HAL open science

Exact and Anytime Approach for Solving the Time Dependent Traveling Salesman Problem with Time Windows

Romain Fontaine, Jilles Dibangoye, Christine Solnon

▶ **To cite this version:**

Romain Fontaine, Jilles Dibangoye, Christine Solnon. Exact and Anytime Approach for Solving the Time Dependent Traveling Salesman Problem with Time Windows. European Journal of Operational Research, In press, 10.1016/j.ejor.2023.06.001 . hal-04125860v2

HAL Id: hal-04125860

https://hal.science/hal-04125860v2

Submitted on 15 Jun 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Exact and Anytime Approach for Solving the Time Dependent Traveling Salesman Problem with Time Windows

Romain Fontaine*, Jilles Dibangoye, Christine Solnon

Univ Lyon, INSA Lyon, Inria, CITI, EA3720, 69621 Villeurbanne, France

Abstract

The Time Dependent (TD) Traveling Salesman Problem (TSP) is a generalization of the TSP which allows one to take traffic conditions into account when planning tours in an urban context, by making the travel time between locations dependent on the departure time instead of being constant. The TD-TSPTW further generalizes this problem by adding Time Window constraints. Existing exact approaches such as Integer Linear Programming and Dynamic Programming usually do not scale well. We therefore introduce a new exact approach based on an anytime extension of A*. We combine this approach with local search, to converge faster towards better solutions, and bounding and time window constraint propagation, to prune parts of the state space. We experimentally compare our approach with state-of-the-art approaches on both TD-TSPTW and TSPTW benchmarks.

Keywords: Travelling Salesman, Dynamic Programming, Time-Dependent cost functions

1. Introduction

The *Time Dependent* (TD) *Traveling Salesman Problem* (TSP) is a generalization of the TSP where travel times vary throughout the day, thus allowing one to take traffic conditions into account when planning delivery tours in an urban context. The TD-TSPTW further generalizes this problem by adding *Time*

*Corresponding author

Email addresses: romain.fontaine@insa-lyon.fr (Romain Fontaine), jilles-steeve.dibangoye@insa-lyon.fr (Jilles Dibangoye), christine.solnon@insa-lyon.fr (Christine Solnon)

Window (TW) constraints. The relevance of considering TD travel times in an urban context is studied in Rifki et al. (2020) on realistic data; it is shown that this reduces TW violations and also, in some cases, tour durations. However, this generalization makes the problem harder to solve and exact approaches such as *Constraint Programming* (CP), *Integer Linear Programming* (ILP), or *Dynamic Programming* (DP) do not scale well.

In this article, we propose a new DP-based approach which aims at finding approximate solutions quickly while being able to prove optimality. Notations and definitions are introduced in Section 2, and a literature review is done in Section 3. Our solving approach is presented in Section 4. It is based on Anytime Column Search (ACS), an anytime variant of A* introduced by Vadlamudi et al. (2012) which progressively widens the exploration of the DP state-transition graph. We combine ACS with TW constraint propagation (to filter the state space) and with local search (to improve upper bounds found by ACS). We also use bounding functions to guide the search, and we describe three bounds which provide different trade-offs between computational cost and tightness. We introduce new rules based on latest departure times to filter edges of the underlying graph as it is essential for computing better bounds. In Section 5, we present experimental results. We first analyse the benefits of combining ACS with TW constraint propagation, local search, and edge filtering by considering the results obtained when disabling these components. This allows us to show that our new filtering rules greatly improve the solving process. Then, we experimentally compare our approach with state-of-the-art approaches based on ILP (Vu et al., 2020; Arigliano et al., 2018b) and DP (Lera-Romero et al., 2022) on three different benchmarks. We show that our approach is able to find reference solutions much faster, and that it is also able to prove optimality on most instances. In Section 6, we experimentally evaluate our approach on classic TSPTW benchmarks, and we show that it outperforms the DP-based approach of Gillard et al. (2021) and the LS-based approach of Da Silva & Urrutia (2010). Finally, conclusions and future works are discussed in Section 7.

2. Definitions and Notations related to the TD-TSPTW

The set of vertices to visit is denoted $\mathcal{V} = \{0, \ldots, n\}$: 0 is the starting vertex, and n the ending vertex (in practice, 0 and n often refer to the same location, *i.e.*, the depot). $\mathcal{C} = \mathcal{V} \setminus \{0, n\}$ denotes the set of customer vertices. t_0 denotes the starting time from vertex 0. Given $i \in \mathcal{V}$, e_i and l_i respectively denote the

earliest and latest visit times of i. We assume that $e_0 = l_0 = e_n = t_0$. The latest visit time of n, l_n, represents the time horizon.

It is possible to arrive before e_i on i but, in this case, we have to wait on i. Given a time t and a TW $[e_i, l_i]$, we note $t_{\uparrow[e_i,l_i]}$ the TW-aware time that includes a waiting time whenever $t < e_i$ and returns ∞ whenever $t > l_i$, *i.e.*, $t_{\uparrow[e_i,l_i]} = e_i$ if $t < e_i$; $t_{\uparrow[e_i,l_i]} = t$ if $e_i \leq t \leq l_i$; and $t_{\uparrow[e_i,l_i]} = \infty$ if $t > l_i$.

Given $i, j \in \mathcal{V}$, $c_{i,j}$ denotes the TD cost function such that $c_{i,j}(t)$ is the travel time from i to j when leaving i at time t, and $a_{i,j}$ denotes the arrival time function such that $a_{i,j}(t) = t + c_{i,j}(t)$. The inverse of $a_{i,j}$ is denoted $a^{-1}_{i,j}$: $a^{-1}_{i,j}(t)$ is the time at which i must be left to arrive on j at time t. We assume that TD cost functions satisfy the First-In First-Out (FIFO) property introduced by Ichoua et al. (2003). This property ensures that every arrival time function $a_{i,j}$ is non-decreasing, *i.e.*, $\forall t_1, t_2 \in [t_0, l_n], t_1 < t_2 \Rightarrow a_{i,j}(t_1) \leq a_{i,j}(t_2)$. In other words, waiting at i cannot allow one to arrive sooner at j. Without loss of generality, we also assume that cost functions satisfy the triangle inequality, *i.e.*, $\forall i, j, k \in \mathcal{V}, \forall t \in [t_0, l_n], a_{j,k}(a_{i,j}(t)) \leq a_{i,k}(t)$. Indeed, whenever this property is not satisfied, we can enforce cost functions to satisfy it by computing shortest paths in a pre-processing step (this may be done in polynomial-time provided that cost functions satisfy the FIFO property (Kaufman & Smith, 1993)).

The goal of the TD-TSPTW is to minimise the makespan, *i.e.*, the arrival time on n of a path that starts from 0 at time t_0, visits each customer $i \in \mathcal{C}$ once within its TW $[e_i, l_i]$ and ends on n no later than l_n. The objective function is defined more formally in Section 3.3.

3. Literature Review

The TD-TSP has been introduced by Malandraki & Daskin (1992). Since then, different approaches have been proposed to solve this problem (or its variants) and a review may be found in Gendreau et al. (2015). Many approaches are based on metaheuristics such as, for example, Ant Colony Optimization (Donati et al., 2008), Tabu Search (Ichoua et al., 2003), or Large Neighborhood Search (Sun et al., 2020). These approaches provide no guarantee on solution quality. In this section, we first describe exact approaches which ensure finding the optimal solution (given enough time and memory), *i.e.*, CP, ILP, and DP, and then we describe exact and anytime variants of DP and A* which share similarities with our approach.

3.1. Constraint Programming

Melgarejo et al. (2015) have introduced the global constraint *TDNoOverlap* which ensures that a set of tasks is not overlapping when transition times between tasks are time-dependent. This constraint may be used to solve the TD-TSPTW, and it is much more efficient than classical CP models for the TD-TSPTW (based on *allDifferent* constraints), but it is not competitive with state-of-the-art ILP approaches.

3.2. Integer Linear Programming

State-of-the-art exact approaches for the TSP are usually based on ILP Applegate et al. (2011). However, the integration of time within ILP models (to add TW constraints or to exploit TD cost functions, for example) usually strongly degrades performance. Time may be discretized into time steps, but this either dramatically increases the number of variables (when considering fine steps) or reduces solution quality (when considering coarse steps). Boland & Savelsbergh (2019) have introduced Dynamic Discretization Discovery to overcome this issue by dynamically refining time steps to strengthen time-indexed ILP models. This approach is used by Vu et al. (2020) for solving the TD-TSPTW. It performs best on instances with very tight TWs, and it dominates the Branch and Cut approach of Montero et al. (2017). A stronger relaxation was proposed by Vu et al. (2022) for routing problems with variable departure time.

Cordeau et al. (2014) consider the *Ichoua, Gendreau, Potvin (IGP)* model introduced by Ichoua et al. (2003) for computing travel times. In this model, the distance between two vertices is assumed to be constant (*i.e.*, the same sequence of road segments is always used to travel between two vertices), whereas speeds are time-dependent and are defined by piecewise-constant functions: the time horizon $[t_0, l_n]$ is decomposed into H time steps and the travel speed v_{ijh} from i to j at time step $h \in [0, H-1]$ is constant. They propose to decompose v_{ijh} in three factors: $v_{ijh} = u_{ij} b_h \delta_{ijh}$ where u_{ij} is the maximum travel speed from i to j during the whole time horizon (*i.e.*, $u_{ij} = \max_{h\in[0,H-1]} v_{ijh}$), b_h is the best congestion factor over all arcs during time step h (*i.e.*, $b_h = \max_{i,j\in\mathcal{V}} v_{ijh}/u_{ij}$), and δ_{ijh} represents the degradation of the congestion factor of arc (i, j) during time step h (*i.e.*, $\delta_{ijh} = v_{ijh}/(u_{ij} b_h)$). A key parameter is Δ, the smallest value of all δ_{ijh} values (*i.e.*, $\Delta = \min_{i,j\in\mathcal{V},h\in[0,H-1]} \delta_{ijh}$): When $\Delta = 1$, all arcs (i, j) have the same congestion factor b_h for all time steps $h \in [0, H-1]$. In this case, the TD-TSP can be solved as a classical asymmetric TSP with constant travel times. When $\Delta < 1$, the optimal solution computed with $\Delta = 1$

provides a lower bound which is used in the branch-and-bound algorithm of Arigliano et al. (2018a). In Arigliano et al. (2018b), the branching strategy of this algorithm is enhanced with a dominance rule induced by TWs. This approach performs best when all arcs share rather similar congestion patterns, *i.e.*, when Δ is very close to 1.

