Exact and anytime approach for solving the Time Dependent Traveling Salesman Problem with Time Windows

Romain Fontaine, Jilles Dibangoye, Christine Solnon

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
{romain.fontaine,jilles-steeve.dibangoye,christine.solnon}@insa-lyon.fr

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 Window* (TW) constraints. The relevance of considering TD travel times in an urban context is studied in [13] on realistic data; it is shown that this reduces TW violations and also, in some cases, tour durations.

Definition of 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). $\mathcal{C} = \mathcal{V} \setminus \{0, n\}$ 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.

TWs are hard constraints, but it is possible to arrive earlier than e_i on node i. 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 the waiting time before the opening if we arrive too early (*i.e.*, $t_{\uparrow[e_i,l_i]} = e_i$ if $t < e_i$). If the constraint is violated (*i.e.*, $t > l_i$), we set $t_{\uparrow[e_i,l_i]} = \infty$. Otherwise, we have $t_{\uparrow[e_i,l_i]} = t$.

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 $a_{i,j}(t) = t + c_{i,j}(t)$. We assume that TD cost functions satisfy the First-In First-Out (FIFO) property [9]. This property ensures that each 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.

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.

Related work. The TD-TSP has been introduced in [11]. Since then, different approaches have been proposed to solve this problem (or its variants) and a review may be found in [7]. It includes approaches based on metaheuristics which provide no guarantee on solution quality. In this paper, we focus on exact approaches. Many of these approaches are based on *Integer Linear Programming* (ILP), and two state-of-the-art approaches are [1] and [15]. In [15], Dynamic Discretization Discovery is used to dynamically refine time steps and state-of-the-art results are obtained on instances with very tight TWs. In [1], a relaxation to the TSPTW is used to compute bounds: this relaxation is tight when congestion patterns are similar among all arcs of the graph, and this approach obtains state-of-the-art results in this case.

An other exact approach is Dynamic Programming (DP), which has been initially proposed in [2] for the TSP, and extended to handle TD cost functions in [12] and TWs in [4]. These approaches basically explore a state space composed of $\mathcal{O}(n \cdot 2^n)$ states in a level-wise manner. To avoid combinatorial explosion, we may consider a State Space Relaxation (SSR) by merging

some states into a single one to compute lower bounds, as proposed in [4]. Another possibility is to limit the number of states stored at each level to compute upper bounds, as suggested in Restricted DP (RDP) [12].

In [3], a framework based on Multivalued Decision Diagrams is introduced for solving problems that have DP formulations. It is based on RDP and SSR and 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). This approach is improved by [8], which computes new bounds and presents results for several problems, including the TSPTW. More generally, various anytime extensions of A* have been proposed to speed-up the exploration of state spaces. For example, *Iterative Memory Bounded A** (IMBA*) is an anytime variant of A* which has won the 2018 ROADEF challenge [10]. The basic idea of IMBA* is to limit the number of stored states to a parameter D which is progressively increased (when $D = 1$, the algorithm behaves like a greedy one; when $D = \infty$, the algorithm is exact).

In this paper, we propose to use *Anytime Column Search* (ACS) which has been introduced in [14]. ACS is both exact and anytime and, like IMBA*, it iterates A*-like searches. However, instead of bounding the number of stored states, ACS only expands the best state of each level.

Organization of the paper. Our solving approach is presented in Section 2. Experimental results are presented in Section 3. Conclusions and future works are discussed in Section 4. Due to space limits, many details on both the solving approach and experimental results have been omitted; more details may be found in the research report [6].

2 New Approach for the TD-TSPTW

DP. Let us first 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: if $\mathcal{S} = \emptyset$, then $p(i, \mathcal{S}) = a_{0,i}(t_0)_{\uparrow[e_i,l_i]}$; otherwise $p(i, \mathcal{S}) = \min_{j \in \mathcal{S}} a_{j,i}(p(j, \mathcal{S} \setminus \{j\}))_{\uparrow[e_i,l_i]}$. The optimal solution is $p(n, \mathcal{C})$, *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.

State-transition graph. The optimal solution 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 initial 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 final depot n. Edges of the state-transition graph correspond to transitions between states: there is an edge from $(i, \mathcal{S}, t)$ to $(j, \mathcal{S} \cup \{i\}, a_{i,j}(t)_{\uparrow[e_j,l_j]})$ for each customer $j \in \mathcal{C} \setminus (\mathcal{S} \cup \{i\})$ such that $a_{i,j}(t) \leq l_j$. The goal is to find a path from an initial state to a final state $(n, \mathcal{C}, t)$ such that t is minimal.

Exploration of the state-transition graph with ACS. Our instantiation of ACS searches for paths in the state-transition graph, 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 (bounding functions are described below).

