memory.

The following theorem shows that Algorithm 1 always terminates and returns the objective value of M.

Theorem 1. *Algorithm 1 terminates after finitely many steps. The return value is* $\text{opt}(M)$.

Proof. In each loop iteration of Algorithm 1, M_{agg} is always replaced by a proper refinement which implies that the algorithm either terminates in line 9 or at some point M_{agg} has to have the same number of constraints and variables as M. Now, if M_{agg} and M have the same number of constraints and variables and M_{agg} is a (C, V)-aggregation of M, then V and C are square matrices with full rank and hence invertible matrices. For the solution y found in line 3, this implies $V^{-1}y \in \text{feas}(M)$. Therefore y is expandable and the algorithm terminates in line 5. So, a situation, where it is not possible anymore to find a proper refinement or the algorithm cycles cannot occur. We conclude that Algorithm 1 always terminates after a finite number of steps.

Now, if Algorithm 1 terminates in line 9, it holds $\text{feas}(M_{\text{agg}}) = \emptyset$ for an aggregation M_{agg} of M. By Lemma 1 (iii), it holds $\text{feas}(M) = \emptyset$. This implies $\text{opt}(M) = +\infty$, the return value. Finally, if Algorithm 1 terminates in line 5, y is expandable and there exists $x \in \text{feas}(M)$ such that $y = Vx \in \text{opt}(M_{\text{agg}})$ for an aggregation M_{agg} of M. By Lemma 1 (vi) then holds $x \in \text{opt}(M)$. Therefore, $\text{opt}(M) = c^\top x = c_{\text{agg}}^\top y$. Hence, the return value is also correct in this case. □

4.2 Branch-and-refine

The second algorithm follows the branch-and-bound paradigm. Branch-and-bound algorithms for MILPs are based on repetitively executing a branching step, where the set of feasible points is divided into two (or possibly more) subsets, and a bounding step, where a lower bound valid for all elements of a given subset is computed. This lower bound is usually found by relaxing the constraints in some way. In the case of MILPs, the relaxed problem can be obtained by discarding the integer condition and solving the LP relaxation instead (see for example Fügenschuh and Martin [14] for a more formal description). Considering the property stated in Lemma 1 (i), it is also possible to use the LP-relaxation of an aggregation.

Following this idea, we propose another refinement algorithm for solving MILPs. The details are given in Algorithm 2, which we call branch-and-refine. Although not obvious at first glance, branch-and-refine is structurally very similar to the General Iterative Refinement Algorithm. Solving M_{agg} in line 3 of Algorithm 1 is usually done by a branch-and-bound algorithm. Branch-and-refine incorporates the refinement step of Algorithm 1 (line 7) into a branch-and-bound algorithm, and therefore makes the loop of Algorithm 1 unnecessary. Before presenting results for Algorithm 2, we remark that if $M_{\text{agg}} = M$

Algorithm 2: Branch-and-refine

```
1  Input: Two MILPs M and M_agg, such that M_agg is an aggregation of M.
2  L_O ← {M_agg}, L_C ← ∅, U ← +∞;
3  while L_O ≠ ∅ do
4      Choose M* ∈ L_O;
5      if feas(L*) = ∅ or opt(L*) ≥ U then
6          Move M* from L_O to L_C;
7      else
8          Find y ∈ opt(L*);
9          if y ∈ feas(M_agg) then
10             if y is expandable then
11                 U ← c_agg^⊤ y;
12                 Move M* from L_O to L_C;
13             else
14                 (Refine:) Find a proper refinement M_new of M_agg, such that M_new
                    is an aggregation of M;
15                 Set M_agg to M_new and transform each problem M' ∈ L_O in
                    accordance with Lemma 2 into a problem M'_new;
16         else
17             (Branch:) Add additional constraints to M* to obtain MILPs M_1* and M_2*
               such that feas(M*) = feas(M_1*) ∪ feas(M_2*). Add M_1*, M_2* to L_O and move
               M* from L_O to L_C;
18 return U;
```

intially then y is always expandable in line 10 and the block in line 14-15 can never be reached. So, in this case Algorithm 2 describes just a usual branch-and-bound algorithm using the LP-relaxation for bounding. We further remark, that the MILPs M_1^* and M_2^* which are added to the list of open problems $\mathcal{L}_O$ in line 17 are obtained by imposing additional constraints to the MILP M^* and we can, therefore, use the formula in Lemma 2 to obtain the problems M_{new}^* in line 15.

The following results show that Algorithm 2 can be used to solve MILPs. For ease of notation, given $K \in \mathbb{Q} \cup \{+\infty\}$ we denote by

$$\text{feas}_K(M) := \{x \in \text{feas}(M) | c^\top x < K\},$$

the set of all feasible points of an MILP M with objective value that is strictly less than K.

Lemma 3. *For any matrices C, V such that M_{agg} is a (C, V)-aggregation of M, it holds that*

$$V(\text{feas}_U(M)) \subseteq \bigcup_{M' \in \mathcal{L}_O} \text{feas}(M') \tag{7}$$

during the execution of Algorithm 2.

Proof. If M_{agg} is a (C, V)-aggregation of M, then by Lemma 1 (ii) $V(\text{feas}(M)) \subseteq \text{feas}(M_{\text{agg}})$ and the assertion holds in line 2, where $\mathcal{L}_O = \{M_{\text{agg}}\}$ and $U = +\infty$.

Furthermore, the elements of $\mathcal{L}_O$ that are removed, are either added to $\mathcal{L}_C$ or, in line 17, divided in such a way that the union of their feasible sets is the original feasible set. Therefore, we can conclude, that if (7) holds at some point then until the 'Refine' step, i.e., line 14-15, is reached, it holds that

$$V(\text{feas}_U(M)) \subseteq \text{feas}(M_{\text{agg}}) = \bigcup_{M' \in (\mathcal{L}_O \cup \mathcal{L}_C)} \text{feas}(M').$$

Now, assume there exists $x \in \text{feas}_U(M)$ and $M' \in \mathcal{L}_C$ such that $Vx \in \text{feas}(M')$. Consequently, $\text{feas}(M') \neq \emptyset$ and, hence, $\text{opt}(M') \geq U$ (otherwise it would not have been added to $\mathcal{L}_C$ in line 5 or line 12). Now, by Lemma 1 (v) it holds that $c^\top x \geq \text{opt}(M) \geq \text{opt}(M') \geq U$, a contradiction to $x \in \text{feas}_U(M)$. We conclude that until we reach the 'Refine' step (7) holds true.

Now, assume line 15 is reached. Let $\mathcal{L}_O^{\text{new}}$ be the transformed list, and C_1, V_1, C_2, V_2 be matrices such that M_{agg} is a (C_1, V_1)-aggregation of M_{new} and M_{new} is a (C_2, V_2)-aggregation of M. By Proposition 1, M_{agg} is a (C_1C_2, V_1V_2)-aggregation of M and, therefore, (7) holds for $V = V_1V_2$, i.e.,

$$V_1(V_2(\text{feas}_U(M))) \subseteq \bigcup_{M' \in \mathcal{L}_O} \text{feas}(M').$$

This implies that for all $x \in V_2(\text{feas}_U(M))$ exists $M' \in \mathcal{L}_O$ such that $V_1 x \in \text{feas}(M')$. This M' is just M_{agg} with additional inequalities of type $Dx \geq d$ (as required in Lemma 2) which were added in previous calls of the 'Branch' step. Therefore, we can apply Lemma 2 (ii) with $V = V_1$, $C = C_1$, $M_1 = M_{\text{new}}$, $M_2 = M_{\text{agg}}$, $M_1^* = M'_{\text{new}}$, $M_2^* = M'$, to show $x \in \text{feas}(M'_{\text{new}})$. It follows that

$$V_2(\text{feas}_U(M)) \subseteq \bigcup_{M'_{\text{new}} \in \mathcal{L}_O^{\text{new}}} \text{feas}(M'_{\text{new}}),$$

which is (7) for the transformed list and new aggregation matrix. In summary, we have shown that (7) holds initially, that while the 'Refine' step is not reached it remains valid, and that after the 'Refine' step is performed (7) holds for the transformed list and any new aggregation matrix. So, we can conclude, that (7) holds true throughout the execution of the algorithm. □

Using this result, we can now prove the following result for Algorithm 2.

Theorem 2. *If Algorithm 2 terminates, the return value is* $\text{opt}(M)$.

Proof. If $\text{feas}(M) = \emptyset$, then y can never be expandable in line 10, therefore U is never changed throughout the algorithm. So, if the algorithm terminates, it has to terminate with $U = +\infty = \text{opt}(M)$.