3.3. Dynamic Programming

The DP approach proposed by Bellman (1962) for the TSP has been extended to handle TD cost functions by Malandraki & Dial (1996) and TWs by Christofides et al. (1981). It has also been extended to *Vehicle Routing Problems* (VRPs) by van Hoorn (2016) and to TD-VRPs by Rifki et al. (2020).

We describe the basic principles of DP for solving the TD-TSPTW as it is a starting point for introducing our approach. Given a vertex $i \in \mathcal{V} \setminus \{0\}$ and a set of vertices $\mathcal{S} \subseteq \mathcal{C} \setminus \{i\}$, let $p(i, \mathcal{S})$ denote the earliest arrival time of a path that starts from 0 at time t_0, visits each vertex of $\mathcal{S}$ exactly once, and ends on i, while satisfying TW constraints of all vertices in $\mathcal{S} \cup \{i\}$ (if no such path exists, then $p(i, \mathcal{S}) = \infty$). We may recursively define $p(i, \mathcal{S})$ as follows:

$$p(i, \mathcal{S}) = \begin{cases} \min_{j \in \mathcal{S}} a_{j,i}(p(j, \mathcal{S} \setminus \{j\}))_{\uparrow[e_i, l_i]} & \text{if } \mathcal{S} \neq \emptyset \quad (1) \\ a_{0,i}(t_0)_{\uparrow[e_i, l_i]} & \text{otherwise} \quad (2) \end{cases}$$

The optimal solution corresponds to $p(n, \mathcal{C})$, *i.e.*, the earliest arrival time on n of a path that starts from 0 at t_0 and visits all vertices of $\mathcal{C}$ during their TWs. It may be computed by searching for a path in a state-transition graph. States are triples $(i, \mathcal{S}, t)$ such that $i \in \mathcal{V} \setminus \{0\}$ is the last visited vertex, $\mathcal{S} \subseteq \mathcal{C} \setminus \{i\}$ is the set of customers that have been visited before i, and $t \in [e_i, l_i]$ is the arrival time on i. A state $(i, \mathcal{S}, t)$ is an initial state whenever $\mathcal{S} = \emptyset$: in this case, i is the first customer visited after the depot 0, and $t = a_{0,i}(t_0)_{\uparrow[e_i, l_i]}$. A state $(i, \mathcal{S}, t)$ is a final state whenever $i = n$ and $\mathcal{S} = \mathcal{C}$: in this case, all customers have been visited and t is the arrival time on the depot n. Edges of the graph correspond to transitions between states. More precisely, for each state $s = (i, \mathcal{S}, t)$:

- if $\mathcal{S} \cup \{i\} \subset \mathcal{C}$ then, for each customer $j \in \mathcal{C} \setminus (\mathcal{S} \cup \{i\})$ such that $a_{i,j}(t) \leq l_j$, there is a transition from s to $(j, \mathcal{S} \cup \{i\}, a_{i,j}(t)_{\uparrow[e_j, l_j]})$;
- if $\mathcal{S} \cup \{i\} = \mathcal{C}$ and $a_{i,n}(t) \leq l_n$, there is a transition from s to the final state $(n, \mathcal{S} \cup \{i\}, a_{i,n}(t))$.

The goal is to find a path from an initial state to a final state $(n, \mathcal{C}, t)$ such that t is minimal. This path may be computed in a level-wise manner, starting from

level 0 that contains all initial states: for each level k ranging from 1 to $n-1$, we compute every state $(j, \mathcal{S}, t)$ such that $\#\mathcal{S} = k$ and t is minimal by exploiting the states $(i, \mathcal{S} \setminus \{i\}, t')$ of level $k-1$ according to Eq. (1).

3.4. Variants of Dynamic Programming

As the number of states explored by DP for the TSP is in $\mathcal{O}(n \cdot 2^n)$, different approaches have been proposed to avoid combinatorial explosion. A first possibility is to consider a *relaxed* state space, where some states are merged into a single one as proposed by Christofides et al. (1981) and Baldacci et al. (2012). In this case, the optimal solution in the relaxed state space provides a lower bound. Lera-Romero et al. (2022) extended this approach to the TDTSP-TW and introduced the *ti-tour* relaxation which outperforms ILP approaches of Arigliano et al. (2018a) and Vu et al. (2020). It is not anytime as it does not yield approximate solutions during search: it either provides the optimal solution (if time or memory limits are not exceeded) or no solution at all.

Another possibility is to use *Restricted* DP (RDP), introduced by Malandraki & Dial (1996), where an upper bound is computed by limiting the number of states stored at each level to the H best ones. RDP is neither exact nor anytime as a single approximate solution is computed at the final layer.

Bergman et al. (2016) have introduced a framework based on Multivalued Decision Diagrams for solving problems that have DP formulations. It is both exact and anytime, *i.e.*, it produces a sequence of solutions of increasing quality until proving optimality (given enough time and memory). It is also generic and it computes bounds without the need of problem-specific implementations, using both RDP and state space relaxations. This approach is improved by Gillard et al. (2021) by computing new bounds (some of them being problem-specific). Results are presented for several problems, including the TSPTW. As far as we know, this kind of approach has never been used to solve the TD-TSPTW.

Hart et al. (1968) have introduced A*, which uses heuristics to speed up the search of shortest paths in state-transition graphs and which is widely used to solve problems that have DP formulations, such as planning problems for example. A* is not anytime: it provides a single optimal solution, and it may have to explore an exponential number of states before finding it. One may convert A* into anytime algorithms such as, for example, anytime weighted and real-time A* algorithms (Hansen & Zhou, 2007; Bulitko & Lee, 2006).

Recently, Libralesso & Fontan (2021) have introduced *Iterative Memory Bounded A** (IMBA*) which allowed them to win the 2018 ROADEF chal-

lenge. The basic idea is to limit the number of stored states with a parameter D, like in RDP. However, in RDP the limit is on the number of states in each level, whereas in IMBA* the limit is on the total number of open states, for all levels. When $D = 1$, the algorithm behaves like a pure greedy one; when $D = \infty$, the algorithm is exact. To obtain an anytime and exact algorithm, IMBA* iterates the process with increasing values of D, according to a geometric progression. Libralesso et al. (2020) have used a very similar approach to obtain state-of-the-art results on the sequential ordering problem.

In this paper, we propose to use *Anytime Column Search* (ACS) which has been introduced by Vadlamudi et al. (2012). ACS is both exact and anytime and, like IMBA*, it iterates A*-like searches. However, instead of bounding the number of stored states, ACS bounds the number of states which are expanded at each level to the w best ones. Vadlamudi et al. (2012) report that, for the TSP, ACS performs best when $w = 1$. Hence, we set w to 1 in this paper.

4. New Approach for the TD-TSPTW

In this section, we first describe ACS and show how to use it to solve the TD-TSPTW (Section 4.1) and how to combine it with TW constraint propagation (Section 4.2) and bounding functions (Section 4.3). Then, we describe a local search algorithm used to improve upper bounds found by ACS (Section 4.4). Finally, we discuss some implementation issues (Section 4.5).

4.1. Anytime Column Search

Our instantiation of ACS to solve the TD-TSPTW is described in Algorithm 1. It searches for paths in the state-transition graph defined in Section 3.3, from initial to final states. As usual in A*-based algorithms, it uses a lower bounding function f to evaluate a state $(i, \mathcal{S}, t)$: $f(i, \mathcal{S}, t)$ is a lower bound of the arrival time of the fastest path that starts from i at time t, visits every vertex of $\mathcal{C} \setminus (\mathcal{S} \cup \{i\})$ within its TW, and ends on n (three bounding functions are described in Section 4.3).

States are created during search: starting from initial states, we iteratively choose an open state and expand it by creating all its successors in the state-transition graph (we say that a state is open whenever it has been created but not expanded). For each level $k \in [0, n-2]$, we maintain a set *open*(k) of open states $(i, \mathcal{S}, t)$ such that $\#\mathcal{S} = k$. Also, we maintain a set *ND* of all created states that are not dominated by another created state, where state $(i, \mathcal{S}, t)$

Algorithm 1: ACS for the TD-TSPTW

```
1  ub ← l_n; ND ← {(i, ∅, a_{0,i}(t_0)_{↑[e_i,l_i]}) : i ∈ C}; open(0) ← ND
2  foreach level k ∈ [1, n − 2] do open(k) ← ∅;
3  while there exists a state s ∈ ∪_{k∈[0,n−2]} open(k) such that useful(s) do
4      foreach k ∈ [0, n − 2] such that there exists s ∈ open(k), useful(s) do
5          repeat
6              remove from open(k) the state s such that f(s) is minimal
7          until useful(s);
8          let s = (i, S, t) be the last state removed from open(k)
9          if k = n − 2 and a_{i,n}(t) < ub then
10             update ub and l_n to a_{i,n}(t) // New solution found
11         else if k < n − 2 then
12             foreach non visited customer j ∈ C \ (S ∪ {i}) do
13                 let s' = (j, S ∪ {i}, a_{i,j}(t)_{↑[e_j,l_j]})
14                 if useful(s') and s' ∉ ND then
15                     remove from ND every state dominated by s'
16                     add s' to ND and to open(k + 1)
```

dominates state $(i, \mathcal{S}, t')$ whenever $t < t'$. Initially, *ND* and *open*(0) contain all initial states whereas *open*(k) is empty for every other level $k > 0$ (lines 1-2).

We know that an open state s can be safely removed from the state-transition graph whenever $f(s) \geq ub$ or s is dominated by a state in *ND*. Hence, *open*(k) should be filtered each time *ub* is decreased or a new state is added to *ND*. As this filtering is expensive, we consider a lazy approach where useless states are removed only when searching for the next state to expand (lines 5-7). We introduce the following predicate to decide whether an open state is still useful or not: $useful(s) \iff (f(s) < ub \wedge \forall s' \in ND, s \text{ is not dominated by } s')$.

