Exact and anytime heuristic search for the Time Dependent Traveling Salesman Problem with Time Windows

Romain Fontaine

N° d'ordre NNT : 2024ISAL0067

THESE de DOCTORAT DE L'INSA LYON,
membre de l'Université de Lyon

Ecole Doctorale N° 512
Informatique et Mathématiques

Spécialité / discipline de doctorat :
Informatique

Soutenue publiquement le 09/07/2024, par :

Romain Fontaine

Exact and anytime heuristic search for the Time Dependent Traveling Salesman Problem with Time Windows

---

Devant le jury composé de :

PRALET	Cédric	Directeur de Recherche	ONERA	Rapporteur
SCHAUS	Pierre	Professeur des Universités	UC Louvain	Rapporteur
BILLOT	Romain	Professeur des Universités	IMT Atlantique	Examinateur
SOLNON	Christine	Professeure des Universités	INSA Lyon	Directrice de thèse
DIBANGOYE	Jilles S.	Maître de Conférences HDR	University of Groningen	Co-directeur de thèse

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Département de la Formation par la Recherche et des Études Doctorales (**FEDORA**)

Bâtiment INSA direction, 1er étage
37, av. J. Capelle
69621 Villeurbanne Cédex
fedora@insa-lyon.fr

Référence : TH1123_FONTAINE Romain

L'INSA Lyon a mis en place une procédure de contrôle systématique via un outil de détection de similitudes (logiciel Compilatio). Après le dépôt du manuscrit de thèse, celui-ci est analysé par l'outil. Pour tout taux de similarité supérieur à 10%, le manuscrit est vérifié par l'équipe de FEDORA. Il s'agit notamment d'exclure les auto-citations, à condition qu'elles soient correctement référencées avec citation expresse dans le manuscrit.

Par ce document, il est attesté que ce manuscrit, dans la forme communiquée par la personne doctorante à l'INSA Lyon, satisfait aux exigences de l'Etablissement concernant le taux maximal de similitude admissible.