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 $(i, \mathcal{S}, t)$

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 $k > 0$. While there exist open nodes, we consider each level k ranging from 0 to $n-2$ and we expand the most promising open state s of that level (*i.e.*, the one that minimizes $f(s)$).

ACS ends when all open sets are empty. In this case, the last solution found is optimal (see proof in [14]). 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 (*i.e.*, the one obtained by a greedy algorithm guided by $f(s)$). Of course, in presence of TWs it may be necessary to iterate more than once before finding a solution.

Propagation of TWs. TW constraints are propagated at the beginning of the search in order to infer precedence relations and tighten other TWs, using the same rules as in [15]. These rules operate on two sets $\mathcal{E}$ and $\mathcal{R}$, where $\mathcal{E}$ represents the set of edges that may be used in a feasible solution, and $\mathcal{R}$ is the set of precedence relations between nodes. Both sets are used to prune the search space by discarding transitions leading to infeasible solutions.

To prevent exploring states leading to solutions worse than the best known, we constrain the latest ending time of a tour l_n to be equal to the ending time of the best known solution and propagate TW constraints again. As far as we know, it is the first time this procedure is used to prune the search space during the search.

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 during search to (i) expand first the most promising state of each level, and (ii) prune the state space when a state cannot lead to a better solution.

A first step to compute $f(s)$ is to compute a graph $G_s = (\mathcal{V}_s, \mathcal{E}_s)$ in which we solve a relaxation of the shortest Hamiltonian path problem from i to n. The vertices of this graph are $\mathcal{V}_s = \{n\} \cup \mathcal{C} \setminus \mathcal{S}$. A straightforward definition of the set of edges 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 further refine $\mathcal{E}_s$ by (i) removing arcs leaving the current node i that violate precedence constraints in set $\mathcal{R}$, and (ii) removing arcs that cannot be used after the current time t, based on their latest departure time. Finally, for each edge $(j, k) \in \mathcal{E}_s$, we consider a constant cost. A basic definition of this constant cost is $\min_{t \in [e_j, l_j]} c_{j,k}(t)$. We improve this definition by tightening l_j to the latest departure time from j to reach k no later than l_k while leaving j no later than l_j.

Given G_s, we may compute different bounds that provide different trade-offs between computational cost and tightness. In [6], we fully describe three different lower bounds: f_{FEA} that checks in constant time that each vertex in $\mathcal{V}_s \setminus \{n\}$ (resp. $\mathcal{V}_s \setminus \{i\}$) has at least one outgoing (resp. incoming) arc in $\mathcal{E}_s$, f_{OIA} that computes in linear time the sum of the minimum-weight outgoing arc for each node in $\mathcal{V}_s \setminus \{n\}$ and the sum of the minimum-weight incoming arc for each node in $\mathcal{V}_s \setminus \{i\}$ and returns the maximum of these two sums, and f_{MSA} that computes in $\mathcal{O}(\#\mathcal{E}_s \log \#\mathcal{V}_s)$ the minimum spanning arborescence rooted at i in G_s.

Hybridization with Local Search. To converge faster towards good solutions, we try to improve each solution provided by ACS using a LS procedure similar to the one used by [5] for the TSPTW. Sets $\mathcal{E}$ and $\mathcal{R}$ are used to reduce the size of the neighborhoods. We use the *First Improvement* strategy, *i.e.*, we accept improving moves until reaching a local optimum.

3 Experimental Results

Considered approaches. We consider two variants of our approach that only differ on the computation of the lower bound f: FEA (resp. OIA) denotes the variant obtained when $f = f_{\text{FEA}}$ (resp. $f = f_{\text{OIA}}$). Results for $f = f_{\text{MSA}}$ are omitted as this bound rarely obtains better results than when $f \in \{f_{\text{FEA}}, f_{\text{OIA}}\}$. Our approach is compared to the ILP approaches of [1] and [15], respectively denoted ARI18 and VU20.

	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	-	0	-	1	3579	1	3575	59	1035	33	916	54	679	51	637	51	565	60	28
.25	0	-	52	1497	55	1337	55	1337	60	258	59	83	60	23	60	26	60	24	60	9
.50	46	1595	60	25	60	17	60	17	60	6	60	1	60	0	60	0	60	0	60	0
Total	46		112		116		116		179		152		174		171		171		180	

TAB. 1: Performance of OIA variants on B_{ARI18} with $n = 31$ (60 instances per value of β).

Benchmarks. The benchmark used by [1] to evaluate ARI18 is denoted B_{ARI18}. It is randomly generated according to the following parameters: the number of vertices $n \in \{21, 31, 41\}$, the congestion factor $\Delta \in \{.7, .8, .9, .95, .98\}$, the traffic pattern $P \in \{B_1, B_2\}$ and the TW tightness $\beta \in \{0, .25, .50, 1\}$ (the smaller β, the wider the TWs). There are 30 instances for each combination (n, β, Δ, P), leading to a total of 3600 instances.