At each iteration of the while loop (lines 3-16), we consider each level k ranging from 0 to $n-2$ and we search for the most promising state in *open*(k), *i.e.*, the state $s = (i, \mathcal{S}, t)$ that is useful and that minimizes $f(s)$ (lines 5-7). When $k = n-2$, we have already visited all customers and if the arrival time at n is lower than *ub*, then we have found a new improving solution (line 10). When $k < n-2$, s is expanded in the loop lines 12-16: for each non visited customer j, we compute the new state s' obtained when going from i to j at time t, and if s' is useful, we update *ND* and *open*$(k+1)$ (lines 15-16).

ACS ends when open sets no longer contain useful states. In this case, the last solution found is optimal (see proof in Vadlamudi et al. (2012)). However, as improving solutions are found progressively, one may stop ACS when a given

limit is reached. If there are no TWs, a first solution is found at the end of the first iteration of the loop lines 3-16 and there are $\mathcal{O}(n^2)$ open states: $open(0)$ contains n–1 states, and $\forall k > 0$, $\#open(k) = \#open(k\text{–}1)\text{–}1$. This first solution is the one that would be computed with a greedy algorithm that selects, at each iteration, the state s that minimizes $f(s)$ among all successors of the last selected state. Of course, in presence of TWs it may be necessary to iterate more than once the loop lines 3-16 before finding a solution. However, TW constraints may also be used to prune parts of the search space, as described in Section 4.2.

4.2. Propagation of time window constraints

TW constraints are propagated to infer precedence relations and tighten other TWs. We use the same propagation rules as Vu et al. (2020), which are adaptations to TD cost functions of the rules described by Dash et al. (2012) for the TSPTW. These rules operate on two sets $\mathcal{E}$ and $\mathcal{R}$.

- $\mathcal{E}$ is the set of edges that may be used to travel from a vertex to its successor when building tours: $\mathcal{E}$ is initialized to $\{(0,i),(i,n),(i,j) : i,j \in \mathcal{C} \wedge i \neq j\}$.
- $\mathcal{R}$ is the set of precedence relations: it contains every couple (i,j) such that i must be visited before j in every feasible solution (intermediate nodes may be visited between i and j). $\mathcal{R}$ is initialized to $\{(0,i),(i,n) : i \in \mathcal{C}\}$.

A first set of rules is used to tighten the TW of a vertex k by propagating TWs of its predecessors and successors in $\mathcal{E}$:

- e_k is increased if k can only be reached later than e_k, *i.e.*, $e_k \leftarrow \max\left\{e_k, \min_{(i,k)\in\mathcal{E}} a_{i,k}(e_i)\right\}$;
- e_k is increased if it implies waiting for all its successors, *i.e.*, $e_k \leftarrow \max\left\{e_k, \min_{(k,i)\in\mathcal{E}} a^{-1}_{k,i}(e_i)\right\}$;
- l_k is decreased if it implies arriving too late at each successor of k, *i.e.*, $l_k \leftarrow \min\left\{l_k, \max_{(k,i)\in\mathcal{E}} a^{-1}_{k,i}(l_i)\right\}$;
- l_k is decreased if k is always reached before l_k when leaving its predecessors as late as possible, *i.e.*, $l_k \leftarrow \min\left\{l_k, \max_{(i,k)\in\mathcal{E}} a_{i,k}(l_i)\right\}$.

When TWs are tightened, $\mathcal{R}$ and $\mathcal{E}$ are updated as follows:

- (i,j) is removed from $\mathcal{E}$ and (j,i) is added to $\mathcal{R}$ whenever $a_{i,j}(e_i) > l_j$;
- A subpath $\langle i,j,k\rangle$ is infeasible if $a_{i,j}(e_i) > l_j$ or if $a_{j,k}(\max\{e_j, a_{i,j}(e_i)\}) > l_k$. For all $i,j \in \mathcal{V}$, if there exists k such that both $\langle i,j,k\rangle$ and $\langle k,i,j\rangle$ are infeasible, then (i,j) is removed from $\mathcal{E}$. If $\langle i,k,j\rangle$ is also infeasible, then (j,i) is added to $\mathcal{R}$.

Finally, the last two rules ensure the transitive closure of $\mathcal{R}$ and exploit precedence relations in $\mathcal{R}$ to filter $\mathcal{E}$:

- $\forall i, j, k \in \mathcal{V}$, if $\{(i,j),(j,k)\} \subseteq \mathcal{R}$, then (i,k) is added to $\mathcal{R}$ and removed from $\mathcal{E}$.
- For each $(i,j) \in \mathcal{R}$, arc (j,i) is removed from $\mathcal{E}$.

These rules are applied until reaching a fixed point where no more rule can be applied. This is done at the beginning of Algorithm 1, as a preprocessing step. This is also done after each improvement of l_n (line 10). As far as we know, it is the first time this procedure is used to prune the search space during the search. This is particularly relevant in our context as tighter TWs lead to better relaxations of TD cost functions into constant cost functions, as explained in Section 4.3. In some cases, these rules may either detect an inconsistency (when a vertex in $\mathcal{V} \setminus \{n\}$ has no outgoing edge in $\mathcal{E}$ or when a vertex in $\mathcal{V} \setminus \{0\}$ has no incoming edge in $\mathcal{E}$), or prove optimality (when the TW of n is tightened to a single value, *i.e.*, $e_n = l_n$).

$\mathcal{E}$ and $\mathcal{R}$ are also used to reduce the number of states explored in the loop lines 12-16: we only consider the customers $j \in \mathcal{C} \setminus (\mathcal{S} \cup \{i\})$ such that $(i,j) \in \mathcal{E}$ and, $\forall k \in \mathcal{V} \setminus (\mathcal{S} \cup \{i\})$, $(k,j) \notin \mathcal{R}$.

4.3. Computation of the lower bound f

Given a state $s = (i, \mathcal{S}, t)$, $f(s)$ is a lower bound of the arrival time of the fastest path that starts from i at time t, visits every customer in $\mathcal{C} \setminus (\mathcal{S} \cup \{i\})$, and ends on n, while satisfying all TWs (f may detect that no such path exists and return ∞). It is used by Algorithm 1 to (i) expand first the most promising state of each level, and (ii) prune the state space when a state cannot lead to a solution with a cost smaller than ub. Bounds are widely used in A* and in Branch & Bound approaches, and there exist many different lower bounds for the TSP, which are often computed by solving relaxations. In this section, we describe three lower bounds for the TD-TSPTW, called f_{FEA}, f_{OIA}, and f_{MSA}, which provide different trade-offs between computational cost and tightness. f_{FEA} is a new bound whereas f_{OIA} and f_{MSA} combine f_{FEA} with classical TSP bounds. Before describing bounds, we define constant edge costs and the graph used for computing these bounds.

4.3.1. Definition of constant costs

The cost of edge (j,k) depends on the departure time from j, which is not known exactly when computing $f(s)$. To ensure that $f(s)$ is a lower bound,

we compute a lower bound $\underline{c}_{j,k}$ of the cost of every edge $(j,k) \in \mathcal{E}$. To this end, we first introduce the *Latest Departure Time* (LDT) from a vertex j to reach another vertex k no later than l_k while leaving j no later than l_j, *i.e.*, $LDT(j,k) = \min\{l_j, a^{-1}_{j,k}(l_k)\}$. The lower bound of the cost of edge (j,k) is the shortest travel time from j to k for each departure time $t \in [e_j, LDT(j,k)]$, plus the waiting time on k when the arrival time on k is earlier than e_k, *i.e.*, $\underline{c}_{j,k} = \min_{t\in[e_j,LDT(j,k)]} c_{j,k}(t) + \max\{0, e_k - a_{j,k}(t)\}$. These constant costs are precomputed and updated every time TWs are tightened using rules described in Section 4.2.

4.3.2. Definition of the graph G_s used to compute $f(s)$

Given a state $s = (i, \mathcal{S}, t)$, the lower bound $f(s)$ is computed by solving a relaxation of the shortest Hamiltonian path problem from i to n in a graph $G_s = (\mathcal{V}_s, \mathcal{E}_s)$ such that $\mathcal{V}_s = \{n\} \cup \mathcal{C} \setminus \mathcal{S}$. A straightforward definition of $\mathcal{E}_s$ is $\mathcal{E}_s = \mathcal{E} \cap ((\mathcal{V}_s \setminus \{n\}) \times (\mathcal{V}_s \setminus \{i\}))$ as $\mathcal{E}$ contains edges than may be used in a feasible solution and the path must start from i and end on n. To tighten the lower bound, we introduce three new rules for removing from $\mathcal{E}_s$ edges that cannot be used when the current state is s.

The first two rules are applied on edges that start from i.

- As the path starts from i, we remove any edge (i,k) such that there exists a vertex l that must be visited before k but has not yet been visited: Rule 1 = $\mathcal{E}_s \leftarrow \mathcal{E}_s \setminus \{(i,k) : k \in \mathcal{V}_s \setminus \{i\} \wedge \exists l \in \mathcal{V}_s \setminus \{i\}, (l,k) \in \mathcal{R}\}$.
- As a vertex must be visited before the end of its TW, we remove any edge (i,k) such that we cannot reach k on time when leaving i at time t: Rule 2 = $\mathcal{E}_s \leftarrow \mathcal{E}_s \setminus \{(i,k) : k \in \mathcal{V}_s \setminus \{i\} \wedge t > LDT(i,k)\}$.

A third rule is applied on edges that start from another vertex $j \in \mathcal{V}_s \setminus \{i\}$. For these edges, we know that we first have to travel from i to j and the departure time from j is lower bounded by $a_{i,j}(t)$. However, as this filtering is expensive to perform, we do it once for all possible vertices $j \in \mathcal{V}_s \setminus \{i\}$ and, therefore, we consider a lower bound t' which is valid for all these vertices, *i.e.*, $t' = \min_{(i,j)\in\mathcal{E}_s} a_{i,j}(t)$. Then, we use t' to remove any edge (j,k) such that we cannot reach k on time when leaving j at time t':
Rule 3 = $\mathcal{E}_s \leftarrow \mathcal{E}_s \setminus \{(j,k) : j \in \mathcal{V}_s \setminus \{i,n\} \wedge k \in \mathcal{V}_s \setminus \{i\} \wedge t' > LDT(j,k)\}$.