Département FEDORA – INSA Lyon - Ecoles Doctorales

SIGLE	ECOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
ED 206 CHIMIE	**CHIMIE DE LYON** **https ://www.edchimie-lyon.fr** Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage **secretariat@edchimie-lyon.fr**	**M. Stéphane DANIELE** C2P2-CPE LYON-UMR 5265 Bâtiment F308, BP 2077 43 Boulevard du 11 novembre 1918 69616 Villeurbanne **directeur@edchimie-lyon.fr**
ED 341 E2M2	**ÉVOLUTION, ÉCOSYSTÈME, MICROBIOLOGIE, MODÉLISATION** **http ://e2m2.universite-lyon.fr** Sec. : Bénédicte LANZA Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 **secretariat.e2m2@univ-lyon1.fr**	**Mme Sandrine CHARLES** Université Claude Bernard Lyon 1 UFR Biosciences Bâtiment Mendel 43, boulevard du 11 Novembre 1918 69622 Villeurbanne CEDEX **e2m2.codir@listes.univ-lyon1.fr**
ED 205 EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTÉ** **http ://ediss.universite-lyon.fr** Sec. : Bénédicte LANZA Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 **secretariat.ediss@univ-lyon1.fr**	**Mme Sylvie RICARD-BLUM** Laboratoire ICBMS - UMR 5246 CNRS - Université Lyon 1 Bâtiment Raulin - 2ème étage Nord 43 Boulevard du 11 novembre 1918 69622 Villeurbanne Cedex Tél : +33(0)4 72 44 82 32 **sylvie.ricard-blum@univ-lyon1.fr**
ED 34 EDML	**MATÉRIAUX DE LYON** **http ://ed34.universite-lyon.fr** Sec. : Yann DE ORDENANA Tél : 04.72.18.62.44 **yann.de-ordenana@ec-lyon.fr**	**M. Stéphane BENAYOUN** Ecole Centrale de Lyon Laboratoire LTDS 36 avenue Guy de Collongue 69134 Ecully CEDEX Tél : 04.72.18.64.37 **stephane.benayoun@ec-lyon.fr**
ED 160 EEA	**ÉLECTRONIQUE, ÉLECTROTECHNIQUE, AUTOMATIQUE** **https ://edeea.universite-lyon.fr** Sec. : Philomène TRECOURT Bâtiment Direction INSA Lyon Tél : 04.72.43.71.70 **secretariat.edeea@insa-lyon.fr**	**M. Philippe DELACHARTRE** INSA LYON Laboratoire CREATIS Bâtiment Blaise Pascal, 7 avenue Jean Capelle 69621 Villeurbanne CEDEX Tél : 04.72.43.88.63 **philippe.delachartre@insa-lyon.fr**
ED 512 INFOMATHS	**INFORMATIQUE ET MATHÉMATIQUES** **http ://edinfomaths.universite-lyon.fr** Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage Tél : 04.72.43.80.46 **infomaths@univ-lyon1.fr**	**M. Hamamache KHEDDOUCI** Université Claude Bernard Lyon 1 Bât. Nautibus 43, Boulevard du 11 novembre 1918 69 622 Villeurbanne Cedex France Tél : 04.72.44.83.69 **direction.infomaths@listes.univ-lyon1.fr**
ED 162 MEGA	**MÉCANIQUE, ÉNERGÉTIQUE, GÉNIE CIVIL, ACOUSTIQUE** **http ://edmega.universite-lyon.fr** Sec. : Philomène TRECOURT Tél : 04.72.43.71.70 Bâtiment Direction INSA Lyon **mega@insa-lyon.fr**	**M. Etienne PARIZET** INSA Lyon Laboratoire LVA Bâtiment St. Exupéry 25 bis av. Jean Capelle 69621 Villeurbanne CEDEX **etienne.parizet@insa-lyon.fr**
ED 483 ScSo	**ScSo**[1] **https ://edsciencessociales.universite-lyon.fr** Sec. : Mélina FAVETON Tél : 04.78.69.77.79 **melina.faveton@univ-lyon2.fr**	**M. Bruno MILLY** (INSA : J.Y. TOUSSAINT) Univ. Lyon 2 Campus Berges du Rhône 18, quai Claude Bernard 69365 LYON CEDEX 07 Bureau BEL 319 **bruno.milly@univ-lyon2.fr**

1. ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

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: travel times between points to visit depend on departure times instead of being constant. The TD-TSPTW further generalizes this problem by adding Time Window constraints, *i.e.,* constraints on visit times. Existing exact approaches such as Integer Linear Programming and Dynamic Programming usually do not scale well; heuristic approaches scale better but provide no guarantees on solution quality.

In this thesis, we introduce a new exact and anytime solving approach for the TD-TSPTW which aims at quickly providing approximate solutions and gradually improving them until proving optimality. We first show how to reduce the TD-TSPTW to the search for a best path in a state-transition graph. We provide an overview of existing search algorithms, with a focus on exact and anytime extensions of A*, and introduce a new one by hybridizing two of them. We show how to combine these exact and anytime search algorithms with local search – in order to faster find solutions of higher quality – and with bounding and time window constraint propagation – in order to filter the search space. Finally, we provide extensive experimental results to (i) validate our main design choices, (ii) compare our approach to state-of-the-art solving approaches on various TD benchmarks with different degrees of realism and different temporal granularities and (iii) compare TD solving approaches to recent TSPTW solvers on constant benchmarks. These experimental results show us that our approach offers a good compromise between the time needed to find good solutions and the time needed to find optimal solutions and prove their optimality for both TD and constant TSPTW instances.

Keywords: Combinatorial Optimization, Traveling Salesman Problem, Time Window constraints, Time-Dependent travel times, Dynamic Programming, Exact and Anytime Search Algorithms, A* algorithm, Scalability.