If $\text{feas}(M) \neq \emptyset$, upon termination holds $\mathcal{L}_O = \emptyset$, and we can apply Lemma 3 to obtain

$$V(\text{feas}_U(M)) \subseteq \bigcup_{M^* \in \mathcal{L}_O} \text{feas}(M^*) = \emptyset.$$

Because we assumed $\text{feas}(M) \neq \emptyset$, it follows that $V(\text{feas}(M)) \neq \emptyset$. So, it has to hold that $\text{opt}(M) \geq U$, which also implies $U \neq +\infty$. Therefore, U has changed during the execution of the algorithm and is equal to the objective of an expandable y. For any y that is expandable exists $x \in \text{feas}(M)$, with the same objective as y. Therefore, also holds that $\text{opt}(M) \leq U$ which concludes the proof. □

Note that in Theorem 2 we do not conjecture, that Algorithm 2 terminates which is a major difference to the statement in Theorem 1. This is because the branching rule is too general to guarantee that it will only branch finitely many times.

To the best of our knowledge, there is no example of an algorithm in the literature that matches Algorithm 2, but in principle it is possible to formulate Algorithm 2 as a classical branch-and-bound algorithm, where the relaxations used for finding lower bounds (i.e., the linear relaxation of different aggregations) are changed during its execution. We further note the similarities to branch-and-cut algorithms that initially leave out a subset of the constraints and then add some of them on demand in form of cutting planes during the exploration of the branch-and-bound tree. Discarding constraints as pointed out before is a special type of aggregation and, therefore, if constraints are only added when the solution of the LP-relaxation of the chosen node is integer, these algorithms fit the framework of branch-and-refine. Although we do not give the details, it should be apparent that Algorithm 2 can be adapted in such a way that the transformation step can also be reached if y has fractional values. Then, most branch-and-cut algorithms would fit into the framework of the so-adapted branch-and-refine.

5 Applications of the Refinement Algorithms

In this section we present two graph-based problems and show that they can be solved with the refinement-based algorithms of the previous section. Throughout this section we assume that $G = (N, A)$ is a directed graph with nodes in N and arcs in A. Furthermore, we assume that each node $i \in N$ has a time-window $I_i = \{e_i, \dots, l_i\}$ associated with it, and each arc $(i, j) \in A$ has parameters $c_{i,j} \in \mathbb{Q}_>$ and $d_{i,j} \in \mathbb{N}$, the costs and traveling times, respectively. In the notations of the expansions of G considered in this work, we make use of the concepts introduced in the following definition.

Definition 4. *Let $I = \{l, \dots, m\}$ with $l, m \in \mathbb{N}$, $l \leq m$, and $I^i \subseteq I, i \in \{1 \dots, k\}$ for some $k \in \mathbb{N}$. The set $P = \{I^1, \dots, I^k\}$ is called an ordered partition (partition) of I if $I = \bigcup_{i=1}^{k} I^i$ and $\max(I^i) + 1 = \min(I^{i+1})$ for all $i = 1, \dots, k-1$.*

Definition 5. *If $P_i = \{I_i^1, \dots, I_i^{k_i}\}$ with $k_i \in \mathbb{N}$ are partitions of the time-windows I_i of the nodes $i \in N$, then $\mathcal{P} = \{P_i \mid i \in N\}$ is called a time-window partition. The time-window partition $\hat{\mathcal{P}} := \{\{\{e_i\}, \{e_i + 1\}, \dots, \{l_i\}\} \mid i \in N\}$ is called the complete time-window partition.*

Now, for a time-window partition $\mathcal{P}$, we introduce the following notations for the expanded sets of nodes and arcs:

$$
\begin{aligned}
&\mathcal{N}_i^{\mathcal{P}} := \{i\} \times P_i, && \forall\, i \in N,\\
&\mathcal{A}_{i,j}^{\mathcal{P}} := \{(i, I, j, J) \in \mathcal{N}_i^{\mathcal{P}} \times \mathcal{N}_j^{\mathcal{P}} \mid \min(I) + d_{i,j} \leq \max(J)\}, && \forall\, (i, j) \in A,\\
&\mathcal{N}^{\mathcal{P}} := \bigcup_{i \in N} \mathcal{N}_i^{\mathcal{P}},\\
&\mathcal{A}^{\mathcal{P}} := \bigcup_{(i,j) \in A} \mathcal{A}_{i,j}^{\mathcal{P}},\\
&\delta_+^{\mathcal{P}}(i, I) := \{(i, I, j, J) \in \mathcal{A}^{\mathcal{P}}\}, && \forall\, (i, I) \in \mathcal{N}^{\mathcal{P}},\\
&\delta_-^{\mathcal{P}}(j, J) := \{(i, I, j, J) \in \mathcal{A}^{\mathcal{P}}\}, && \forall\, (j, J) \in \mathcal{N}^{\mathcal{P}}.
\end{aligned}
$$

Based on these, we can define the types of time-expanded graphs.

Definition 6. *Let $\mathcal{P}$ be a time-window partition. The graph $\mathcal{G}^{\mathcal{P}} = (\mathcal{N}^{\mathcal{P}}, \mathcal{A}^{\mathcal{P}})$ is called the time-expanded graph given by $\mathcal{P}$. The time-expanded graph given by the complete time-window partition $\hat{\mathcal{P}}$ is called the completely time-expanded graph. Given an arc $(i, I, j, J) \in$*

$\mathcal{A}^{\mathcal{P}}$ then the difference between the earliest point represented by (j, J) and the earliest point represented by (i, I), i.e., $\min(J) - \min(I)$, is called the length of (i, I, j, J).

In the derivation of the refinement algorithms, we additionally use the following terms.

Definition 7. *Let P, P^* be two partitions of the same set I. The partition P^* is called a refinement of P, if for all $I^* \in P^*$ there exists a $J \in P$ such that $I^* \subseteq J$. If additionally $I \subset J$ for at least one $I^* \in P^*$ and one $J \in P$, then P^* is called a proper refinement.*

Note that, the only refinement that is not also a proper refinement is the identity.

Definition 8. *Let $\mathcal{P} = \{P_i \mid i \in N\}$ and $\mathcal{P}^* = \{P_i^* \mid i \in N\}$ be time-window partitions. The time-window partition $\mathcal{P}^*$ is called a refinement of $\mathcal{P}$ (and $\mathcal{P}$ an aggregation of $\mathcal{P}^*$) if P_i^* is a refinement of P_i for all $i \in N$. If additionally, P_i^* is a proper refinement of P_i for at least one $i \in N$, then we call $\mathcal{P}^*$ a proper refinement of $\mathcal{P}$.*

Now, we can derive refinement algorithms for two time-dependent problems on G, the SPPTW and the TSPTW.

5.1 A Refinement Algorithm for the SPPTW

Given two nodes $s, t \in N$, $s \neq t$ the SPPTW asks for a path $p = (p_1, \dots, p_k)$ in G with $s = p_1$ and $t = p_k$ that has minimal cost among all paths that are time-feasible, i.e., for which there exist $\theta_{p_j} \in I_{p_j}$ for all $j \in \{1, \dots, k\}$, such that $\theta_{p_j} \geq \theta_{p_{j-1}} + d_{(p_{j-1}, p_j)}$ for $j \in \{2, \dots, k\}$.

In this section we want to find solutions of the SPPTW on G by finding solutions of the SPP on time-expanded graphs $\mathcal{G}^{\mathcal{P}}$ given by time-window partitions $\mathcal{P}$. Solutions of the SPP on $\mathcal{G}^{\mathcal{P}}$ can be found by solving the LP-relaxation of the MILP

$$\min_{z \in \mathbb{Q}_{\geq}^{|\mathcal{A}|}} \quad \sum_{(i,j) \in A} c_{i,j} \sum_{\substack{I,J: \\ (i,I,j,J) \in \mathcal{A}^{\mathcal{P}}}} z_{i,I,j,J} \tag{8a}$$

s.t.

$$\sum_{\substack{(j,J): \\ (i,I,j,J) \in \mathcal{A}^{\mathcal{P}}}} z_{i,I,j,J} - \sum_{\substack{(j,J): \\ (j,J,i,I) \in \mathcal{A}^{\mathcal{P}}}} z_{j,J,i,I} = \begin{cases} 1, & (i,I) = (s, \{e_s\}), \\ -1, & (i,I) = (t, \{l_t\}), \\ 0, & \text{else}, \end{cases} \qquad \forall (i,I) \in \mathcal{N}^{\mathcal{P}} \tag{8b}$$

with the revised simplex algorithm. The path is then constructed from the arcs $(i, I, j, J) \in \mathcal{A}^{\mathcal{P}}$ for which $z_{i,I,j,J} = 1$.

For some fixed time-window partition $\mathcal{P}$ we denote the system (8) by $L_{\text{SPP}}^{\mathcal{P}}$. Additionally, we remark, that it suffices to consider the linear relaxation of $L_{\text{SPP}}^{\mathcal{P}}$, because the constraint matrix is totally unimodular and the right-hand side is integral, see for instance [22, 29].

Now, although solutions of the SPPTW can be found by directly solving $L_{\text{SPP}}^{\hat{\mathcal{P}}}$, our goal is to find solutions by using the general iterative refinement algorithm of the previous section. For doing so, we have to find aggregations of the LP relaxation of $L_{\text{SPP}}^{\hat{\mathcal{P}}}$.