4.3.3. Feasibility bound f_{FEA}

This bound performs a simple feasibility check on G_s: If any vertex in $\mathcal{V}_s \setminus \{n\}$ (resp. $\mathcal{V}_s \setminus \{i\}$) has no outgoing (resp. incoming) arc in $\mathcal{E}_s$, then s cannot lead

to a feasible solution (as there exists no Hamiltonian path from i to n in G_s). In this case, $f_{\text{FEA}}(i, \mathcal{S}, t) = \infty$. Otherwise, $f_{\text{FEA}}(i, \mathcal{S}, t) = t$.

We also experimented with other feasibility checks, such as ensuring that each node in $\mathcal{V}_s \setminus \{i\}$ is reachable from i (with a linear-time graph traversal), or ensuring that there is a unique topological order among the set of strongly connected components of G_s (when the topological order is not unique, no Hamiltonian path exists). Both these feasibility checks were implemented but discarded as they did not bring enough benefits relatively to their cost.

4.3.4. Outgoing/Incoming Arcs bound f_{OIA}

This bound is an adaptation to the TDTSP-TW of the I/O bound used by Libralesso et al. (2020) for the sequential ordering problem. It is weaker than the assignment relaxation (*i.e.*, the minimum assignment in the bipartite graph between $\mathcal{V}_s \setminus \{n\}$ and $\mathcal{V}_s \setminus \{i\}$). Indeed, it relaxes the constraint that each vertex in $\mathcal{V}_s \setminus \{n\}$ must be connected to a different vertex in $\mathcal{V}_s \setminus \{i\}$. It is also an order cheaper to compute as it is computed in linear time with respect to the number of edges in $\mathcal{E}_s$. As G_s is not necessarily symmetric, we combine two different bounds: f_{OA}, which considers Outgoing Arcs, and f_{IA}, which considers Incoming Arcs. More precisely, f_{OA} adds to t the sum of the minimum-weight outgoing arc for each node in $\mathcal{V}_s \setminus \{n\}$, *i.e.*, $f_{\text{OA}}(i, \mathcal{S}, t) = t + \sum_{j \in \mathcal{V}_s \setminus \{n\}} \min \left\{ \underline{c}_{j,k} : (j,k) \in \mathcal{E}_s \right\}$, whereas f_{IA} considers incoming arcs, *i.e.*, $f_{\text{IA}}(i, \mathcal{S}, t) = t + \sum_{k \in \mathcal{V}_s \setminus \{i\}} \min \left\{ \underline{c}_{j,k} : (j,k) \in \mathcal{E}_s \right\}$.
Finally, we define $f_{\text{OIA}}(i, \mathcal{S}, t) = \max \{ f_{\text{FEA}}(s), f_{\text{OA}}(s), f_{\text{IA}}(s) \}$, which can be computed using a single traversal of edge set $\mathcal{E}_s$.

4.3.5. Minimum Spanning Arborescence bound f_{MSA}

The *Minimum Spanning Arborescence* (MSA) is a classic relaxation of the Asymmetric TSP (Roberti & Toth, 2012). We extend it to the TD-TSPTW using the cost lower bound $\underline{c}$ and the graph G_s. More precisely, given a state $s = (i, \mathcal{S}, t)$, $f_{\text{MSA}}(s)$ is equal to t plus the cost of the MSA rooted at i in G_s. When no such arborescence exists (*i.e.*, there exists at least one node in $\mathcal{V}_s \setminus \{i\}$ that cannot be reached from i) or if a node in $\mathcal{V}_s \setminus \{n\}$ has no outgoing arc, we set $f_{\text{MSA}}(s) = \infty$. The MSA is computed in $O(\#\mathcal{E}_s \log \#\mathcal{V}_s)$ with the algorithm of Gabow et al. (1986).

4.4. Local Search (LS)

In order to converge faster towards good solutions, we combine Algorithm 1 with a LS procedure which tries to improve every solution provided by ACS

(line 10). We use an approach similar to the one of Da Silva & Urrutia (2010) for the TSPTW, based on the following neighborhoods: 1-shift (that moves a single vertex backward or forward in the path), and 2-opt (that reverses a subsequence of vertices of the path). We exploit the two sets $\mathcal{E}$ and $\mathcal{R}$ to reduce the size of the neighborhoods. For each possible candidate move, we update visit times of each vertex impacted by it. If a TW is violated or the travel time exceeds ub, the move is rejected. Otherwise it is accepted and ub is updated. Moves are repeated until reaching a local optimum (*i.e.*, a solution that cannot be improved using a single move).

4.5. Implementation Issues

Our algorithm has been implemented in C++ [1]. In this section, we detail some implementation choices that have a strong impact on time or space.

Data structures. Bitsets are used to efficiently represent the set $\mathcal{S}$ in a state $(i, \mathcal{S}, t)$. A hash table is used to represent *ND*: the key is a couple $(i, \mathcal{S})$, and the value is the smallest time t such that state $(i, \mathcal{S}, t)$ has been created. This allows us to check in amortized constant time if a state already belongs to *ND* or if there is a dominance relation between two states. For each level k, a priority queue is used to represent *open(k)*, where the priority of a state s is defined by $f(s)$. The best state of *open(k)* is found in constant time. Removals and insertions are performed in $\mathcal{O}(\log_2(\#open(k))$. When $k = 0$, $open(k)$ contains $\mathcal{O}(n)$ states and when $k > 1$, $open(k)$ contains $\mathcal{O}(nC_k^n)$ states.

Computation of $c_{i,j}(t)$. The implementation of the function $c_{i,j}(t)$ that returns the travel time from i to j when leaving at time t depends on the considered benchmark. In many benchmarks (such as the ones introduced by Melgarejo et al. (2015) or Rifki et al. (2020)), TD cost functions are provided as piecewise-constant functions: the time horizon is split into h consecutive time-steps and, for each edge (i, j) and each time step s, there is an input value that gives the travel time from i to j when leaving at time-step s. This representation is compact, but it does not necessarily verify the FIFO property. In this case, we use the transformation described by Melgarejo et al. (2015) to ensure the FIFO property. This allows us to compute $c_{i,j}(t)$ in constant time.

In benchmarks that consider the IGP model (described in Section 3), travel times may be computed from distances and speeds in such a way that the FIFO

[1] Source code available at `https://github.com/romainfontaine/tdtsptw-ejor23`

property is ensured (Ichoua et al., 2003). This is done in linear time with respect to the number of time steps involved when traveling from i to j. Similarly to Arigliano et al. (2018b) and Vu et al. (2020), we round times to the nearest integer.

Computation of $a^{-1}_{i,j}(t)$*.* Our implementation assumes that all times have integer values. We use binary search[2] to search for the value $t' = a^{-1}_{i,j}(t)$ such that $a_{i,j}(t') = t$ in $\mathcal{O}(\log_2(l_i - e_i))$. If t' has not an integer value or if there exist more than one value for t' (this occurs when a_{ij} has constant parts), we ensure correctness by returning the largest integer value t' such that $a_{i,j}(t') \leq t$.

Computation of $\mathcal{E}_s$*.* Building the graph G_s to compute $f(s)$ is one of the main bottlenecks of our approach, as filtering arcs according to their LDT (using Rules 2 and 3 of Section 4.3.2) requires $\mathcal{O}(n^2)$ comparisons and memory accesses. To speed up this step, we precompute a set $\mathcal{E}_t = \{(i,j) \in \mathcal{E} : t \leq LDT(i,j)\}$ for each time $t \in RT$ where $RT = \{LDT(i,j) : (i,j) \in \mathcal{E}\}$ is the set of all relevant times. Each set $\mathcal{E}_t$ is encoded with bitsets (for compact storage and fast computation of set intersections), and it is updated when $\mathcal{E}$ is modified or TWs are tightened. Given a state $s = (i, \mathcal{S}, t)$, we compute $\mathcal{E}_s$ as follows: if $t > \max RT$, then $\mathcal{E}_s = \emptyset$ (as t is larger than the LDT of all edges); otherwise, we search for the smallest time $t' \in RT$ such that $t' \geq t$ and we define $\mathcal{E}_s = \mathcal{E} \cap \mathcal{E}_{t'}$. Using bitsets to encode $\mathcal{E}_t$ allows us to efficiently implement f_{FEA}: to check if a vertex has no outgoing or incoming edges, we test bitset emptiness. Hence, although f_{FEA} and f_{OIA} have the same asymptotic complexity, f_{FEA} is much faster in practice.

5. Experimental results on TD benchmarks

In this section, our goal is to (i) evaluate the relevance of the various components of our approach (Section 5.2); (ii) compare our approach with the ILP approaches of Arigliano et al. (2018b) and Vu et al. (2020) and the state-of-the-art DP approach of Lera-Romero et al. (2022) (Sections 5.3 and 5.4); and (iii) evaluate our approach and the one of Lera-Romero et al. (2022) on a realistic benchmark (Section 5.5). Before reporting experimental results, we first describe the experimental setting in Section 5.1.

[2]In the special case of benchmarks based on the IGP model, $a^{-1}_{i,j}(t)$ can be computed using the backward algorithm described in Ichoua et al. (2003).

5.1. Experimental Setting

Considered approaches. We consider three variants of our approach that only differ on the computation of the lower bound f: FEA (resp. OIA and MSA) denotes the variant obtained when $f = f_{\text{FEA}}$ (resp. $f = f_{\text{OIA}}$ and $f = f_{\text{MSA}}$). Our approach is compared with the ILP approaches of Arigliano et al. (2018b) and Vu et al. (2020), respectively denoted Ari18 and Vu20, and with the DP approach of Lera-Romero et al. (2022), denoted Ler22.

Measure of TW tightness. The hardness of a TD-TSPTW instance depends on the number of vertices n and also on TW tightness. To allow us to compare the tightness of instances generated with different models, we compute the percentage of customer pairs that have *Overlapping TWs*, denoted OTW, *i.e.*, $OTW = 100 * \frac{\#\{\{i,j\}\subseteq\mathcal{C}:[e_i,l_i]\cap[e_j,l_j]\neq\emptyset\}}{\#\{\{i,j\}\subseteq\mathcal{C}\}}$.