Résumé

Le problème du voyageur de commerce (TSP, pour *Traveling Salesman Problem*) dépendant du temps (TD, pour *Time Dependent*) est une généralisation du TSP qui permet de prendre en compte les conditions de trafic lors de la planification de tournées en milieu urbain : les temps de trajet varient en fonction des horaires de départ au lieu d'être constants. Le TD-TSPTW généralise ce problème en associant à chaque point de passage une fenêtre temporelle (TW, pour *Time Window*) qui restreint les horaires de visite. Les approches de résolution exactes telles que la programmation linéaire en nombres entiers ou la programmation dynamique passent mal à l'échelle, tandis que les approches heuristiques ne garantissent pas la qualité des solutions obtenues.

Dans cette thèse, nous proposons une nouvelle approche exacte et anytime pour le TD-TSPTW visant à obtenir rapidement des solutions approchées puis à les améliorer progressivement jusqu'à prouver leur optimalité. Nous montrons d'abord comment rapporter le TD-TSPTW à une recherche de meilleur chemin dans un graphe états-transitions. Nous décrivons ensuite des algorithmes permettant de résoudre ce problème en nous concentrant sur les extensions exactes et anytime d'A*, et en proposons une nouvelle par hybridation. Nous montrons comment combiner ces algorithmes avec de la recherche locale – afin de trouver plus rapidement de meilleures solutions – ainsi qu'avec des bornes et de la propagation de contraintes de TW – afin de réduire la taille de l'espace de recherche. Enfin, nous fournissons des résultats expérimentaux visant à (i) valider nos principaux choix de conception, (ii) comparer notre approche à l'état de l'art en considérant des benchmarks ayant différents degrés de réalisme et différentes granularités temporelles et (iii) comparer ces approches TD à de récents solveurs pour le TSPTW dans le cas constant. Ces résultats montrent que notre approche apporte un bon compromis entre le temps nécessaire pour (i) trouver de bonnes solutions et (ii) trouver des solutions optimales et prouver leur optimalité, aussi bien dans le cas TD que dans le cas constant.

Mots-clés : Optimisation Combinatoire, Problème du Voyageur de Commerce, Contraintes de fenêtres temporelles, Temps de trajets Time-Dependent, Programmation Dynamique, Algorithmes de recherche Exacts et Anytime, Algorithme A*, Passage à l'échelle.

Contents

Introduction

With the increase in the number of individual cars and in the density of population in urban and peri-urban areas, congestion of transportation infrastructures has become a major issue: it has well-known hazardous effects on the environment and on the quality of life of inhabitants. Such congestions lead to variations in traffic conditions which need to be taken into account when planning itineraries or tours, whether for transportation of passengers or goods.

Taking into account traffic conditions when planning itineraries or tours has been made possible by the availability of traffic data (which may come, *e.g.,* from physical sensors placed on the road networks or from increasingly widespread mobile navigation services), which in turn permitted to build predictive models [Sal19].

From an academical point of view, finding optimal tours is often associated to the well-known *Traveling Salesman Problem* (TSP): given a list of points to visit and pairwise travel times between them, solving this problem requires finding the fastest route which visits each point once before returning to its starting point. The time-dependent TSP (TD-TSP) [MD92] generalizes the TSP by modeling variations in traffic conditions throughout the day and thus allows one to plan tours in a urban context. The TSP with Time Windows (TSPTW) is another generalization of the TSP, which consists in restricting the set of possible visit times at each location. In this thesis, we consider a combination of both these generalizations (*i.e.,* the TD-TSPTW), in which (i) travel times are time-dependent (TD) and (ii) hard Time Window (TW) constraints are associated to each location. TW constraints are pertinent both in the context of delivery tours (in order to model the availability of customers), for on-demand passenger transportation (which may be multi-modal), and – more generally – whenever synchronization is necessary.

The relevance of considering TD travel times when planning tours in an urban context has been studied on realistic data in [RCS20]: results have shown that considering TD travel times when planning (instead of assuming them to be constant) tends to lead to fewer constraint violations. Minimizing the chances of such violations is critical when considering on-demand passenger transportation, as such a service is unlikely to be accepted by end-users if it is not the case.