In [15], this benchmark is extended to instances with $n \in \{60, 80, 100\}$ and tight TWs only. This benchmark is denoted B_{VU20}.

Considered hardware and performance measures. FEA and OIA are run on 2.1GHz Intel Xeon E5-2620 v4 processors with 64GB RAM. Run times of ARI18 and VU20 are those reported by [1] and [15], 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.

An instance is considered 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 (*i.e.*, the best known solution), and t_r denotes the average time needed to find the reference solution for these instances. 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. $\#r$). Note that for ARI18 and VU20, $\#r$ and t_r are not available.

Analysis of the algorithm's components. ACS is combined with three key components described in Section 2, *i.e.*, TW constraint propagation, LS and the filtering of edge set $\mathcal{E}_s$ used to compute f. To evaluate the relevance of these components, we report results obtained with different variants obtained by disabling them. We consider the following variants of f_{OIA} (conclusions are similar using f_{FEA}): in OIA$_0$, all key components are disabled; OIA$_1$ is obtained from OIA$_0$ by enabling TW constraint propagation before starting the search; OIA$_2$ is obtained from OIA$_1$ by also enabling TW constraint propagation during the search; OIA$_3$ is obtained from OIA$_2$ by enabling LS, and 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 B_{ARI18}. The left-hand side of Table 1 shows that all components except LS improve solving performance. Similarly, the right-hand side of Table 1 demonstrates that all components improve convergence speed, except the propagation of TW constraints during the search.

Experimental Comparison with ARI18. We now compare our approach with ARI18 on benchmark B_{ARI18}. In Table 2a, we report the number of solved instances and the solving time of ARI18, FEA, and OIA. When $n \leq 31$, most instances are solved relatively quickly by FEA and OIA. Generally, both bounds provide different trade-offs between solving abilities, convergence speed, and memory use.

Overall, OIA solves 701 more instances than ARI18 on the full benchmark. Even if ARI18 has been run on a different computer, on most instance classes the difference in solving times cannot come from this sole fact. However, when $n = 41$ and $\beta \in \{0, 0.25\}$, only 17 (resp. 5) instances are solved by FEA (resp. OIA) whereas ARI18 is able to solve 241 instances. Of course, when an instance is not solved, our approach has found an approximate solution, as

		Solved instances						Ref. solutions			
		ARI18		FEA		OIA		FEA		OIA	
n	β	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#r$	t_r	$\#r$	t_r
21	0	248	660	300	1	300	1	300	0	300	0
	.25	286	383	300	0	300	0	300	0	300	0
	.50	296	289	300	0	300	0	300	0	300	0
	1	300	29	300	0	300	0	300	0	300	0
31	0	155	1631	300	614	292	997	300	85	300	30
	.25	199	1274	300	163	300	257	300	22	300	7
	.50	157	1433	300	4	300	6	300	1	300	0
	1	233	608	300	0	300	0	300	0	300	0
41	0	110	2263	0	-	0	-	173	463	251	554
	.25	131	1950	17	2747	5	2732	205	267	288	235
	.50	55	2276	252	511	280	614	299	57	300	5
	1	106	528	300	0	300	0	300	0	300	0
Total		2276		2969		2977		3377		3539	

(a) Results of ARI18, FEA, and OIA (300 inst. per row)

		$P = B_1$					$P = B_2$					
	$\beta \setminus \Delta$	.70	.80	.90	.95	.98	.70	.80	.90	.95	.98	Total
ARI18	0	6	8	10	19	28	1	0	3	12	23	110
	.25	6	8	13	23	29	1	0	5	16	30	131
	.50	1	2	4	9	18	1	0	2	5	13	55
	1	14	11	14	12	29	8	4	3	4	7	106
	Total	27	29	41	63	104	11	4	13	37	73	
FEA	0	0	0	0	0	0	0	0	0	0	0	0
	.25	1	1	1	1	2	3	2	2	2	2	17
	.50	24	25	25	25	25	28	25	25	25	25	252
	1	30	30	30	30	30	30	30	30	30	30	300
	Total	55	56	56	56	57	61	57	57	57	57	

(b) Number of instances solved by ARI18 and FEA with respect to P, Δ and β for $n = 41$ (30 instances per class)

TAB. 2: Performance on B_{ARI18}.

shown on the right-hand side of Table 2a. The success of ARI18 is strongly related to Δ as it relies on bounds which are tighter when Δ is closer to 1. To illustrate this, we detail in Table 2b 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 the success of our approach mainly depends on the TW width β instead. Our approach indeed has comparable performance when using the realistic benchmark [13] with similar TWs, which (i) was generated using a realistic traffic simulator and (ii) models the fact that the fastest path between two customers may change depending on the departure time. In this benchmark, Δ never exceeds 0.35. Given that the performance of ARI18 drops when $\Delta < 0.9$ (see Table 2b), difficulties can be excpected from this approach when trying to solve these instances.