We start with an example. Consider the graphs depicted in Figure 2. The graph on the right-hand side is the completely time-expanded graph of the one on the left-hand side and we assume that the costs are equal to the traveling times assigned to the arcs.

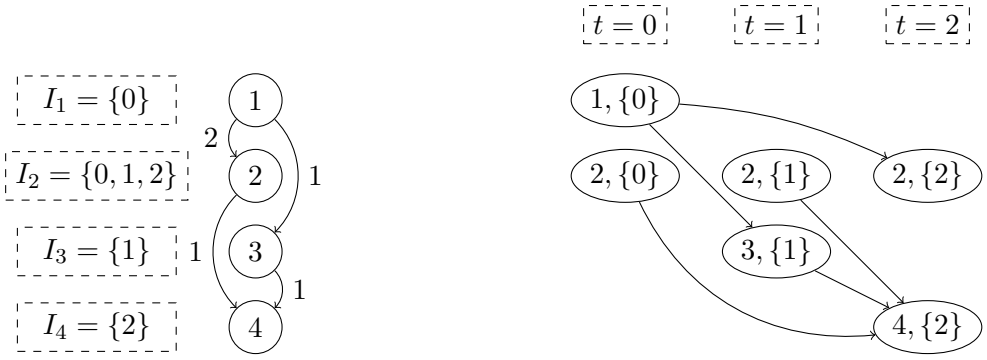


Figure 2: A graph with time-windows (left) and the corresponding time-expanded graph (right).

The LP formulation of the SPP with $s = 1$ and $t = 4$ on the graphs illustrated in Figure 2 are

$$\begin{aligned} \min_{z \in \mathbb{Q}^4} \quad & 2z_{1,2} + z_{1,3} + z_{2,4} + z_{3,4} \\ \text{s.t.} \quad & \begin{matrix} 1: \\ 2: \\ 3: \\ 4: \end{matrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & -1 \end{pmatrix} \begin{pmatrix} z_{1,2} \\ z_{1,3} \\ z_{2,4} \\ z_{3,4} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix} \end{aligned} \tag{LP1}$$

and

$$\begin{aligned} \min_{z \in \mathbb{Q}^5} \quad & 2z_{1,\{0\},2,\{2\}} + z_{1,\{0\},3,\{1\}} + z_{2,\{0\},4,\{2\}} + 2z_{2,\{1\},4,\{2\}} + z_{3,\{1\},4,\{2\}} \\ \text{s.t.} \quad & \begin{matrix} (1,\{0\}): \\ (2,\{0\}): \\ (2,\{1\}): \\ (2,\{2\}): \\ (3,\{1\}): \\ (4,\{2\}): \end{matrix} \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & -1 & -1 & -1 \end{pmatrix} \begin{pmatrix} z_{1,\{0\},2,\{2\}} \\ z_{1,\{0\},3,\{1\}} \\ z_{2,\{0\},4,\{2\}} \\ z_{2,\{1\},4,\{2\}} \\ z_{3,\{1\},4,\{2\}} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1 \end{pmatrix}. \end{aligned} \tag{LP2}$$

A simple calculation verifies that (LP1) is a (C, V)-aggregation of (LP2) with

$$V = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, C = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

We remark, that this example can be adapted to any graph G and its completely time-expanded graph $\mathcal{G}^{\hat{\mathcal{P}}}$. For a fixed arc $(i, j) \in A$, the matrix V then aggregates all variables which are indexed by arcs $(i, I, j, J) \in \mathcal{A}^{\hat{\mathcal{P}}}_{i,j}$. Therefore, it has a row for each original arc $(i, j) \in A$ and a column for each arc $(i', I', j', J') \in \mathcal{A}^{\hat{\mathcal{P}}}$. Its entries are 1 if $(i', j') = (i, j)$ and 0 otherwise. The matrix C aggregates the constraints of all the nodes $(i, I) \in \mathcal{N}^{\hat{\mathcal{P}}}_i$, for all $i \in N$. So, it has a row for each original node $i \in N$ and a column for each node $(i', I') \in \mathcal{N}^{\hat{\mathcal{P}}}$. Its entries are 1 if $i = i'$ and 0 otherwise.

Now, note that, if $\mathcal{P} = \{\{I_i\} \mid i \in N\}$, then G has the same topology as $\mathcal{G}^{\mathcal{P}}$. So, we have given an example of a time-window partition $\mathcal{P}$, for which $L^{\mathcal{P}}_{\text{SPP}}$ is an aggregation of $L^{\hat{\mathcal{P}}}_{\text{SPP}}$. The following results show, that this actually holds for any time-window partition $\mathcal{P}$. We start by proving a lemma.

Lemma 4. *Let $\mathcal{P} = \{P_i \mid i \in N\}$ and $\mathcal{P}^* = \{P_i^* \mid i \in N\}$, be two time-window partitions. Furthermore, let $P_{\tilde{i}}^* = (P_{\tilde{i}} \cup \{I_{\tilde{i}}^n \cup I_{\tilde{i}}^{n+1}\}) \setminus \{I_{\tilde{i}}^n, I_{\tilde{i}}^{n+1}\}$ for some $n \in \{1, \dots, |P_{\tilde{i}}| - 1\}$ and some node $\tilde{i} \in N \setminus \{s,t\}$, and $P_j = P_j^*$ for all nodes $j \in N \setminus \{\tilde{i}\}$. Then there exist matrices C, V such that $L_{\mathrm{SPP}}^{\mathcal{P}^*}$ is a (C,V)-aggregation of $L_{\mathrm{SPP}}^{\mathcal{P}}$.*

Proof. For proving the assertion, we find two matrices C, V for which all conditions in Definition 3 hold true. Since there are no integer constraints, in fact we do not have to assign a block structure to V.

Let $\tilde{I} := I_{\tilde{i}}^n \cup I_{\tilde{i}}^{n+1}$. Then, we define for $(i,I,j,J) \in \mathcal{A}^{\mathcal{P}^*}$

$$\omega(i,I,j,J) := \begin{cases} \{(i, I_i^n, j, J), (i, I_i^{n+1}, j, J)\} \cap \mathcal{A}^{\mathcal{P}}, & (i,I) = (\tilde{i}, \tilde{I}), \\ \{(i, I, j, I_i^n), (i, I, j, I_i^{n+1})\} \cap \mathcal{A}^{\mathcal{P}}, & (j,J) = (\tilde{i}, \tilde{I}), \\ \{(i,I,j,J)\}, & \text{else.} \end{cases}$$

For the nodes $(i,I) \in \mathcal{N}^{\mathcal{P}^*}$ we define

$$\psi(i,I) := \begin{cases} \{(i, I_i^n), (i, I_i^{n+1})\}, & (i,I) = (\tilde{i}, \tilde{I}), \\ \{(i,I)\}, & \text{else.} \end{cases}$$

These sets are defined precisely in such a way that it holds

$$\omega(\delta_+^{\mathcal{P}^*}(i,I)) = \delta_+^{\mathcal{P}}(\psi(i,I)), \qquad \forall (i,I) \in \mathcal{N}^{\mathcal{P}^*}, \tag{9a}$$

$$\omega(\delta_-^{\mathcal{P}^*}(i,I)) = \delta_-^{\mathcal{P}}(\psi(i,I)), \qquad \forall (i,I) \in \mathcal{N}^{\mathcal{P}^*}. \tag{9b}$$

Now, let C be the matrix that realizes the constraint aggregation with $\alpha = 1$ of the rows of $L_{\mathrm{SPP}}^{\mathcal{P}}$for $(\tilde{i}, I_{\tilde{i}}^n)$ and $(\tilde{i}, I_{\tilde{i}}^{n+1})$. Since $\tilde{i} \notin \{s,t\}$ (and hence the right-hand sides of both aggregated constraints are 0), the aggregated constraints are

$$\sum_{\substack{(j,J):\\ (i,I,j,J) \in \delta_+^{\mathcal{P}}(\psi(i,I))}} z_{i,I,j,J} - \sum_{\substack{(j,J):\\ (j,J,i,I) \in \delta_-^{\mathcal{P}}(\psi(i,I))}} z_{j,J,i,I} = \begin{cases} 1, & (i,I) = (s, \{e_s\}), \\ -1, & (i,I) = (t, \{l_t\}), \\ 0, & \text{else,} \end{cases} \quad \forall (i,I) \in \mathcal{N}^{\mathcal{P}^*}. \tag{10}$$

Now, define $V \in \mathbb{Q}^{|\mathcal{A}^{\mathcal{P}^*}| \times |\mathcal{A}^{\mathcal{P}}|}$ as the matrix that realizes the variable refinement

$$(Vz)_{i,I,j,J} = \sum_{(i',I',j',J') \in \omega(i,I,j,J)} z_{i',I',j',J'} \qquad \forall (i,I,j,J) \in \mathcal{A}^{\mathcal{P}^*}. \tag{11}$$