The TSP is NP-Hard and the size of the search space grows exponentially with respect to the number of locations to visit (unless P = NP). Obviously, the presence of TW constraints reduces the number of possibilities, but makes it harder to find feasible solutions. Moreover, optimizing with TD travel times may, depending on the solving approach adopted, bring additional difficulties. Although the TD-TSP was introduced more than thirty years ago, the TD-TSPTW has recently

received a lot of attention (*e.g.,* [Ari+19], [Vu+20], [LMS22], [Pra23]). Nevertheless, existing *exact* approaches – which are usually based on Integer Linear Programming or Dynamic Programming (DP) – have the advantage of providing guarantees on the quality of the solutions found, but typically do not scale well and may fail to provide a solution within reasonable space and time constraints. On the other hand, *heuristic* approaches have better scaling abilities but are not able to provide guarantees on solution quality.

Consequently, we propose a TD-TSPTW solving approach which seeks to get the best of both worlds by being both exact and *anytime* (*i.e.,* able to quickly provide approximate solutions). Such hybrid approaches aim to be more scalable than purely exact approaches, which fail to provide a solution when lacking space and time resources to prove optimality.

Thesis outline and contributions. This thesis is composed of three parts: in Chapters 1 to 3, we introduce fundamental notions which are necessary to understand our solving approach. We then introduce our solving approach in Chapters 4 and 5 and provide extensive experimental results in Chapters 6 to 8.

In Chapter 1, we formally define the TD-TSPTW, present common objective functions (in this thesis, we consider the *makespan* objective), and study how the TD-TSPTW relates to other routing problems. We then provide an overview of existing solving approaches and describe classic TW constraint propagation rules. Also, we discuss how TD travel time functions may be obtained from real-world data and describe common ways to represent these functions. Finally, we provide an overview of benchmarks commonly used to evaluate TD-TSPTW solvers: we show that some of them are based on unrealistic assumptions and consequently suggest more realistic alternatives.

In Chapter 2, we show how Dynamic Programming may be used to solve the TSP and some of its variants. We also discuss of classic techniques which may be used to improve its scalability, and present problem-agnostic optimization frameworks related to this paradigm.

In Chapter 3, we show how optimization problems such as the TSP may be solved by looking for optimal paths in *state transition graphs*. Of course, the size of these graphs may grow too large for fundamental shortest path algorithms to be able to solve large problem instances. Consequently, we describe the A* algorithm – which uses heuristic functions to search more efficiently – before providing an overview and a taxonomy of existing Exact and Anytime Search (EAS) algorithms related to A*.

In Chapter 4, we describe how to instantiate EAS algorithms to solve the TD-TSPTW in a scalable way: we start by defining a state transition graph associated to its DP formulation and then instantiate four EAS algorithms to solve it. We compare these algorithms by highlighting their similarities and differences, which leads us to proposing an improved version of an existing EAS algorithm. Also, we discuss major – yet often disregarded – implementation issues regarding data structures and show that obvious implementations of A* are not necessarily suited to all EAS algorithms. Finally, we introduce three heuristic functions for the TD-TSPTW and describe how they differ from those used in related works: these functions are used by EAS algorithms to

guide search and to prune the search space.

In Chapter 5, we propose to enhance the performance of our solving approach by combining EAS algorithms with (i) a local search procedure which tries to improve the solutions found, and (ii) TW constraint propagation, in order to reduce the size of the search space and to compute tighter bounds.

In Chapter 6, we experimentally validate the relevance of our solving approach's key components and compare four EAS algorithms after having tuned their parameters and studied the influence of various implementation decisions.

In Chapter 7, we experimentally compare three variants of our solving approach – which only differ in the heuristic function used – to state-of-the-art solvers. We consider both (i) benchmarks commonly used to evaluate TD-TSPTW solving approaches and (ii) a benchmark with more realistic TD travel time functions. Finally, we study how these solvers' performance varies when considering TD travel time functions with different degrees of realism and different temporal granularities. Results show us that our approach (i) offers a good compromise between the time needed to find good solutions and the time needed to find optimal solutions and to prove their optimality, and (ii) is relatively robust to increases in the granularity of TD travel time functions.