Benchmark B_{Ari18}. This benchmark has been used by Arigliano et al. (2018b) to evaluate Ari18. It has been randomly generated according to the following parameters: the number of vertices $n \in \{16, 21, 31, 41\}$, the congestion factor $\Delta \in \{.7, .8, .9, .95, .98\}$ (defined in Section 3), the traffic pattern $P \in \{B_1, B_2\}$ and the TW tightness $\beta \in \{0, .25, .50, 1\}$. All TD cost functions contain 73 time-steps and are computed with the IGP model (described in Section 3). There are 30 instances for each combination (n, β, Δ, P), leading to a total of 4800 instances. The smaller β, the wider the TWs: the average OTW for instances with $\beta = 0$ (resp. .25, .50, and 1) is equal to 100 (resp. 90, 67, and 15).

Benchmark B_{Vu20}. This benchmark has been used by Vu et al. (2020) to evaluate Vu20. It is an extension of B_{Ari18}, where the number of vertices has been increased to $n \in \{60, 80, 100\}$, using a similar model to generate TD cost functions except that only four values of Δ are considered, *i.e.*, $\Delta \in \{.7, .8, .9, .98\}$. TWs have been generated differently: instead of using a parameter β, there is a parameter $w \in \{40, 60, 80, 100, 120, 150\}$ that determines the TW width (which is the same for all customers of the instance). There are 10 instances for each combination of (n, w, Δ), leading to a total of 720 instances. TWs of most instances of this benchmark are much tighter than those of B_{Ari18}: the average OTW over the 120 instances with $w = 40$ (resp. 60, 80, 100, 120, and 150) is equal to 5 (resp. 8, 11, 14, 16, and 20).

Benchmark B_{Rif20}. B_{Ari18} and B_{Vu20} have been randomly generated according to a rather simple model: customers are randomly distributed in three concentric

circular zones which are used to define TD travel speeds, and the distance between two points is constant. This is not very realistic as in real urban contexts the fastest path between two customers may change depending on the departure time. In order to evaluate our approach on more realistic TD cost functions, we consider the benchmark described by Rifki et al. (2020), denoted B_{Rif20}. TD cost functions of this benchmark were generated by computing shortest paths in the road network of Lyon for all possible departure times, using a realistic traffic simulation built from real-world data. Different TD cost functions are provided, depending on two parameters σ and l that define the spatial and temporal granularity of traffic data. We report results with $\sigma = 100$ and $l = 6$ which are the finest possible values (similar results were obtained with other values). In this case, TD cost functions are piecewise-constant functions composed of 120 time-steps. To ease the comparison with results obtained on B_{Ari18}, we consider instances with $n \in \{21, 31, 41\}$ and TWs were generated using the same model, *i.e.*, TW tightness is controlled by $\beta \in \{0, .25, .50, 1\}$. There are 150 instances for each combination of (n, β), leading to a total of 1800 instances. The average OTW over the 450 instances with $\beta = 0$ (resp. .25, .50, and 1) is equal to 100 (resp. 86, 62, and 16).

Considered hardware. Ler22, fea, oia, and msa are run on 2.1GHz Intel Xeon E5-2620 v4 processors with 64GB RAM. To favor reproducibility, experiments were executed on Grid5000 (Balouek et al., 2013). As suggested by Fichte et al. (2021), *Turbo Boost* was disabled and each machine solved one instance at a time, using a single processor. Run times of Ari18 and Vu20 are those reported by Arigliano et al. (2018b) and Vu et al. (2020), as source codes are not available: Ari18 is run on a 2.33GHz Intel Core 2 Duo processor with 4GB RAM and Vu20 on a 3.4GHz Intel Core i7-2600 processor (unknown RAM).

Performance measures. We say that an instance is solved by an approach whenever it finds the optimal solution and proves its optimality within one hour. $\#s$ denotes the number of solved instances, and t_s the average solving time for the solved instances. $\#r$ denotes the number of instances for which the approach has found the reference solution, and t_r denotes the average time needed to find the reference solution for these instances. The reference solution is either the optimal solution, when Ler22 or at least one of our approaches has solved the instance, or the best solution obtained by running oia and msa with an extended time limit of 3 hours. The reference solution is optimal for all instances

of B_{VU20} and for all instances of B_{ARI18} and B_{RIF20} such that $n \leq 31$ or $\beta \geq .50$. When $n = 41$, the percentage of instances for which the reference solution is known to be optimal is equal to 42% (resp. 96%) for B_{ARI18} when $\beta = 0$ (resp. $\beta = .25$), and to 9% (resp. 100%) for B_{RIF20} when $\beta = 0$ (resp. $\beta = .25$).

When displaying performance measures of different approaches, we underline the maximal value of $\#s$ or $\#r$ and we highlight in blue (resp. green) the smallest value of t_s (resp. t_r) among all approaches that maximize $\#s$ (resp. $\#r$). For ARI18 and VU20, we do not report $\#r$ and t_r as they are not available. For LER22, we do not report $\#r$ and t_r as they are equal to $\#s$ and t_s, given this approach is not anytime.

5.2. Analysis of the algorithm's components

ACS is combined with three key components, *i.e.*, TW constraint propagation (described in Section 4.2), LS (described in Section 4.4), and rules that exploit LDTs to filter the set $\mathcal{E}_s$ of edges used to compute f (described in Section 4.3.2). To evaluate the relevance of these components, we report results obtained with different variants obtained by disabling them. We consider $f = f_{\text{OIA}}$ as similar conclusions are observed with f_{FEA} and f_{MSA}. We consider the following variants:

- OIA_0 is the variant where the three components are disabled;
- OIA_1 is obtained from OIA_0 by enabling TW constraint propagation before starting the search, during a preprocessing step;
- OIA_2 is obtained from OIA_1 by also enabling TW constraint propagation during the search, each time ub is decreased;
- OIA_3 is obtained from OIA_2 by enabling LS;
- OIA is obtained from OIA_3 by enabling the filtering of $\mathcal{E}_s$.

In Table 1, we display performance measures of these variants on a representative subset of 180 instances with $n = 31$ and $\beta \in \{0, 0.25, 0.50\}$ coming from B_{ARI18} (similar results are obtained with other benchmarks). When looking at the number of solved instances (left side of Table 1), we see that all components but LS improve performance: OIA_1, which propagates TW constraints before the search, solves 69 more instances than OIA_0; OIA_2, which also propagates TW constraints during the search, solves three more instances; and the filtering of $\mathcal{E}_s$ (OIA) allows us to solve 62 more instances. To compare solving time distributions, we display in the top part of Figure 1 the evolution of the percentage

	Solved instances										Reference solutions									
	OIA$_0$		OIA$_1$		OIA$_2$		OIA$_3$		OIA		OIA$_0$		OIA$_1$		OIA$_2$		OIA$_3$		OIA	
β	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r
0	0	-	1	3213	1	3065	1	3100	59	819	29	907	54	662	53	674	54	652	60	18
.25	0	-	53	1487	56	1333	56	1333	60	220	58	91	60	13	60	13	60	12	60	3
.50	45	1612	60	26	60	18	60	18	60	6	60	1	60	0	60	1	60	0	60	0
Total	45		114		117		117		179		147		174		173		174		180	

Table 1: Performance of OIA variants on a subset of B_{ARI18} instances with $n = 31$ (60 instances per value of β). Left: Number of solved instances ($\#s$) and solving time (t_s). Right: Number of reference solutions found ($\#r$) and time to reference solution (t_r).

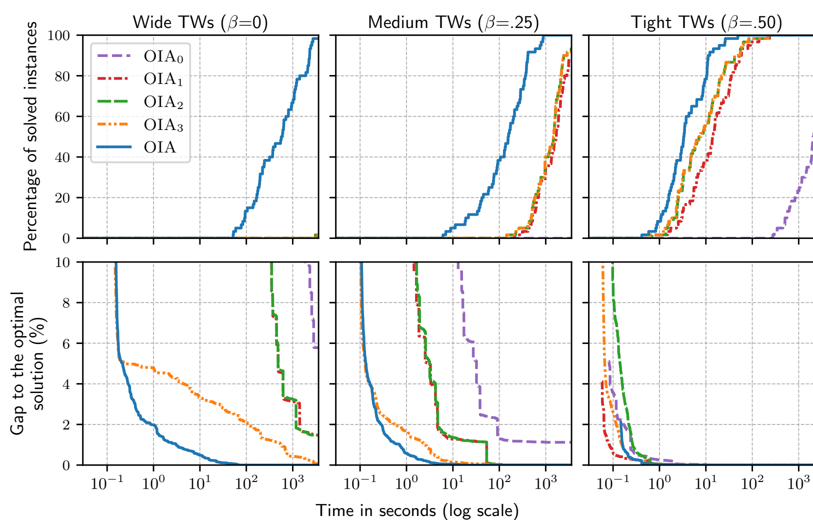

Figure 1: Comparison of OIA variants on 60 instances of B_{ARI18} with n=31 and $\beta \in \{0, .25, .50\}$. Top: Evolution of the percentage of solved instances with respect to time. Bottom: Evolution of the average gap to the reference solution (in percentage) with respect to time.

of solved instances with respect to time. It shows us that similar improvements are observed for time limits shorter than one hour.