The refined constraints of $L_{\mathrm{SPP}}^{\mathcal{P}^*}$ are

$$\sum_{\substack{(j,J):\\ (i,I,j,J) \in \delta_+^{\mathcal{P}^*}(i,I)}} \sum_{\substack{(i',I',j',J') \in \\ \omega(i,I,j,J)}} z_{i',I',j',J'} - \sum_{\substack{(j,J):\\ (j,J,i,I) \in \delta_-^{\mathcal{P}^*}(i,I)}} \sum_{\substack{(j',J',i',I') \in \\ \omega(j,J,i,I)}} z_{j',J',i',I'} = \begin{cases} 1, & (i,I) = (s, \{e_s\}), \\ -1, & (i,I) = (t, \{l_t\}), \\ 0, & \text{else,} \end{cases} \quad \forall (i,I) \in \mathcal{N}^{\mathcal{P}^*}.$$

Since the sets in the image of ω are pairwise disjoint, this is equivalent to

$$\sum_{(i',I',j',J')\in\omega(\delta^{\mathcal{P}^*}_+(i,I))} z_{i',I',j',J'} - \sum_{(j',J',i',I')\in\omega(\delta^{\mathcal{P}^*}_-(i,I))} z_{j',J',i',I'} = \begin{cases} 1, & (i,I) = (s,\{e_s\}), \\ -1, & (i,I) = (t,\{l_t\}), \\ 0, & \text{else}, \end{cases} \quad \forall (i,I) \in \mathcal{N}^{\mathcal{P}^*}. \quad (12)$$

Now, by the relations in (9), (12) is equivalent to (10). Therefore, holds for the corresponding matrices $CA^{\mathcal{P}}_{\text{SPP}} = A^{\mathcal{P}^*}_{\text{SPP}}V$ and $Cb^{\mathcal{P}}_{\text{SPP}} = b^{\mathcal{P}^*}_{\text{SPP}}$. Additionally, substituting (11) in the objective function of $L^{\mathcal{P}^*}_{\text{SPP}}$ directly yields that also $c^{\mathcal{P}}_{\text{SPP}} = V^\top c^{\mathcal{P}^*}_{\text{SPP}}$, which concludes the proof. □

Using the result of this lemma, we can proof the following result.

Proposition 3. *Given a time-window partition $\mathcal{P}$, then:*

(i) $L^{\mathcal{P}^*}_{\text{SPP}}$ is an aggregation of $L^{\mathcal{P}}_{\text{SPP}}$for any aggregation $\mathcal{P}^*$ of $\mathcal{P}$,

(ii) $L^{\mathcal{P}^*}_{\text{SPP}}$ is a refinement of $L^{\mathcal{P}}_{\text{SPP}}$for any refinement $\mathcal{P}^*$ of $\mathcal{P}$,

(iii) $L^{\mathcal{P}}_{\text{SPP}}$is an aggregation of the time-indexed formulation $L^{\hat{\mathcal{P}}}_{\text{SPP}}$.

Proof.

(i) If $\mathcal{P}^*$ is an aggregation of $\mathcal{P}$, we can split the time-windows repeatedly in such a way that we obtain a sequence $\mathcal{P}_1, \dots, \mathcal{P}_n$ of aggregations, such that $\mathcal{P}_1 = \mathcal{P}$ and $\mathcal{P}_n = \mathcal{P}^*$, and P_i and P_{i+1} are related as the partitions in Lemma 4. For P_i and P_{i+1} then holds $L^{\mathcal{P}_i}_{\text{SPP}}$ is an aggregation of $L^{\mathcal{P}_{i+1}}_{\text{SPP}}$. So we can apply Lemma 1 inductively to show that $L^{\mathcal{P}^*}_{\text{SPP}}$ is an aggregation of $L^{\mathcal{P}}_{\text{SPP}}$.

(ii) This follows by reversing the roles of $\mathcal{P}$ and $\mathcal{P}^*$ in statement (i).

(iii) This is an application of statement (i) to $\mathcal{P}^* = \hat{\mathcal{P}}$.

□

Algorithm 3: Iterative Refinement Algorithm for the SPPTW

```
Input: The LP L^P̂_SPP and a time-window partitioning P, for an SPPTW instance.
while feas(L^P_SPP) ≠ ∅ do
    Find y ∈ opt(L^P_SPP);
    if y is expandable then
        return (c^P_SPP)^⊤ y;
    else
        (Refine: ) Find a proper refinement of P_new of P;
        P ← P_new;
return +∞;
```

So, by Proposition 3 all time-expanded formulations $L^{\mathcal{P}}_{\text{SPP}}$ are aggregations of $L^{\hat{\mathcal{P}}}_{\text{SPP}}$, and by Theorem 1, Algorithm 1 can be used to solve an MILP (and therefore also its LP relaxation), by solving only aggregations of it. Together, this implies that we can

apply Algorithm 1 to find solutions of the SPPTW by only solving $L^{\mathcal{P}}_{\text{SPP}}$ for more and more refined time-window partitions $\mathcal{P}$. The details are given in Algorithm 3. Because Algorithm 3 is an iterative refinement algorithm, by Theorem 1 we can state the following result.

Proposition 4. *Algorithm 3 terminates after finitely many steps. The return value is* $\text{opt}(L^{\hat{\mathcal{P}}}_{\text{SPP}})$.

In Section 6 we discuss an implementation of Algorithm 3 and perform computational experiments on randomly generated SPPTW instances.

5.2 Refinement Algorithms for the TSPTW

Given the graph G, the TSPTW asks for a tour visiting all nodes in the graph G of minimal cost that is starting and ending in the depot node 1 and that is time-feasible. Given a time-window partition $\mathcal{P}$, consider the following MILP:

$$\min_{\substack{x\in\{0,1\}^{|A|},\\ z\in\{0,1\}^{|\mathcal{A}^{\mathcal{P}}|}}} \quad \sum_{(i,j)\in A} c_{i,j} \sum_{\substack{I,J:\\ (i,I,j,J)\in\mathcal{A}^{\mathcal{P}}}} z_{i,I,j,J} \tag{13a}$$

s.t.

$$\sum_{\substack{(j,J):\\ (i,I,j,J)\in\mathcal{A}^{\mathcal{P}}}} z_{i,I,j,J} - \sum_{\substack{(j,J):\\ (j,J,i,I)\in\mathcal{A}^{\mathcal{P}}}} z_{j,J,i,I} = \begin{cases} 1, & (i,I) = (s,\{e_s\}), \\ -1, & (i,I) = (t,\{l_t\}), \\ 0, & \text{else}, \end{cases} \qquad \forall (i,I) \in \mathcal{N}^{\mathcal{P}}, \tag{13b}$$

$$\sum_{i:(i,j)\in A} x_{i,j} = 1, \qquad \forall\, j \in N, \tag{13c}$$

$$\sum_{I,J:(i,I,j,J)\in\mathcal{A}^{\mathcal{P}}} z_{i,I,j,J} - x_{i,j} = 0, \qquad \forall\, (i,j) \in A, \tag{13d}$$

This is $L^{\mathcal{P}}_{\text{SPP}}$ with additional binary variables $x_{i,j}$ for all $(i,j) \in A$ and two additional types of constraints. For some fixed time-window partition $\mathcal{P}$ we denote the system (13) by $M^{\mathcal{P}}_{\text{TSP}}$. The x-variables, similar to the z-variables are used to describe a path, but in G instead of $\mathcal{G}^{\mathcal{P}}$. The additional constraints in (13c) ensure that each node of G is part of the path and the constraints in (13d) then link the paths described by the x and z-variables. For any arc $(i,j) \in A$, some time indexed arc $(i,I,j,J) \in \mathcal{A}^{\mathcal{P}}_{i,j}$ has to be chosen, ensuring that they describe the same path in G once the time-indices are ignored. If all arcs in $(i,I,j,J) \in \mathcal{A}^{\mathcal{P}}$ have positive length and hence $\mathcal{G}^{\mathcal{P}}$ is cycle-free, this guarantees that no subtours can be included. Additionally, if all arcs lengths are greater or equal the traversing time it also guarantees that the tour is time-feasible. So, if s and t are two copies of the depot node, $M^{\hat{\mathcal{P}}}_{\text{TSP}}$ is a valid MILP formulation of the TSPTW.

Since $M^{\mathcal{P}}_{\text{TSP}}$ is very similar to $L^{\mathcal{P}}_{\text{SPP}}$, the following results and proofs are also similar to the ones for the SPPTW. Therefore, we omit the details of the arguments in the proofs, if they already appeared in the proofs of the previous section. The first result implies that we can apply the refinement algorithms of the previous section to the TSPTW.