Finally, in Chapter 8, we study how TD-TSPTW solvers compete with various recent TSPTW solving approaches when considering benchmarks in which travel times are constant.

Publications. The TD-TSPTW solving approach we propose and part of the experimental results have been published in [FDS23b]:

> Romain Fontaine, Jilles Dibangoye, and Christine Solnon. "Exact and anytime approach for solving the time dependent traveling salesman problem with time windows". In: *European Journal of Operational Research* 311.3 (2023). ISSN: 0377-2217

These works were presented during the 2023 edition of the national ROADEF conference[1] (our paper was one of the six finalists out of thirteen candidates for the best paper award) and during the doctoral program of CP2023[2].

[1] Romain Fontaine, Jilles Dibangoye, and Christine Solnon. "Exact and Anytime Approach for Solving the Time Dependent Traveling Salesman Problem with Time Windows". In: *24ème congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision*. 2023

[2] *International Conference on Principles and Practice of Constraint Programming.*

Glossary

Abbreviation	Meaning
BFS	Breadth-First Search
DAG	Directed Acyclic Graph
DFS	Depth-First Search
DP	Dynamic Programming
EAS	Exact and Anytime Search
FIFO	First-In, First-Out
LDT	Latest Departure Time
OTW	Overlapping Time Windows
RDP	Restricted Dynamic Programming
SOP	Sequential Ordering Problem
TD-TSPTW	Time-Dependent Traveling Salesman Problem with Time Windows

Notations

Category	Notation	Description	
Sets	$\lvert\mathcal{A}\rvert$	Cardinality of $\mathcal{A}$	
	$2^{\mathcal{A}}$	Power set of $\mathcal{A}$	
	$[i, j]$	All integers from i to j included ($\emptyset$ whenever $i > j$)	
	$[i, j[$	All integers from i to j excluded ($\emptyset$ whenever $i \geq j$)	
	$\mathbb{N}$	Set of natural numbers, 0 included	
	$\mathbb{N}^+$	Set of natural numbers, 0 excluded	
	$\mathbb{R}_0^+$	Set of non-negative real numbers	
TD-TSPTW	$(\mathcal{V}, c, e, l)$	TD-TSPTW instance	p. 12
	$\mathcal{V} = \{0, 1, ..., n\}$	Vertex Set	p. 12
	$\mathcal{C} = \mathcal{V} \setminus \{0, n\}$	Customer Set	p. 12
	$c_{i,j}(t)$	Travel time from i to j when departing from i at time t	p. 12
	$a_{i,j}(t)$	Arrival time at j when departing from i at time t	p. 12
	$a_{i,j}^{-1}(t)$	LDT from i to reach j no later than time t	p. 13
	e_i	Earliest visiting time of vertex i	p. 12
	l_i	Latest visiting time of vertex i	p. 12
	$t_{\uparrow[e_i,l_i]}$	TW-aware time at vertex i	p. 12
	$G_{\text{PR}} = (\mathcal{V}, \mathcal{R})$	Precedence graph	p. 20
	$G_{\text{UA}} = (\mathcal{V}, \mathcal{E})$	Usable arcs graph	p. 20
TD travel times	$\mathcal{K} = [0, K[$	Set of time-steps	p. 17
DP	$G_{\text{ST}} = (\mathcal{N}, \mathcal{A}, w)$	State transition graph	p. 34
Planning	$(\mathcal{S}_P, \mathcal{A}_P, s_0, s_{\text{f}}, F, \tau, c_P)$	Planning problem	p. 44
	$\mathcal{S}_P$	Set of states	p. 44
	$\mathcal{A}_P$	Set of actions	p. 44
	$s_0,\ s_{\text{f}}$	Initial and final states	p. 44
	$F(s)$	Set of feasible actions in state s	p. 44
	$\tau(s, a)$	State resulting from applying action a in state s	p. 44
	$c_P(s, a)$	Cost of applying action a in state s	p. 44
	G_{ST}^P	State transition graph associated to planning problem P	p. 45
Shortest paths	$g(s)$	Upper bound on the cost-so-far to reach state s	p. 47
	$g^*(s)$	Optimal cost-so-far to reach state s	p. 47
	$h(s)$	Lower bound on the cost-to-go from state s	p. 52
	$h^*(s)$	Optimal cost-to-go from state s	p. 52
	$f(s) = g(s) + h(s)$	Evaluation function of state s	p. 52
Solving approach	$LDT(i, j)$	LDT from i to reach j no later than its deadline	p. 85
	$\underline{c}_{i,j}$	Lower bound on the travel cost from i to j	p. 85
	$G_s = (\mathcal{V}_s, \mathcal{E}_s)$	Subgraph associated to state s	p. 85