When looking at the number of reference solutions found (right side of Table 1), we see that the propagation of TW constraints during the search slightly degrades performance (OIA$_2$ finds one less reference solution than OIA$_1$): this step is rather time consuming and becomes interesting only when the optimal solution has been found, to shorten the time spent to prove optimality. We also see that LS allows OIA$_3$ to find reference solutions quicker. To compare the ability of our different variants to quickly converge towards good solutions, we display in the bottom part of Figure 1 the evolution of the gap to the reference

		Solved instances										Reference solutions					
		Ari18		Ler22		FEA		OIA		MSA		FEA		OIA		MSA	
n	β	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r
16	0	287	299	300	5	300	0	300	0	300	0	300	0	300	0	300	0
	.25	299	143	300	3	300	0	300	0	300	0	300	0	300	0	300	0
	.50	299	26	300	2	300	0	300	0	300	0	300	0	300	0	300	0
	1	300	2	300	0	300	0	300	0	300	0	300	0	300	0	300	0
21	0	248	660	300	198	300	1	300	1	300	3	300	0	300	0	300	1
	.25	286	383	300	87	300	0	300	0	300	1	300	0	300	0	300	0
	.50	296	289	300	19	300	0	300	0	300	0	300	0	300	0	300	0
	1	300	29	300	0	300	0	300	0	300	0	300	0	300	0	300	0
31	0	155	1631	300	1788	300	496	294	808	235	1334	300	64	300	20	300	67
	.25	199	1274	300	1084	300	145	300	219	299	637	300	19	300	5	300	12
	.50	157	1433	300	389	300	5	300	6	300	16	300	1	300	0	300	1
	1	233	608	300	0	300	0	300	0	300	0	300	0	300	0	300	0
41	0	110	2263	126	2778	0	-	0	-	0	-	160	450	234	567	200	700
	.25	131	1950	244	2593	35	2444	16	2986	0	-	209	315	280	207	265	282
	.50	55	2276	300	1837	252	566	280	645	235	1069	299	75	300	6	300	19
	1	106	528	300	0	300	0	300	0	300	0	300	0	300	0	300	0
Total		3461		4570		4187		4190		4069		4568		4714		4665	

Table 2: Performance of Ari18, Ler22, FEA, OIA, and MSA on B_{Ari18} (300 instances per row).

solution with respect to time. It demonstrates that LS allows OIA_3 to find better solutions than OIA_2 at the beginning of the search, especially for wide TWs.

Finally, let us mention that all components but LS significantly reduce memory use. For example, when $\beta = .25$, OIA_0 (resp. OIA_1, OIA_2, OIA_3, and OIA) used on average 63.8 (resp. 19.1, 15.6, 15.6, and 1.9) GB of memory.

5.3. Experimental Comparison on Arigliano et al. (2018b)'s benchmark

Let us now compare our approach with Ari18 and Ler22 on benchmark B_{Ari18}. In Table 2, we report the number of solved instances and solving times. MSA is always outperformed by both OIA and FEA, showing that a tighter (but more expensive) bound does not pay off on this benchmark. FEA and OIA have very close performance when $n \leq 21$. When $n = 31$, FEA solves all instances and outperforms OIA but when $n = 41$, OIA performs better than FEA, showing that a tighter bound pays off when considering larger instances. Similar conclusions are drawn from the right side of Table 2, which considers the ability to quickly find reference solutions: OIA outperforms both FEA and MSA.

Ler22 manages to solve more instances than our approach when $n = 41$ and $\beta \leq .50$. However, when $n = 31$ and $\beta \leq .50$, FEA is faster than Ler22. Also, unlike Ler22, our approach is anytime: when our approach has not solved

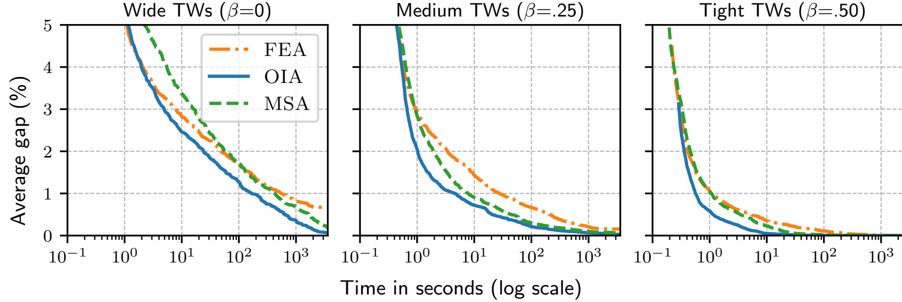

Figure 2: Evolution of the average gap (%) to the reference solution with respect to time for FEA, OIA and MSA on the instances with $n = 41$ and $\beta \in \{0, .25, .50\}$ (300 instances per class).

an instance, it has found approximate solutions which are often optimal. To evaluate the ability of our approach to quickly converge towards good solutions, we display the evolution of the gap to reference solutions with respect to time for the hardest instance classes (*i.e.*, $n = 41$ and $\beta \leq .50$) in Figure 2. It shows us that OIA converges faster than FEA and MSA and that it reaches an average gap to the reference solution of 1% in 168s (resp. 5s and .6s) when $\beta = 0$ (resp. $\beta = .25$ and $\beta = .50$). As a comparison, LER22 either obtains the optimal solution, or no solution at all. When $n = 41$ and $\beta = 0$ (resp. $\beta = .25$ and $\beta = .50$), LER22 has found 126 (resp. 244 and 300) optimal solutions in an average time of 2778s (resp. 2593s and 1837s). Regarding memory use, LER22, FEA, OIA and MSA respectively used 6, 35, 25 and 7 GB on average, when $n{=}41$. This demonstrates that using tighter bounds reduces memory needs.

Table 2 also presents results for ARI18. Even if it has been run on a different computer, we can see that our approach is more successful on many classes: OIA solves 729 more instances than ARI18 on the full benchmark and, on a large number of classes the difference in solving times cannot only come from the fact that they have been run on different computers. However, when $n = 41$ and $\beta \in \{0, 0.25\}$, only 35 (resp. 16) instances are solved by FEA (resp. OIA) whereas ARI18 is able to solve 241 instances.

The success of ARI18 is strongly related to Δ as it relies on bounds which are tighter when Δ is closer to 1, as explained in Section 3. To illustrate this, we detail in Table 3 the number of solved instances for each value of Δ and each traffic pattern P when $n = 41$. It shows us that ARI18 is very sensitive to Δ and P, whereas our approach is mainly sensitive to the TW width β. Note that B_{ARI18} has been randomly generated according to a model which allows one to

	Ari18											FEA										
	$P = B_1$					$P = B_2$						$P = B_1$					$P = B_2$					
$\beta \backslash \Delta$	.70	.80	.90	.95	.98	.70	.80	.90	.95	.98	Total	.70	.80	.90	.95	.98	.70	.80	.90	.95	.98	Total
0	6	8	10	19	28	1	0	3	12	23	110	0	0	0	0	0	0	0	0	0	0	0
.25	6	8	13	23	29	1	0	5	16	30	131	2	2	3	4	4	4	4	4	4	4	35
.50	1	2	4	9	18	1	0	2	5	13	55	24	25	25	25	25	28	25	25	25	25	252
1	14	11	14	12	29	8	4	3	4	7	106	30	30	30	30	30	30	30	30	30	30	300
Total	27	29	41	63	104	11	4	13	37	73		56	57	58	59	59	62	59	59	59	59	

Table 3: Number of instances solved by Ari18 and fea with respect to P, Δ and β for $n = 41$ (30 instances per class).

control Δ. In benchmarks generated from real-world data such as the one of Rifki et al. (2020), for example, the value of Δ is not controlled and it is much lower than 0.7 (see Section 5.5).

5.4. Experimental Comparison on Vu et al. (2020)'s benchmark

Let us now compare our approach with Ler22 and Vu20[3] on benchmark B_{Vu20} which has larger numbers of customers to visit and very tight TWs. In Table 4, we report the number of solved instances and solving times. msa is always outperformed by oia which is always outperformed by fea. This comes from the fact that TWs are very tight: in this case, the propagation of TW constraints and the filtering of arcs based on LDTs remove many edges of $\mathcal{E}$ and the simple feasibility check of f_{FEA} is often enough to detect inconsistencies.

If fea is able to solve all instances, Ler22 and Vu20 respectively fail at solving two and 19 instances. fea is almost always more than ten times as fast as Ler22 and Vu20 and, for some classes it is more than 100 times as fast. This difference is large enough to allow us to conclude that fea is more efficient than Ler22 and Vu20 (even though the latter was run on a different computer).

The right part of Table 4 also shows us that fea always finds the reference solution very quickly, in a few tenths of a second for all classes except when $n = 100$ and $w = 150$, where 3.6 seconds are needed to find it, on average.

Regarding memory, Ler22 used on average 0.8 GB for instances where $n = 100$, whereas fea, oia and msa all used 0.2 GB. This can be explained by the fact that our bounds prune the search space efficiently because of TW tightness.

[3]Results of Vu20 have been sent to us by authors in a personal communication.

		Solved instances										Reference solutions					
		LER22		VU20		FEA		OIA		MSA		FEA		OIA		MSA	
n	w	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r
60	≤ 80	120	1.0	120	3.3	120	0.1	120	0.1	120	0.1	120	0.0	120	0.0	120	0.0
	100	40	8.0	40	15.5	40	0.1	40	0.1	40	0.1	40	0.0	40	0.1	40	0.0
	120	40	25.9	40	84.8	40	0.1	40	0.1	40	0.2	40	0.1	40	0.1	40	0.1
	150	40	154.7	39	219.6	40	0.2	40	0.4	40	1.3	40	0.1	40	0.1	40	0.1
80	≤ 80	120	8.4	120	65.4	120	0.2	120	0.2	120	0.2	120	0.1	120	0.1	120	0.1
	100	40	52.7	39	198.3	40	0.2	40	0.3	40	0.7	40	0.1	40	0.1	40	0.1
	120	40	96.6	37	433.3	40	0.4	40	0.8	40	2.3	40	0.2	40	0.2	40	0.2
	150	40	193.2	39	629.4	40	1.5	40	4.3	40	14.3	40	0.2	40	0.2	40	0.3
100	≤ 80	120	58.2	120	59.4	120	0.4	120	0.5	120	1.2	120	0.2	120	0.2	120	0.2
	100	40	219.0	39	292.5	40	1.3	40	3.4	40	11.7	40	0.3	40	0.3	40	0.4
	120	40	365.9	39	435.8	40	5.0	40	16.2	40	55.9	40	0.3	40	0.4	40	0.5
	150	38	722.5	29	1291.1	40	79.4	39	165.1	39	564.6	40	3.6	40	9.5	40	33.9
Total		718		701		720		719		719		720		720		720	

Table 4: Performance of LER22, VU20, FEA, OIA, and MSA on B_{VU20} (40 instances per row when $w \in \{100, 120, 150\}$, and 120 instances per row when $w \leq 80$)

5.5. Experimental Comparison on Rifki et al. (2020)'s benchmark

We cannot report results of ARI18 or VU20 on B_{RIF20} as source codes of these approaches are not available. However, TD cost functions of B_{RIF20} have been generated by computing shortest paths using a realistic traffic simulation. In this case, Δ cannot be controlled and it is much smaller than in B_{ARI18} and B_{VU20}: in B_{RIF20}, Δ is always smaller than 0.35, and it has an average value of 0.09. As ARI18's performance drops when $\Delta < 0.9$ (as illustrated in Table 3), we may assume that ARI18 should have difficulties in solving these instances.