Lemma 5. *Let* $\mathcal{P} = \{P_i \mid i \in N\}$ *and* $\mathcal{P}^* = \{P^*_i \mid i \in N\}$*, be two time-window partitions. Furthermore, let* $P^*_{\tilde{i}} = (P_{\tilde{i}} \cup \{I^n_{\tilde{i}} \cup I^{n+1}_{\tilde{i}}\}) \setminus \{I^n_{\tilde{i}}, I^{n+1}_{\tilde{i}}\}$ *for some* $n \in \{1,\dots,|P_{\tilde{i}}|-1\}$ *and some node* $\tilde{i} \in N \setminus \{s,t\}$*, and* $P_j = P^*_j$ *for all nodes* $j \in N \setminus \{\tilde{i}\}$*. Then, there exist matrices* C, V *such that* $M^{\mathcal{P}^*}_{\text{TSP}}$ *is a* (C,V)*-aggregation of* $M^{\mathcal{P}}_{\text{TSP}}$.

Proof. Using the same notations as in the proof of Lemma 4, let V be given by

$$(Vz)_{i,I,j,J} = \sum_{(i',I',j',J') \in \omega(i,I,j,J)} z_{i',I',j',J'} \qquad \forall (i,I,j,J) \in \mathcal{A}^{\mathcal{P}^*}$$
$$(Vx)_{i,j} = x_{i,j} \qquad \forall (i,j) \in A$$

and C be the constraint aggregation of the constraints of $M^{\mathcal{P}}_{\text{TSP}}$ for $(\tilde{i}, I^n_{\tilde{i}})$ and $(\tilde{i}, I^{n+1}_{\tilde{i}})$. We remark, that V has the required block structure, because there are no non-integer variables. Then we can apply the same arguments as in the proof of Lemma 4 to show that $CA^{\mathcal{P}}_{\text{TSP}} = A^{\mathcal{P}^*}_{\text{TSP}} V$, $Cb^{\mathcal{P}}_{\text{TSP}} = b^{\mathcal{P}^*}_{\text{TSP}}$ and $c^{\mathcal{P}}_{\text{TSP}} = V^\top c^{\mathcal{P}^*}_{\text{TSP}}$. □

Proposition 5. *Given a time-window partition $\mathcal{P}$, then:*

1. *For any aggregation $\mathcal{P}^*$ of $\mathcal{P}$ is $M^{\mathcal{P}^*}_{\text{TSP}}$ an aggregation of $M^{\mathcal{P}}_{\text{TSP}}$,*
2. *For any refinement $\mathcal{P}^*$ of $\mathcal{P}$ is $M^{\mathcal{P}^*}_{\text{TSP}}$ a refinement of $M^{\mathcal{P}}_{\text{TSP}}$,*
3. *The $M^{\mathcal{P}}_{\text{TSP}}$ is an aggregation of the time-indexed formulation $M^{\hat{\mathcal{P}}}_{\text{TSP}}$.*

Proof. The same arguments stated in the proof of Proposition 3 can also be applied here, by using Lemma 5 instead of Lemma 4. □

Before we go into the details of the refinement algorithms, we note that if the partition $\mathcal{P}$ is not chosen in such a way that all arcs of $\mathcal{A}^{\mathcal{P}}$ have positive length, then it is possible for solutions of $M^{\mathcal{P}}_{\text{TSP}}$ to contain subtours. Hence, it can be strengthened by subtour elimination constraints. For the TSPTW the so-called (π)-inequalities and (σ)-inequalities introduced by Balas et al. [3] dominate the traditional subtour elimination constraints. We express these inequalities in the notations of Dash et al. [9] and write $i \prec j$ for nodes $i, j \in N$ if in any time-feasible tour the time-step assigned to j is larger than the one assigned to i. For sets $S, S' \subset N$, we write $S \prec S'$, if for all $i \in S$ and $i' \in S'$ holds $i \prec i'$. Now, for any $S \subseteq N$ we define $\pi(S) := \{i \prec j \text{ for some } j \in S \mid i \in N\}$ and $\sigma(S) := \{i \prec j \text{ for some } i \in S \mid j \in N\}$. Furthermore, for any set $S \subseteq N$ let

$$\delta_\pi(S) := \{(i,j) \in A \mid i \in S \setminus \pi(S), j \in N \setminus (\pi(S) \cup S)\},$$
$$\delta_\sigma(S) := \{(i,j) \in A \mid i \in S \setminus \sigma(N \setminus S), j \in N \setminus (\sigma(N \setminus S) \cup S)\}.$$

Then, the following inequalities can be added to $M^{\mathcal{P}}_{\text{TSP}}$ without removing any expandable solutions:

$$\sum_{(i,j) \in \delta_\pi(S)} x_{i,j} \geq 1, \qquad \forall\, S \subseteq N \setminus \{1\}, \tag{π}$$
$$\sum_{(i,j) \in \delta_\sigma(N \setminus S)} x_{i,j} \geq 1, \qquad \forall\, S \subseteq N \setminus \{1\}. \tag{σ}$$

The details of an Iterative Refinement Algorithm for the TSPTW making use of these inequalities are given in Algorithm 4. We now have to show, that this is in fact an iterative refinement algorithm.

Proposition 6. *Algorithm 4 terminates after finitely many steps. The return value is* $\text{opt}(M^{\hat{\mathcal{P}}}_{\text{TSP}})$.

Algorithm 4: Iterative Refinement for the TSPTW

```
1  Input: A time-window partitioning 𝒫.
2  M_agg ← M_TSP^𝒫.
3  while feas(M_agg) ≠ ∅ do
4      ℒ_O ← {M_agg}, ℒ_C ← ∅, U ← ∞, y* ← ∅;
5      while ℒ_O ≠ ∅ do
6          Choose M* ∈ ℒ_O;
7          if feas(L*) = ∅ or opt(L*) ≥ U then
8              Move M* from ℒ_O to ℒ_C;
9          else
10             Find y ∈ opt(L*);
11             if y ∈ feas(M_agg) then
12                 U ← c_agg^⊤ y;
13                 y* ← y;
14                 Move M* from ℒ_O to ℒ_C;
15             else
16                 (Branch:) Find a non-integral entry y_k ∉ {0,1} of y . Divide M* into
                   two problems M_1^* and M_2^*, which are obtained by adding inequalities
                   that set the k-th variable to 0 or 1, respectively. Add M_1^*, M_2^* to ℒ_O
                   and move M* from ℒ_O to ℒ_C;
17     if y* is expandable then
18         return U;
19     else
20         if y* contains a subtour S then
21             Add the constraints (π) and (σ) for S to M_agg;
22         else
23             Find a refinement 𝒫_new of 𝒫;
24             𝒫 ← 𝒫_new, M_agg ← M_TSP^{𝒫_new};
25             Add all previously added (π) and (σ) constraints to M_agg;
26 return ∅;
```

Proof. We show that Algorithm 4 is a variant of Algorithm 1. The inner loop line 5-16 is just a branch-and-bound algorithm solving M_{agg} and hence matches line 3 of Algorithm 1.

Now, note that we can implicitly assume that all constraints (π) and (σ) are included in M. Therefore adding them in line 21 is a type of refinement. By Proposition 5 this is also true for the MILP found in line 23. These are, therefore, just two possibilities of finding the refinement in line 7 of Algorithm 1.

So, Algorithm 4 is indeed a variant of Algorithm 1 and the assertion holds by Theorem 1. □

The details of applying branch-and-refine are given in Algorithm 5. Note, that by the chosen transformation, none of the transformed variables show up in the additional inequalities added in the branching step.

Proposition 7. *Algorithm 5 terminates after finitely many steps. The return value is* $\text{opt}(M_{\text{TSP}}^{\hat{\mathcal{P}}})$.

Proof. For the second part of the assertion it suffices to show that Algorithm 5 is a variant of Algorithm 2, because it then directly follows from Theorem 2.

Algorithm 5: Branch-and-refine for the TSPTW

```
1  Input: A time-window-partitioning 𝒫.
2  ℒ_O ← {M_TSP^𝒫},ℒ_C ← ∅,U ← ∞,S ← ∅;
3  S ← ∅;
4  while ℒ_O ≠ ∅ do
5      Choose M* ∈ ℒ_O;
6      if feas(L*) = ∅ or opt(L*) ≥ U then
7          Move M* from ℒ_O to ℒ_C;
8      else
9          Find y ∈ opt(L*);
10         if y ∈ feas(M_agg) then
11             if y is expandable then
12                 U ← c_agg^⊤ y;
13                 Move M* from ℒ_O to ℒ_C;
14             else
15                 if y contains a subtour S then
16                     Add the constraints (π) and (σ) for S to M_agg;
17                 else
18                     Find a refinement 𝒫_new of 𝒫;
19                     𝒫 ← 𝒫_new, M_agg ←M_TSP^𝒫new;
20                     Add all previously added (π) and (σ) constraints to M_agg;
21                 Transform each problem in ℒ_O in accordance with Lemma 2;
22         else
23             (Branch:) Find a non-integral entry y_k ∉ {0,1} of y . Divide M* into two
               problems M_1* and M_2*, which are obtained by adding inequalities that set
               the k-th variable to 0 or 1, respectively. Add M_1*, M_2* to ℒ_O and move M*
               from ℒ_O to ℒ_C;
24 return U;
```

The lines 15-20 of Algorithm 5 are identical to the lines 20-25 of Algorithm 4. So, by the same argument as in the proof of Proposition 6, these are just two possibilities of finding refinements of M_{agg}, which is required in line 14 of Algorithm 2.