Part

I

Background

Chapter

1 Time Dependent TSP with Time Windows

Contents

The *Traveling Salesman Problem* (TSP) consists in finding a tour of minimal cost visiting a given set of locations and returning to the origin location. The *Time-Dependent* TSP (TD-TSP) [MD92] is a generalization of the TSP in which travel times vary throughout the day, thus allowing one to take traffic conditions into account when planning tours in an urban context. The TSP with *Time Windows* (TSPTW) [CMT81] generalizes the TSP by adding hard time window constraints which restrict the possible visit times at each location, and thus the set of feasible tours. The subject of this thesis is the TD-TSPTW, which considers both time-dependent travel times and time window constraints.

In this chapter, we first formalize the general problem of the TD-TSPTW in Section 1.1. We then briefly present how TD travel times are obtained and modeled in Section 1.2, and describe commonly used TW constraint propagation techniques in Section 1.3. We finally present state-of-the-art solving approaches in Section 1.4 and benchmarks commonly used to evaluate them in Section 1.5.

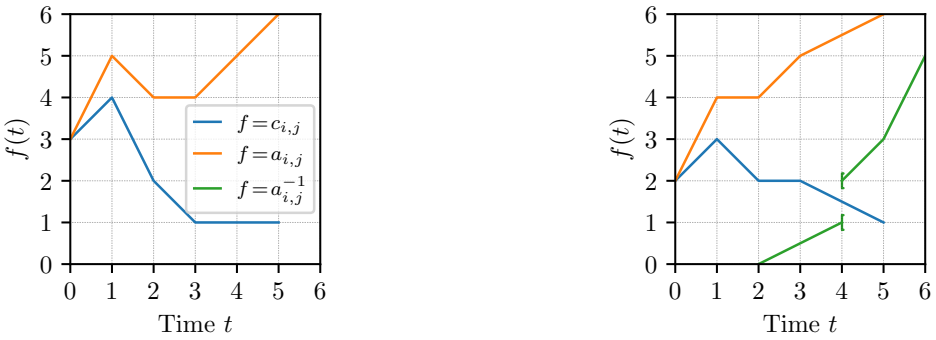
1.1 Definitions and notations

A TD-TSPTW instance is defined as a tuple $(\mathcal{V}, c, e, l)$, where $\mathcal{V}$ denotes the set of vertices to visit and $c : \mathcal{V} \times \mathcal{V} \times \mathbb{N} \to \mathbb{N}$ the TD-travel time matrix. Each vertex $i \in \mathcal{V}$ has an earliest visit time $e_i \in \mathbb{N}$ and a latest visit time $l_i \in \mathbb{N}$. In this thesis, we consider that all times are integers. This assumption is not unrealistic given floating point values may be transformed into integer values through multiplication by a constant coefficient.