In Table 5, we report performance measures of LER22 and our approach on this benchmark. The results of our approach are quite similar to those obtained on B_{ARI18} (see Table 2). In other words, changing the benchmark does not significantly changes the performance of our approach, and OIA still offers the best compromise between bound tightness and computational cost.

On the contrary, LER22's performance is worse on this benchmark than on B_{ARI18}. Table 5 shows that it failed to solve two instances when $n = 21$ and $\beta = 0$, and solved only 7% of instance class $n = 41$ and $\beta = 0$ whereas it solved 42% of them on B_{ARI18}. These differences may stem from the fact that the TD travel times functions of this benchmark vary more often (in B_{ARI18} and B_{RIF20}, they respectively contain 73 and 120 timesteps).

		Solved instances								Reference solutions					
		Ler22		FEA		OIA		MSA		FEA		OIA		MSA	
n	β	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r
21	0	148	363	150	0	150	1	150	2	150	0	150	0	150	0
	.25	150	89	150	0	150	0	150	0	150	0	150	0	150	0
	.50	150	15	150	0	150	0	150	0	150	0	150	0	150	0
	1	150	0	150	0	150	0	150	0	150	0	150	0	150	0
31	0	149	2176	149	397	148	488	136	1151	150	67	150	19	150	67
	.25	150	1503	150	84	150	68	149	152	150	11	150	1	150	3
	.50	150	431	150	1	150	1	150	3	150	0	150	0	150	0
	1	150	0	150	0	150	0	150	0	150	0	150	0	150	0
41	0	11	2902	0	-	0	-	0	-	69	414	138	448	117	622
	.25	132	2744	12	2236	27	1950	15	1800	120	413	149	51	147	199
	.50	149	1450	150	40	150	35	150	120	150	1	150	1	150	3
	1	150	0	150	0	150	0	150	0	150	0	150	0	150	0
Total		1639		1511		1525		1500		1689		1787		1764	

Table 5: Performance of Ler22, fea, oia, and msa on B_{Rif20} (150 instances per row).

Reference	Name	#inst	n		OTW		S
			Min	Max	Min	Max	
Ascheuer (1996)	Asc	50	11	232	5.3	100.0	
Da Silva & Urrutia (2010)	DaS	125	201	401	0.2	4.5	✓
Dumas et al. (1995)	Dum	135	21	201	3.9	58.9	✓
Gendreau et al. (1998)	Gen	130	21	101	21.3	88.9	✓
Langevin et al. (1993)	Lan	70	20	60	2.0	12.9	✓
Ohlmann & Thomas (2007)	Ohl	25	151	201	24.2	37.2	✓
Pesant et al. (1998)	Pes	27	20	45	24.1	100.0	
Potvin & Bengio (1996)	Pot	30	4	46	23.3	100.0	

Table 6: Description of TSPTW benchmarks. Each line displays: a reference that describes the benchmark, the name used to refer to this benchmark, the number of instances in the benchmark, the minimum and maximum number of vertices n, and the minimum and maximum value of OTW. Column S contains ✓ whenever cost functions are symmetrical.

6. Experimental evaluation on the TSPTW

In this section, we experimentally evaluate our approach for solving TSPTW instances. Our approach is adapted to use constant cost functions in a straightforward way, by setting $\underline{c}_{i,j} = \max(l_i + c_{i,j}, e_j) - l_i$, and replacing $a^{-1}_{j,k}(t)$ by $t - c_{j,k}$. We use the same set of benchmarks as in Gillard et al. (2021), plus the benchmark introduced in Da Silva & Urrutia (2010), leading to a total of 592 instances. The main features of these benchmarks are described in Table 6.

We compare our approach with the exact and anytime approach of Gillard et al. (2021), denoted Gil21: it is based on DP and relies on state space relaxations to compute lower bounds and on RDP to compute upper bounds, as

	Solved instances								Reference solutions									
	Gil21		FEA		OIA		MSA		Gil21		DaS10		FEA		OIA		MSA	
	#s	t_s	#s	t_s	#s	t_s	#s	t_s	#r	t_r	#r	t_r	#r	t_r	#r	t_r	#r	t_r
Asc	22	384	50	18	49	1	49	2	47	56	-	-	50	0	50	0	50	0
DaS	110	6	125	4	125	4	125	4	125	7	124	17	125	2	125	2	125	3
Dum	109	188	135	0	135	0	135	0	135	4	135	0	135	0	135	0	135	0
Gen	27	522	117	84	113	44	111	54	129	21	98	154	130	0	130	1	130	0
Lan	70	0	70	0	70	0	70	0	70	0	70	0	70	0	70	0	70	0
Ohl	0	-	20	41	20	61	20	7	25	271	18	520	25	32	25	116	25	42
Pes	8	118	25	152	26	13	27	66	23	143	-	-	27	6	27	4	27	1
Pot	15	247	27	14	28	42	29	83	25	52	-	-	30	4	30	1	30	9
Tot.	361		569		566		566		579		-		592		592		592	

Table 7: Performance of Gil21, fea, oia, and msa on TSPTW benchmarks.

explained in Section 3.3. We also compare our approach with the LS-based approach of Da Silva & Urrutia (2010), denoted DaS10. As DaS10 only considers symmetrical instances, we do not report results of DaS10 for Asc, Pes and Pot instances. As DaS10 assumes that triangle inequality is satisfied, we have preprocessed all symmetrical instances to ensure it. As DaS10 is not deterministic, it was run five times and we report the median value.

We consider the same experimental setting as in Section 5, and both Gil21 and DaS10 were executed on the same hardware as our approach. For asymmetrical instances, reference solutions come from `https://lopez-ibanez.eu/tsptw-instances`. For symmetrical instances, ensuring triangle inequality may change the optimal solution (as some costs are decreased) and, as in the previous section, we have computed reference solutions by running oia and msa with a time limit of 3 hours. The reference solution has been proven optimal for all but 18 symmetrical instances (*i.e.*, 4% of them). Reference solution costs never exceed those listed at `https://lopez-ibanez.eu/tsptw-instances`.

Table 7 reports performance of the considered approaches. On the whole set of 592 instances, fea solves three more instances than oia and msa. However, on two benchmarks with wide TWs (*i.e.*, Pes and Pot), msa solves more instances than fea. The three variants of our approach solve more instances than Gil21 for all benchmarks except Lan (these instances are solved in less than one second by all approaches).

On the left part of Figure 3, we display the evolution of the percentage of solved instances with respect to time, showing that fea is more successful than Gil21 for time limits shorter than one hour, except for execution times smaller than 8 milliseconds (for clarity, we do not display results of oia and msa as

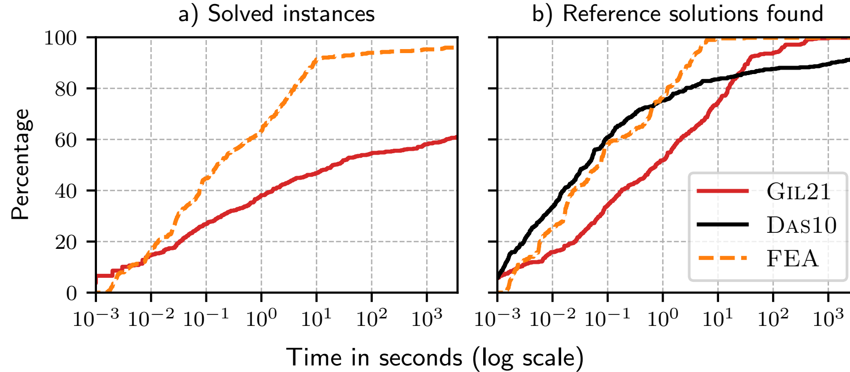

Figure 3: Left: Evolution of the percentage of solved instances by FEA and Gil21 with respect to time, for the full set of 592 TSPTW instances. Right: Evolution of the percentage of reference solutions found by FEA, Gil21, and Das10 for the 485 symmetrical instances.

they are very close to FEA's and FEA is slightly better).

If some instances are not solved by our approach within one hour, reference solutions are always found rather quickly for all instances, whereas Gil21 is not able to find them for 13 instances. Das10 also fails at finding them for 40 instances (*i.e.*, 8% of the 485 symmetrical instances).

On the right part of Figure 3, we display the evolution of the percentage of reference solutions found with respect to time (when considering only the 485 symmetrical instances). Das10 finds more reference solutions for time limits shorter than one second, but it is outperformed by FEA for longer time limits, and also by Gil21 for time limits longer than 23 seconds. This shows us that exact approaches find reference solutions rather steadily, while the heuristic approach Das10 quickly finds reference solutions to easy instances, but struggles for the harder ones (very few reference solutions are found after 100s). Also, Gil21 finds more reference solutions than FEA for very short time limits, smaller than two milliseconds. This may come from the fact that FEA spends time propagating TW constraints. However, for longer time limits, FEA finds more reference solutions and it is able to find all reference solutions of symmetrical instances whereas Gil21 fails at finding one reference solution.

Finally, let us note that our approach requires less memory than Gil21, but more than Das10: on average, FEA (resp. OIA, MSA, Gil21, and Das10) used 2.3 (resp. 0.5, 0.2, 6.7, and $5 * 10^{-3}$) GBs of memory.

7. Conclusions and perspectives

We have introduced a new approach for the TD-TSPTW which combines ACS, TW constraint propagation, and LS. This approach is both able to quickly find good solutions and to prove optimality given enough time and memory. We have considered three bounds with different tightness/cost trade-offs and experiments have shown us that f_{OIA} offers a good compromise. We also proposed new filtering rules based on latest departure times to compute tighter bounds. Our approach is able to find reference solutions much faster than Ler22, the state-of-the-art DP approach of Lera-Romero et al. (2022). It also manages to prove optimality, and does so faster than Ler22 when TWs are tight, and slower otherwise. Our approach also outperforms the ILP approach of Vu et al. (2020) on all instances of B_{Vu20} which have very tight TWs, as well as the ILP approach of Arigliano et al. (2018b) on most instances of B_{Ari18}. Our approach may also be used to solve the TSPTW and we have shown that it outperforms the DP-based approach of Gillard et al. (2021) and the LS-based approach of Da Silva & Urrutia (2010).