Furthermore, the 'Branch' step in line 23 is just branching on an integer variable that has a fractional value in the solution of the LP-relaxation and hence it holds $\text{feas}(M^*) = \text{feas}(M_1^*) \cup \text{feas}(M_2^*)$ and it matches line 17 of Algorithm 2. The other lines follow the structure of Algorithm 2, and we conclude that Algorithm 5 is a variant of Algorithm 2.

For proving that Algorithm 5 terminates after finitely many steps, we first note that in each iteration of the while loop, there are only two cases, in which the size of $\mathcal{L}_O$ does not decrease: Either the branching step in line 23 is reached, or the MILP M_{agg} is replaced by a refinement in line 11-21. In the branching step a node is replaced by two nodes, whose MILPs have an additional constraints, which fixes the value of a binary variable and restricts the depths of the search tree to the number of variables in M. In line 11-21 of Algorithm 5 M_{agg} is replaced by a proper refinement, which can only be done finitely many times, because either the number of variables or the number of constraints of M_{agg} is increased. So, after finitely many steps all nodes are guaranteed to be removed from $\mathcal{L}_O$ and none added anymore. It then directly follows that after finitely many iterations has to hold $\mathcal{L}_O = \emptyset$, and Algorithm 5 terminates in line 24. □

With this, we have shown that both, branch-and-refine and the general iterative refinement algorithm, can be used to solve the TSPTW. Taking a closer look at the pseudo-code of Algorithm 4 and Algorithm 5 the aforementioned similarities between the two refinement algorithms become apparent. The only conceptual difference is that, in Algorithm 5 all branching decisions are preserved when the MILP is refined, and Algorithm 4 always starts with a newly initiated branch-and-bound tree in each iteration. Both approaches have their advantages and disadvantages. In Algorithm 5, it is possible that the time-window partition is refined based on solutions that would be cut off later in the branch-and-bound process anyway. In Algorithm 4, the aggregated MILP is solved to optimality before refining, so this cannot happen. On the other hand, in Algorithm 4, no knowledge obtained from the branching decisions is carried over to the subsequent iterations, and, therefore, similar branching decisions might be repeated in each iteration reconstructing the discarded search tree. In the following section we present computational results about the computational performance of these algorithms.

6 Implementation Details and Computational Experiments

In this section we present strategies for implementing Algorithms 3-5 and also the results of a computational study that compares these implementations with each other. All experiments were run on a 2018 MacMini with a 3.2 GHz i7 processor and 64 GB of RAM. All linear programs were solved with the dual simplex algorithm using IBM ILOG CPLEX 12.10.0.0 restricted to use one thread. All other parameters were set to their default values unless specified otherwise.

Before we go into specific implementation details of the two applications, we note here that in Section 5 the definition of the time-expanded graphs for a time-window partition $\mathcal{P}$ includes many arcs that model an unnecessarily long traveling time. While these are very useful for establishing the theoretical results, including them in the LPs and MILPs of our computational study makes the models unnecessarily large, because any solution using those arcs can be replaced by one that includes the same nodes, but only uses the arcs with the shortest traveling times. Our implementations, therefore, use the following definition of $\mathcal{A}^{\mathcal{P}}_{i,j}$:

$$\mathcal{A}^{\mathcal{P}}_{i,j} := \{\, (i, I, j, J) \in \mathcal{N}^{\mathcal{P}}_i \times \mathcal{N}^{\mathcal{P}}_j \mid \max(\{\min(I) + d_{i,j}, e_j\}) \in J \}\,.$$

This definition guarantees that each node $(i, I) \in \mathcal{N}^{\mathcal{P}}_i$ can be connected to at most one node $(j, J) \in \mathcal{N}^{\mathcal{P}}_j$ which has to be earliest reachable one.

6.1 Implementation details of the iterative refinement algorithm for the SPPTW

In our computational study for the SPPTW we investigate the performance of Algorithm 3, the iterative refinement algorithm. Since this algorithm solves $L^{\hat{\mathcal{P}}}_{\text{SPP}}$, we compare it to a direct solution approach using the LP solver.

The pseudo-code in Algorithm 3 is formulated in general terms which allow a variety of different implementations. More precisely, it does not specify: what the initial time-window partition $\mathcal{P}$ is, how to find y in line 3, how to check if y is expandable in line 4, and how to chose the proper refinement $\mathcal{P}_{\text{new}}$.

We use the unpartitioned time-windows as the initial time-window partition $\mathcal{P}$.

In our implementations of line 3 we solve $L^{\mathcal{P}}_{\text{SPP}}$ in three different ways: (1) with the 'advance' routine of IBM ILOG CPLEX switched off (no information from previous iterations is used), (2) with the advance routine switched on but no additional user input (if only some of the constraints and variables are changed, IBM ILOG CPLEX has an internal

routine for advancing from the previously calculated optimal solution), and configuration (3), where the advance routine is switched on and a start vector given by the formula in Proposition 2 is used.

For checking whether y is expandable in line 3, we first reconstruct the path in G described by the arcs in y with value 1. Afterwards we calculate the earliest time steps θ_i, at which each node i of the path can be reached, when following the path. If $\theta_i \notin I_i$ for one of the nodes, then y cannot be expandable. So, we do not have to actually work with the time-expanded graph to check for feasibility of y.

Because we always have to check if y is expandable, before we try to find a refinement line 7, we can use the θ_i in our implementation of that line. By construction of the time-expanded graphs, we know that there exists at least one node $i \in N$, for which $\theta_i \in I_i$, but it is not the minimum of an interval of the partition of the time-window I_i (the arcs chosen in y would otherwise represent the correct traveling times and it would be expandable). Therefore, when we split the partition of each node at θ_i, at least one partition is changed and we find a proper refinement of the current time-window refinement.

6.2 Computational experiments for the SPPTW

In our computational study for the SPPTW we investigate the effectiveness of Algorithm 3 for solving instances of the SPPTW, i.e., $L^{\hat{\mathcal{P}}}_{\text{SPP}}$ for different graphs G. We, therefore, compare the summed up time the LP solver takes during the execution of Algorithm 3 to the time it takes the LP-solver to solve $L^{\hat{\mathcal{P}}}_{\text{SPP}}$ directly.

The SPPTW instances that we used were constructed in the following way.

1. Randomly generate the graph $G = (N, A)$ with start node s and terminal node t such that each node is included in some s-t path.

2. Assign random integer values to the parameters $c_{i,j}$, $d_{i,j}$ for all $(i, j) \in A$.

3. Set $I_s = \{0\}$ and use the earliest time each node $i \in N \setminus \{0\}$ can be reached from the start node s as the lower bound of its time-window e_i.

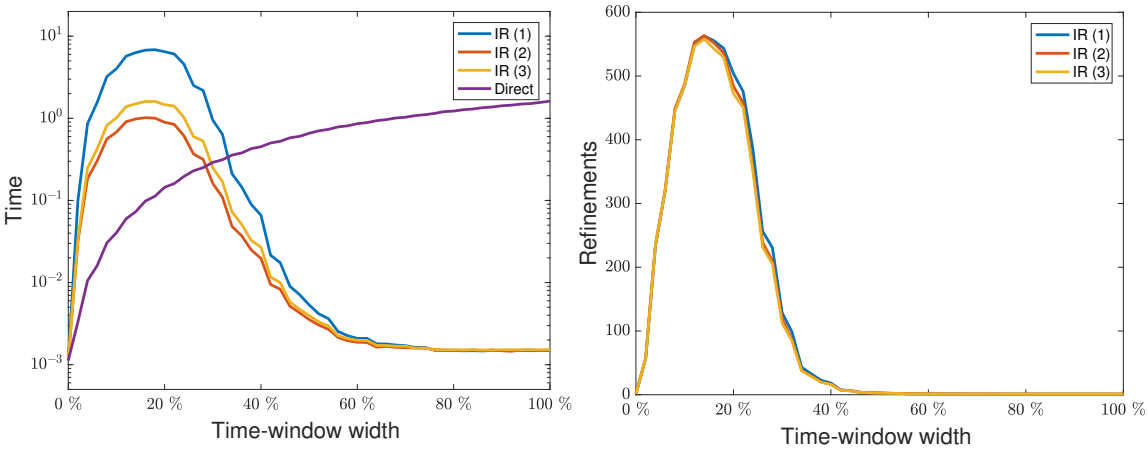
4. For t choose a value l_t such that $l_t > e_t$ and for each node $i \in N \setminus \{0\}$ assign the latest time t can be reached from it to l_i.

Following these steps we generated 50 graphs with 250 nodes plus start and terminal node. For investigating the effect of the restrictions imposed by the time-windows on the computation time, we generated 50 instances from each of these graphs by incrementally reducing the upper bounds l_i for all nodes $i \in N$ proportional to the width $l_i - e_i$ of the original time-window and rounding to integers. All instances are available for download via DOI 10.26127/btuopen-5199.