More precisely, the set of vertices to visit is defined as $\mathcal{V} = \{0, \dots, n\}$ where 0 is the origin vertex and n the destination vertex (in practice, 0 and n may refer to the same location, *i.e.,* the depot). For convenience, we note $\mathcal{C} = \mathcal{V} \setminus \{0, n\}$ the set of customer vertices. Each vertex $i \in \mathcal{V}$ must be visited during time interval $[e_i, l_i]$. Consequently, planning happens during time horizon $\mathcal{H} = [e_0, l_n]$, and each TW is contained in $\mathcal{H}$, *i.e.,* $[e_i, l_i] \subseteq \mathcal{H}, \forall i \in \mathcal{V}$. It is possible to arrive earlier than time e_i on vertex i but, in this case, it is necessary to wait on i until time e_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.,*

$$t_{\uparrow[e_i,l_i]} = \begin{cases} \max(e_i, t) & \text{if } t \leq l_i \\ \infty & \text{if } t > l_i \end{cases} \tag{1.1}$$

Given a pair of vertices $i, j \in \mathcal{V}$, we note $c_{i,j}$ the TD travel time function such that $c_{i,j}(t)$ is the travel time from i to j when leaving i at time t. For convenience, we note $a_{i,j}$ the arrival time function such that $a_{i,j}(t) = t + c_{i,j}(t)$. We assume that TD travel time functions satisfy the First-In First-Out (FIFO) property (see, *e.g.,* [IGP03]). This property ensures that every arrival time function $a_{i,j}$ is non-decreasing, *i.e.,* $\forall t_1, t_2 \in \mathcal{H}, t_1 < t_2 \Rightarrow a_{i,j}(t_1) \leq a_{i,j}(t_2)$, or equivalently that $c_{i,j}$ has a minimum slope of -1. In other words, when traveling from i to j, delaying the departure from i cannot allow one to arrive sooner at j. We illustrate this property on two travel time functions in Figure 1.1. We also assume that travel time functions satisfy triangle inequality, *i.e.,* $\forall i, j, k \in \mathcal{V}, \forall t \in \mathcal{H}, a_{j,k}(a_{i,j}(t)) \leq a_{i,k}(t)$.



(a) Non-FIFO travel time function

(b) FIFO travel time function

Figure 1.1: Illustration of the FIFO property. Blue curves are TD travel time functions $c_{i,j}$, orange curves are the corresponding arrival time functions $a_{i,j}$. Left: the FIFO property is not satisfied as $a_{i,j}$ decreases between $t = 1$ and $t = 2$. Right: the FIFO property is satisified (*i.e.,* $a_{i,j}$ is non-decreasing) and the inverse arrival time function $a^{-1}_{i,j}$ is displayed in green.

The inverse of $a_{i,j}$ is denoted $a_{i,j}^{-1}$: $a_{i,j}^{-1}(t)$ represents the time at which vertex i must be left to arrive on vertex j at time t. However, we only require travel time functions $c_{i,j}$ to satisfy the FIFO property and do not assume that arrival time functions $a_{i,j}$ are bijective: therefore, when traveling from vertex i to vertex j, (i) it might not be possible to reach j at time t, and (ii) multiple departure times t' may allow one to reach j at time t (this occurs when $a_{i,j}$ has constant parts). Consequently, we define $a_{i,j}^{-1}(t)$ as the latest departure time t' from vertex i allowing one to reach vertex j no later than time t, *i.e.,* $a_{i,j}^{-1}(t) = \text{argmax}_{t' \in \mathcal{H}} \{a_{i,j}(t') \leq t\}$. By the FIFO property, $a_{i,j}(t)$ is non-decreasing: its inverse can therefore be computed in logarithmic time through binary search. We illustrate a sample inverse arrival time function in Figure 1.1b.