We plan to extend our approach to other TD routing problems such as, for example, TD vehicle routing problems (Chen et al., 2006), TD orienteering problems (Khodadadian et al., 2022), TD inventory routing problems (Touzout et al., 2022) or TD profitable pickup and delivery problems (Sun et al., 2020). Our approach could also be extended to scheduling problems with transition times between tasks, as they are very close to TSP problems and often have DP formulations (van Hoorn, 2016). In some cases, these transition times appear to be TD such as, for example, agile earth observation satellite scheduling problems with TD transition times and TWs (Liu et al., 2017), or order acceptance and scheduling problems with processing times (He et al., 2019).

Acknowledgements. We thank Gillard *et al.* and Lera-Romero *et al.* for helping us to reproduce their results, as well as Vu *et al.* for sharing their benchmark and results.

References

Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2011). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.

Arigliano, A., Calogiuri, T., Ghiani, G., & Guerriero, E. (2018a). A branch-and-bound algorithm for the TD-TSP. *Networks*, *72*, 382–392.

Arigliano, A., Ghiani, G., Grieco, A., Guerriero, E., & Plana, I. (2018b). Time-dependent asymmetric traveling salesman problem with time windows: Properties and an exact algorithm. *Discrete Applied Mathematics*, *261*, 28–39.

Ascheuer, N. (1996). *Hamiltonian path problems in the on-line optimization of flexible manufacturing systems*. Ph.D. thesis Zuse Institute Berlin.

Baldacci, R., Mingozzi, A., & Roberti, R. (2012). New State-Space Relaxations for Solving the Traveling Salesman Problem with Time Windows. *INFORMS Journal on Computing*, *24*, 356–371.

Balouek, D., Amarie, A. C., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Perez, C., Quesnel, F., Rohr, C., & Sarzyniec, L. (2013). Adding Virtualization Capabilities to the Grid'5000 Testbed. In *Cloud Computing and Services Science* (pp. 3–20). Springer Verlag volume 367 of *Communications in Computer and Information Science*.

Bellman, R. (1962). Dynamic Programming Treatment of the Travelling Salesman Problem. *Journal of the ACM (JACM)*, *9*, 61–63.

Bergman, D., Ciré, A. A., van Hoeve, W. J., & Hooker, J. (2016). *Decision Diagrams for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer.

Boland, N. L., & Savelsbergh, M. W. (2019). Perspectives on integer programming for time-dependent models. *TOP*, *27*, 147–173.

Bulitko, V., & Lee, G. (2006). Learning in real-time search: A unifying framework. *Journal of Artificial Intelligence Research*, *25*, 119–157.

Chen, H. K., Hsueh, C. F., & Chang, M. S. (2006). The real-time time-dependent vehicle routing problem. *Transportation Research Part E: Logistics and Transportation Review*, *42*, 383–408.

Christofides, N., Mingozzi, A., & Toth, P. (1981). State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, *11*, 145–164.

Cordeau, J. F., Ghiani, G., & Guerriero, E. (2014). Analysis and branch-and-cut algorithm for the time-dependent travelling salesman problem. *Transportation Science*, *48*, 46–58.

Da Silva, R. F., & Urrutia, S. (2010). A General VNS heuristic for the traveling salesman problem with time windows. *Discrete Optimization*, *7*, 203–211.

Dash, S., Günlük, O., Lodi, A., & Tramontani, A. (2012). A Time Bucket Formulation for the Traveling Salesman Problem with Time Windows. *INFORMS Journal on Computing*, *24*, 132–147.

Donati, A., Montemanni, R., Casagrande, N., Rizzoli, A., & Gambardella, L. (2008). Time dependent vehicle routing problem with a multi ant colony system. *European Journal of Operational Research*, *185*, 1174–1191.

Dumas, Y., Desrosiers, J., Gelinas, E., & Solomon, M. (1995). An Optimal Algorithm for the Traveling Salesman Problem with Time Windows. *Operations Research*, *43*, 367–371.

Fichte, J. K., Hecher, M., McCreesh, C., & Shahab, A. (2021). Complications for Computational Experiments from Modern Processors. *27th Conference on Principles and Practice of Constraint Programming*, *210*, 25:1–25:21.

Gabow, H. N., Galil, Z., Spencer, T., & Tarjan, R. E. (1986). Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, *6*, 109–122.

Gendreau, M., Ghiani, G., & Guerriero, E. (2015). Time-dependent routing problems: A review. *Computers and Operations Research*, *64*, 189–197.

Gendreau, M., Hertz, A., Laporte, G., & Stan, M. (1998). A Generalized Insertion Heuristic for the Traveling Salesman Problem with Time Windows. *Operations Research*, *46*, 330–335.

Gillard, X., Coppé, V., Schaus, P., & Cire, A. (2021). Improving the Filtering of Branch-and-Bound MDD Solver. *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, *LNCS 12735*, 231–247.

Hansen, E. A., & Zhou, R. (2007). Anytime heuristic search. *Journal of Artificial Intelligence Research*, *28*, 267–297.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE transactions on Systems Science and Cybernetics*, *4*, 100–107.

He, L., Guijt, A., de Weerdt, M., Xing, L., & Yorke-Smith, N. (2019). Order acceptance and scheduling with sequence-dependent setup times: A new memetic algorithm and benchmark of the state of the art. *Computers and Industrial Engineering*, *138*, 106102.

van Hoorn, J. J. (2016). *Dynamic Programming for Routing and Scheduling: Optimizing Sequences of Decisions*. Ph.D. thesis Vrije Universiteit.

Ichoua, S., Gendreau, M., & Potvin, J. Y. (2003). Vehicle dispatching with time-dependent travel times. *European Journal of Operational Research*, *144*, 379–396.

Kaufman, D. E., & Smith, R. L. (1993). Fastest paths in time-dependent networks for intelligent vehicle-highway systems application. *Journal of Intelligent Transportation Systems*, *1*, 1–11.

Khodadadian, M., Divsalar, A., Verbeeck, C., Gunawan, A., & Vansteenwegen, P. (2022). Time dependent orienteering problem with time windows and service time dependent profits. *Computers and Operations Research*, *143*, 105794.

Langevin, A., Desrochers, M., Desrosiers, J., Gélinas, S., & Soumis, F. (1993). A two-commodity flow formulation for the traveling salesman and the makespan problems with time windows. *Networks*, *23*, 631–640.

Lera-Romero, G., Miranda Bront, J.-J., & Soulignac, F. J. (2022). Dynamic Programming for the Time-Dependent Traveling Salesman Problem with Time Windows. *INFORMS Journal on Computing*, *34*, 3292–3308.

Libralesso, L., Bouhassoun, A., Cambazard, H., & Jost, V. (2020). Tree search for the sequential ordering problem. In *ECAI 2020 - 24th European Conference on Artificial Intelligence* (pp. 459–465). IOS Press volume 325 of *Frontiers in Artificial Intelligence and Applications*.

Libralesso, L., & Fontan, F. (2021). An anytime tree search algorithm for the 2018 ROADEF/EURO challenge glass cutting problem. *European Journal of Operational Research*, *291*, 883–893.

Liu, X., Laporte, G., Chen, Y., & He, R. (2017). An adaptive large neighborhood search metaheuristic for agile satellite scheduling with time-dependent transition time. *Computers and Operations Research*, *86*, 41–53.

Malandraki, C., & Daskin, M. S. (1992). Time dependent vehicle routing problems: Formulations, properties and heuristic algorithms. *Transportation Science*, *26*, 185–200.

Malandraki, C., & Dial, R. B. (1996). A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem. *European Journal of Operational Research*, *90*, 45–55.

Melgarejo, P. A., Laborie, P., & Solnon, C. (2015). A time-dependent no-

overlap constraint: Application to urban delivery problems. *Lecture Notes in Computer Science*, *9075*, 1–17.

Montero, A., Méndez-Díaz, I., & Miranda-Bront, J. (2017). An integer programming approach for the time-dependent traveling salesman problem with time windows. *Computers and Operations Research*, *88*, 280–289.

Ohlmann, J., & Thomas, B. (2007). A Compressed-Annealing Heuristic for the Traveling Salesman Problem with Time Windows. *INFORMS Journal on Computing*, *19*, 80–90.

Pesant, G., Gendreau, M., Potvin, J.-Y., & Rousseau, J.-M. (1998). An Exact Constraint Logic Programming Algorithm for the Traveling Salesman Problem with Time Windows. *Transportation Science*, *32*, 12–29.

Potvin, J.-Y., & Bengio, S. (1996). The vehicle routing problem with time windows. *INFORMS Journal on Computing*, *8*, 165–172.

Rifki, O., Chiabaut, N., & Solnon, C. (2020). On the impact of spatio-temporal granularity of traffic conditions on the quality of pickup and delivery optimal tours. *Transportation Research Part E*, *142*, 102085.

Roberti, R., & Toth, P. (2012). Models and algorithms for the Asymmetric Traveling Salesman Problem: an experimental comparison. *EURO Journal on Transportation and Logistics*, *1*, 113–133.

Sun, P., Veelenturf, L. P., Hewitt, M., & Van Woensel, T. (2020). Adaptive large neighborhood search for the time-dependent profitable pickup and delivery problem with time windows. *Transportation Research Part E*, *138*, 101942.

Touzout, F. A., Ladier, A. L., & Hadj-Hamou, K. (2022). An assign-and-route matheuristic for the time-dependent inventory routing problem. *European Journal of Operational Research*, *300*, 1081–1097.

Vadlamudi, S. G., Gaurav, P., Aine, S., & Chakrabarti, P. P. (2012). Anytime Column Search. In *Advances in Artificial Intelligence - Australasian Joint Conference* (pp. 254–265). Springer volume 7691 of *LNCS*.

Vu, D. M., Hewitt, M., Boland, N., & Savelsbergh, M. (2020). Dynamic Discretization Discovery for Solving the Time-Dependent Traveling Salesman Problem with Time Windows. *Transportation Science*, *54*, 703–720.

Vu, D. M., Hewitt, M., & Vu, D. D. (2022). Solving the time dependent minimum tour duration and delivery man problems with dynamic discretization discovery. *European Journal of Operational Research*, *302*, 831–846.