An overview of the computational results for the three different configurations of Algorithm 3 and directly solving $L^{\hat{\mathcal{P}}}_{\text{SPP}}$ are given in Figure 3.

The horizontal axes of the plots are the width of the time-windows in percentages of the width of the original time-window. The vertical axis in Figure 3a has a logarithmic scale and shows the geometric mean of the computation time of solving the 50 randomly generated graphs. The number of refinement steps performed by the three configurations of Algorithm 3 is given in Figure 3b.

The graphs for directly solving $L^{\hat{\mathcal{P}}}_{\text{SPP}}$ is monotonously increasing, i.e., the wider the time-windows the higher the computation time. The shape of the graphs of the implementations of Algorithm 3 indicate a different effect of the width of the time-windows. The graphs are not monotonously increasing. They increase sharply at first and then after



(a) Geometric mean of computation times.

(b) Number of refinements.

Figure 3: Computational results for the SPPTW.

reaching a maximum at about 15 % monotonously decreasing until they are almost constant at about 70 %. The number of refinement steps mirrors this behaviour. So, solving instances with both large time-windows and very small time-windows, i.e., not very restrictive time-windows and very restrictive time-windows, needs much less time and fewer refinements than medium sized time-windows.

Comparing the graphs of the 3 parameter configurations of Algorithm 3 with each other, it becomes apparent that configuration (3) using the feasible warm-start performs best, closely followed by configuration (2). Configuration (1) without start vector, performed considerably worse than these two. Not visible in the figures is that in the experiments using configuration (2), IBM ILOG CPLEX almost always reported that the initial solution is infeasible, so it is surprising that it performed so much better than configuration (1).

Now, comparing Algorithm 3 set to parameter configuration (3) with the benchmark of directly solving $L^{\hat{\mathcal{P}}}_{\text{SPP}}$, the latter performed better on only about a third of the instances, the instances with the smallest time-windows. When the time-windows pose no restriction on the shortest path, the LP based on the completely time-expanded graph has much more constraints and variables than the aggregations used in Algorithm 3, whose solution in this case is expandable and can be calculated in much less time. In the other extreme case, the time-windows are so small that the time-expanded graphs have the same topology as the time-expanded graph, only one LP has to be solved and the computation times are the same. In between the extremes, there can be time-infeasible paths in the time-expanded graphs used during the execution of Algorithm 3. The more of these there are, the more likely it is for Algorithm 3 to need additional iterations and it performs worse. For practical applications we can conclude that it depends on the type of instances whether Algorithm 3 should be applied. Only if the time-windows are expected to be not very tight, should one consider using it.

6.3 Implementation details of the refinement algorithms for the TSPTW

In our computational study for the TSPTW we investigate the effectiveness of Algorithm 4 and Algorithm 5 for solving instances of the TSPTW, i.e., $L^{\hat{\mathcal{P}}}_{\text{SPP}}$ for different graphs G. We, therefore, compare the summed up time the LP solver takes during the execution of Algorithm 4 and Algorithm 5 to the time it takes the MILP-solver to solve $M^{\hat{\mathcal{P}}}_{\text{TSP}}$ directly.

In Algorithm 5 we do not specify, how to chose the node M^* in line 5 (node selection), the refinement $\mathcal{P}^*$ in line 18 (refinement selection), or the arc a in line 23 (variable

selection).

Finding the strategies that can cope with state-of-the-art, however, is not the focus of this work and we therefore make use of basic ideas that have been found effective in the literature for branch-and-bound frameworks and are also applicable to branch-and-refine.

The node selection strategy in our computational experiments is sometimes referred to as *best first*, meaning that we always chose the node for which we have the best known lower bound.

For variable selection we follow the pseudo-cost branching strategy by Eckstein [12]. The concept of pseudo-cost branching is to choose the branching variable based on how effective branching on it was in previously explored nodes of the branch-and-bound tree.

For the starting partition we used the full time-window of each node and for the refinement selection we implemented the recursive path refinement, which was introduced by Riedler et al. [25]. The steps in our notation are given in the following.

We denote the current time-window partition by $\mathcal{P}$, and if we found a subtour-free feasible integer solution y of the current MILP M_{agg} then the partition $\mathcal{P}_{\text{new}}$ is obtained by the following steps:

1. Reconstruct the path P in G described by the arcs in y with value 1 and calculate the earliest time-steps θ_i, at which each node i of the path can be reached, when their order is fixed by P.

2. For any node $i \in P$ such that $\theta_i \notin I_i$ backtrack in $\mathcal{G}^{\mathcal{P}}$ along the arcs of P and split the partition of the respective nodes. When the length of the arc is the traversing time, stop the backtracking.

3. Repeat this process until the path P is no longer present in $\mathcal{P}_{\text{new}}$.

Because we are just splitting intervals of the time-window partition $\mathcal{P}$ to obtain $\mathcal{P}_{\text{new}}$, the partition found in this way fits line 18 of Algorithm 5 and line 23 of Algorithm 4.

6.4 Computational experiments for the TSPTW

For the test instances of our computational experiments of the TSPTW, we constructed instances to investigate the impact of the restrictions imposed by the time-windows. We made use of the following steps to generate 30 TSPTW instances with 20 nodes each.

1. Choose a complete graph $G = (N, A)$. Randomly choose $|N|$ of points in the plane and assign their euclidean distance rounded to integers to the costs $c_{i,j}$.

2. Find a near optimal tour S for the maximization traveling salesman problem (TSP-MAX) on G. For each node $i \in N$ assign to $\overline{\theta}_i$ the time it the node is reached when following S with traveling times $d_{i,j} = c_{i,j}$. Assign to $\underline{\theta}_i$ the time each node is reached when going through S in reverse.

3. Find a near optimal solution S^* of the TSP on G and assign arrival times assign to θ_i^* the arrival times by following S^*.

4. For each node i assign one time-window bound to $\overline{\theta}_i$ and the other either to $\underline{\theta}_i$ or θ_i^* depending on which results in the bigger interval.

Because we fix one of the bounds to the arrival times of the feasible tour S, the instances generated in this way are guaranteed to have a solution. To produce interesting instances, we noticed that using one other feasible tour was not enough, so we chose both the reversed TSP-MAX solution and the TSP solution to produce the other interval bound. Analogous to the procedure described for the SPPTW, we varied the width of the time-windows for

the 30 instances generated in that way. Because we have to guarantee the existence of a tour, we cannot always fix the lower bound or upper bound, instead for all nodes $i \in N$ we fixed the bound that is equal to $\overline{\theta}_i$. All of these instances are available for download via DOI 10.26127/btuopen-5199.

In our computational experiments we tested five different approaches:

(1) A self-implemented branch-and-refine algorithm with feasible start vector (see Proposition 2) for the refined LPs,

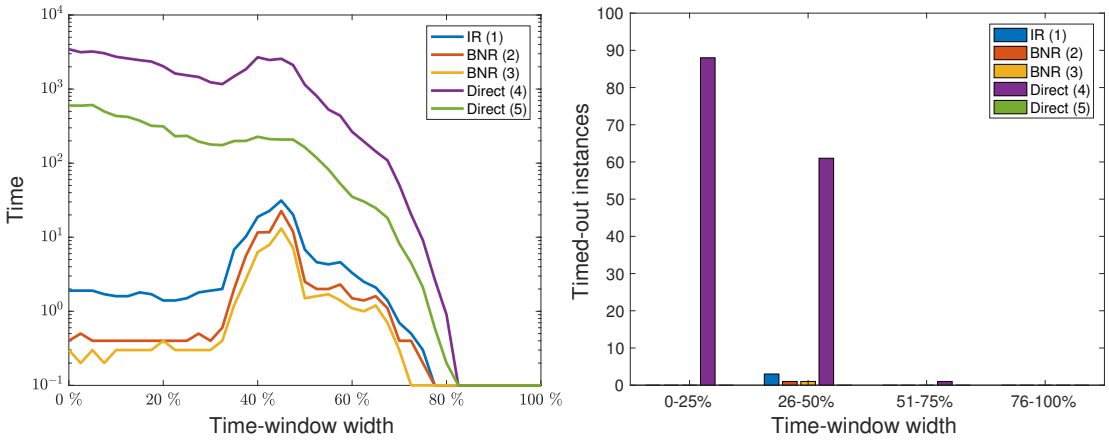
(2) The branch-and-refine algorithm from (1) with no start vector for the refined LPs,

(3) The general iterative refinement algorithm using IBM ILOG CPLEX 12.10.0.0 for solving the aggregated MILPs,

(4) Solving $M_{\text{TSP}}^{\hat{\mathcal{P}}}$ directly by the branch-and-refine algorithm from (1) which is just a branch-and-bound algorithm in this case,

(5) Solving $M_{\text{TSP}}^{\hat{\mathcal{P}}}$ by using IBM ILOG CPLEX 12.10.0.0.

The results of the computational experiments can be found in Figure 4.



(a) Geom. mean of computations times.

(b) Timed-out instances.

Figure 4: Computational results for the TSPTW.