Experimental Comparison with VU20. Let us now compare our approach with VU20 on benchmark B_{VU20} which has more customers but tighter TWs. FEA solves all 720 instances in 5s on average, whereas OIA (resp. VU20) solves all but 1 (resp. 19) in 11s (resp. 203s) on average. FEA is always at least ten times as fast as VU20. This difference is large enough to allow us to conclude that FEA is more efficient than VU20 even though the two approaches have been run on different computers. Also, all reference solutions are found in a matter of seconds for both FEA and OIA.

Experimental evaluation on the TSPTW. We also compared our approach on TSPTW benchmarks (with constant cost functions), both with the LS-based approach of [5] (denoted DAS10) and with the recent exact approach of [8] (denoted GIL21). We considered a classical set of benchmarks (592 instances containing up to 400 nodes), and all algorithms ran on the same hardware. We only report general conclusion due to space limits. Both FEA and OIA outperform GIL21 in terms of optimality proofs, as they respectively solve 96%, 96% and 61% of the instances. FEA and OIA find all reference solutions whereas GIL21 does not find 2% of them. DAS10 is competitive with our approach for very short execution times, but it does not find the reference solutions for 8% of the instances. Finally, let us note that our approach requires less memory than GIL21, but more than DAS10.

4 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 considered multiple bounds providing different trade-offs and experiments have shown us that f_{OIA} offers a good compromise between tightness and computational cost. Our approach outperforms the ILP approach of [15] on all instances of B_{Vu20} which have very tight TWs. It also outperforms the ILP approach of [1] on most instances of B_{Ari18}: it is outperformed on the largest instances that share similar congestion patterns (*i.e.*, when $\Delta \geq 0.95$), but benchmarks generated from real data have much smaller values of Δ. Our approach may also be used to solve the TSPTW and we have shown that it outperforms the DP-based approach of [8] and the LS-based approach of [5].

We plan to extend our approach to other TD problems such as, for example, VRPs and Pickup and Delivery Problems, and to compare ACS with other anytime A*-based approaches such as the one of [10].

References

[1] A. Arigliano, G. Ghiani, A. Grieco, E. Guerriero, and I. Plana. TD-ATSPTW: Properties and an exact algorithm. *DAM*, 261:28–39, may 2018.

[2] R. Bellman. Dynamic Programming Treatment of the Travelling Salesman Problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962.

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

[4] N. Christofides, A. Mingozzi, and P. Toth. State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2):145–164, 1981.

[5] R. F. Da Silva and S. Urrutia. A General VNS heuristic for the traveling salesman problem with time windows. *Discrete Optimization*, 7(4):203–211, 2010.

[6] R. Fontaine, J. Dibangoye, and C. Solnon. Exact and Anytime Approach for Solving the TD-TSPTW. Technical report, Université de Lyon ; INSA Lyon ; INRIA, November 2022.

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

[8] X. Gillard, V. Coppé, P. Schaus, and A. A. Cire. Improving the Filtering of Branch-and-Bound MDD Solver. *CPAIOR*, 12735 LNCS:231–247, 2021.

[9] S. Ichoua, M. Gendreau, and J. Y. Potvin. Vehicle dispatching with time-dependent travel times. *EJOR*, 144(2):379–396, 2003.

[10] L. Libralesso and F. Fontan. An anytime tree search algorithm for the 2018 ROADEF/EURO challenge glass cutting problem. *EJOR*, 291(3):883–893, 2021.

[11] C. Malandraki and M. S. Daskin. Time dependent vehicle routing problems: Formulations, properties and heuristic algorithms. *Transportation Science*, 26(3):185–200, 1992.

[12] C. Malandraki and R. B. Dial. A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem. *EJOR*, 90(1):45–55, 1996.

[13] O. Rifki, N. Chiabaut, and C. Solnon. On the impact of spatio-temporal granularity of traffic conditions on the quality of pickup and delivery optimal tours. *Transportation Research Part E: Logistics and Transportation Review*, 142, 2020.

[14] S. G. Vadlamudi, P. Gaurav, S. Aine, and P. P. Chakrabarti. Anytime Column Search. In *AI 2012: Advances in Artificial Intelligence*, volume 7691 of *LNCS*. Springer, 2012.

[15] D. M. Vu, M. Hewitt, N. Boland, and M. Savelsbergh. Dynamic Discretization Discovery for Solving the TD-TSPTW. *Transportation Science*, 54(3):703–720, 2020.