The horizontal axes of the plot in Figure 4a are the width of the time-windows and the vertical axis in Figure 4a is the geometric mean of the computation times for the 30 randomly generated graphs. The number of timed-out instances are given in Figure 4a, where the different time-window widths are collected in four buckets for a better visualization.

The computation times for the implementation of Algorithm 4 and the two implementations of Algorithm 5 are very close to each other with a slight advantage for the branch-and-refine algorithm. The margin, however, is too small to conclude that this advantage persists for other types of instances, with for example more customers or clustered customers. The direct solution approaches, however, performed far worse, which matches the results found by Riedler et al. [25] on various benchmark instances. Comparing the two direct solution approaches, especially with the number of time-outs depicted in Figure 4b in mind, it is apparent that our branch-and-bound implementation performed far worse than the off-the-shelf solver. The results of the two implementations of branch-and-refine indicate that using feasible start vectors for the refined LPs reduces computation times. This effect is much less apparent than it was for the SPPTW. We attribute this mostly to the fact, that only the refined LPs are treated differently and not those created by branching.

7 Conclusions and Future Work

We introduced a relation between MILPs that we called aggregations/refinements. We then proved some basic results, which show that in some cases solving an aggregation of a MILP can be very helpful for solving the MILP itself. Based on these results we additionally proposed two refinement algorithms, the general iterative refinement algorithm and the branch-and-refine algorithm. For showing that these algorithms are not only of theoretical nature, but have potential for efficiently solving real-world problems, we discussed, how the SPPTW and the TSPTW can be solved by our algorithms and, with computational experiments, explored for which of problems instances they are the most promising.

The results of the computational experiments for the SPPTW suggest, that refinement algorithms should only be applied if the time-windows, and therefore the completely time-expanded graphs, are large. Furthermore, we showed that the revised simplex algorithm used for solving the LP relaxations can be initiated with a feasible starting vector derived from solutions of previous iterations. Making use of this proved to be computationally beneficial independent of the instance we used.

In case of the TSPTW, the refinement algorithms outperformed the direct approach of solving a MILP derived from the completely time-expanded graphs for all time-window widths. This observation matches the results obtained by Riedler et al. [25] on benchmark instances. We additionally showed that branch-and-refine, is an alternative to more established algorithms, which are structured like our general iterative refinement algorithm. The kind of MILP transformations required in our branch-and-refine algorithms, are not compatible with off-the-shelf solvers and therefore require more implementation effort. We do not see this as a general downside, because this also shows that there is a lot of unexplored potential for incorporating more sophisticated heuristics and preprocessing routines, which off-the-shelf solvers benefit from.

In this work we neither performed computational experiments for fine tuning our algorithms, nor did we investigate if they can compete with state-of-the-art solution methods from the literature. Therefore, exploring different strategies for node selection, variable selection or the time-window refinements in the branch-and-refine algorithm is part of future work. Additionally, we want to apply the algorithms to other time-dependent problems that can be modeled with a time-expanded formulation fitting our framework. Since the algorithms are not restricted to time-dependent problems it would be interesting to investigate other applications, that also allow for the use of aggregations. Finally, refinement algorithms have already been successfully applied to problems in which traversing times are time-dependent and make-span objectives are used, e.g. [27, 17]. The MILPs and relaxations they use do not fit our notion of aggregations, because the objective value is often strictly underestimated. In the terms of Definition 3 this would require to allow $c_1^\intercal x > c_2^\intercal V x$ for some solutions x. An adaptation of the branch-and-refine algorithm to these kinds of MILPs is another direction for future research.

Acknowledgements

The authors acknowledge the funding by the German Research Association (DFG), grant number FU 860/1-1.

References

[1] Leif H Appelgren. A column generation algorithm for a ship scheduling problem. *Transportation Science*, 3(1):53–68, 1969.

[2] Norbert Ascheuer, Matteo Fischetti, and Martin Grötschel. A polyhedral study of the asymmetric traveling salesman problem with time windows. *Networks: An International Journal*, 36(2):69–79, 2000.

[3] Egon Balas, Matteo Fischetti, and William R Pulleyblank. The precedence-constrained asymmetric traveling salesman polytope. *Mathematical Programming*, 68(1-3):241–265, 1995.

[4] Cynthia Barnhart, Ellis L Johnson, George L Nemhauser, Martin WP Savelsbergh, and Pamela H Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations research*, 46(3):316–329, 1998.

[5] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[6] Natashia Boland, Mike Hewitt, Luke Marshall, and Martin Savelsbergh. The price of discretizing time: a study in service network design. *EURO Journal on Transportation and Logistics*, 8(2):195–216, 2019.

[7] Natashia Boland, Mike Hewitt, Duc Minh Vu, and Martin Savelsbergh. Solving the traveling salesman problem with time windows through dynamically generated time-expanded networks. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 254–262. Springer, 2017.

[8] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research society of America*, 2(4):393–410, 1954.

[9] Sanjeeb Dash, Oktay Günlük, Andrea Lodi, and Andrea Tramontani. A time bucket formulation for the traveling salesman problem with time windows. *INFORMS Journal on Computing*, 24(1):132–147, 2012.

[10] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[11] Moshe Dror. Note on the complexity of the shortest path models for column generation in vrptw. *Operations Research*, 42(5):977–978, 1994.

[12] Jonathan Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the cm-5. *SIAM Journal on Optimization*, 4(4):794–814, 1994.

[13] Lester R Ford Jr. Network flow theory. Technical report, Rand Corp Santa Monica Ca, 1956.

[14] Armin Fügenschuh and Alexander Martin. Computational Integer Programming and Cutting Planes. In R. Weismantel K. Aardal, G.L. Nemhauser, editor, *Handbooks in Operations Research and Management Science, Vol. 12, Discrete Optimization*, pages 69–121. Elsevier, Amsterdam, 2005.

[15] Michel Gendreau, Alain Hertz, Gilbert Laporte, and Mihnea Stan. A generalized insertion heuristic for the traveling salesman problem with time windows. *Operations Research*, 46(3):330–335, 1998.

[16] Fabian Gnegel and Armin Fügenschuh. An iterative graph expansion approach for the scheduling and routing of airplanes. *Computers & Operations Research*, 114:104832, 2020.

[17] Edward He, Natashia Boland, George Nemhauser, and Martin Savelsbergh. A dynamic discretization discovery algorithm for the minimum duration time-dependent shortest path problem. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 289–297. Springer, 2018.

[18] Felipe Lagos, Natashia Boland, and Martin Savelsbergh. The continuous-time inventory-routing problem. *Transportation Science*, 54(2):375–399, 2020.

[19] Amos Levin. Scheduling and fleet routing models for transportation systems. *Transportation Science*, 5(3):232–255, 1971.

[20] Manuel López-Ibáñez and Christian Blum. Beam-aco for the travelling salesman problem with time windows. *Computers & Operations Research*, 37(9):1570–1583, 2010.

[21] Leonid Mirsky. *An introduction to linear algebra*. Courier Corporation, 2012.

[22] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience, USA, 1988.

[23] Jeffrey W Ohlmann and Barrett W Thomas. A compressed-annealing heuristic for the traveling salesman problem with time windows. *INFORMS Journal on Computing*, 19(1):80–90, 2007.

[24] Gilles Pesant, Michel Gendreau, Jean-Yves Potvin, and Jean-Marc Rousseau. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32(1):12–29, 1998.

[25] Martin Riedler, Mario Ruthmair, and Günther R Raidl. Strategies for iteratively refining layered graph models. In *International Workshop on Hybrid Metaheuristics*, pages 46–62. Springer, 2019.

[26] Martin Skutella. An introduction to network flows over time. In *Research trends in combinatorial optimization*, pages 451–482. Springer, 2009.

[27] Duc Minh Vu, Mike Hewitt, Natashia Boland, and Martin Savelsbergh. Dynamic discretization discovery for solving the time-dependent traveling salesman problem with time windows. *Transportation Science*, 2019.

[28] Xiubin Wang and Amelia C Regan. Local truckload pickup and delivery with hard time window constraints. *Transportation Research Part B: Methodological*, 36(2):97–112, 2002.

[29] L.A. Wolsey. *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 1998.

IMPRESSUM

Brandenburgische Technische Universität Cottbus-Senftenberg
Fakultät 1 | MINT - Mathematik, Informatik, Physik, Elektro- und Informationstechnik
Institut für Mathematik
Platz der Deutschen Einheit 1
D-03046 Cottbus

Professur für Ingenieurmathematik und Numerik der Optimierung
Professor Dr. rer. nat. Armin Fügenschuh

E fuegenschuh@b-tu.de
T +49 (0)355 69 3127
F +49 (0)355 69 2307

Cottbus Mathematical Preprints (COMP), ISSN (Print) 2627-4019
Cottbus Mathematical Preprints (COMP), ISSN (Online) 2627-6100

www.b-tu.de/cottbus-mathematical-preprints
cottbus-mathematical-preprints@b-tu.de
doi.org/10.26127/btuopen-5199