Note that service (or *processing*) times $s_i \in \mathbb{N}$ may be associated to vertices $i \in \mathcal{C}$ in order to model the time required to serve customers. More precisely, TW of vertex $i \in \mathcal{C}$ constrains its service to start during time range $[e_i, l_i]$: consequently, the range of possible departure times from i is shifted to the interval $[e_i + s_i, l_i + s_i]$. For clarity – and without loss of generality – we do not consider service times in this thesis, given one can straightforwardly handle them by using an alternative travel time function $c'_{i,j}$ that includes s_i, *i.e.,*

$$c'_{i,j}(t) = \begin{cases} s_i + c_{i,j}(t + s_i) & \text{if } i \in \mathcal{C} \\ c_{i,j}(t) & \text{otherwise.} \end{cases} \tag{1.2}$$

We define a candidate solution to the TD-TSPTW as a couple (t_0, p) where $t_0 \in [e_0, l_0]$ is the departure time from the origin vertex and p a permutation of vertex set $\mathcal{V}$ starting from vertex 0 and ending on vertex n. We note $p[i]$ the i^{th} vertex of p. Given a vertex index $i \in [0; n]$ and a candidate solution (t_0, p), we note $vt(t_0, p, i)$ the visit time at the i^{th} vertex of path p that departs from the origin vertex 0 at time t_0, and recursively define it as:

$$vt(t_0, p, i) = \begin{cases} t_0 & \text{if } i = 0 \\ a_{p[i-1],p[i]}(vt(t_0, p, i-1))_{\uparrow [e_{p[i]}, l_{p[i]}]} & \text{otherwise.} \end{cases} \tag{1.3}$$

A candidate solution (t_0, p) is feasible if it respects TW constraints at each visited vertex, *i.e.,* if $vt(t_0, p, n) \leq l_n$ (given $t_{\uparrow [e_i, l_i]} = \infty$ whenever $t > l_i$).

Objective functions. When considering the general case of seeking a Hamiltonian path of minimal duration, the objective value of a feasible solution (t_0, p) is the difference between the visit time at the destination vertex n and the departure time from the origin vertex 0, *i.e.,* $vt(t_0, p, n) - t_0$. This criterion is known as the *duration* objective, and we refer to this variant of the problem as the TD-TSPTW$_d$. Solving this problem requires finding a feasible solution of minimal duration (*i.e.,* both a starting time t_0 and a Hamiltonian path p) or proving that no such solution exists.

Another objective is the *travel time* objective: as its name implies, it consists in minimizing the sum of travel times, *i.e.,* waiting times are excluded from the objective function. More formally, the travel time of a feasible solution (t_0, p) is defined as: $\sum_{i=0}^{n-1} c_{p[i],p[i+1]}(vt(t_0, p, i))$. We refer to this variant of the problem as the TD-TSPTW$_{\Sigma t}$.

Another common objective criterion is the *makespan* objective, for which the goal is to minimize the arrival time at the destination vertex n, *i.e.,* $vt(t_0, p, n)$. When TD travel time functions satisfy the FIFO property, this implies that $t_0 = e_0$, given delaying departure from the origin vertex cannot allow one to arrive sooner at the destination vertex. In this thesis, we focus on the makespan objective and refer to this variant of the problem as the TD-TSPTW$_m$. The TD-TSPTW$_m$ can be seen as a specific case of the TD-TSPTW$_d$ in which the departure time from the origin vertex is fixed (*i.e.,* $e_0 = l_0 = t_0$).

Special cases. In the absence of TWs (*i.e.,* when $e_i = 0$ and $l_i = \infty, \forall i \in \mathcal{V}$), the problem is called TD-TSP$_d$ and it is equivalent to the TD-TSP$_{\Sigma t}$. Moreover, when the departure time from the origin vertex is fixed (*i.e.,* when $e_0 = l_0$), the TD-TSP$_d$ becomes equivalent to the TD-TSP$_m$.

In the case where travel times $c_{i,j}$ are constant, the TD-TSP and the TD-TSPTW respectively correspond to the TSP and the TSPTW. In the TSPTW, the duration and makespan objectives are analogous to their time-dependent counterparts, and are sometimes respectively referred to as the Minimum Tour Completion Problem (MTCP) and the Minimum Tour Duration Problem (MTDP). Note that the TSPTW$_{\Sigma t}$ – contrarily to the TD-TSPTW$_{\Sigma t}$ – does not require optimizing the departure time from the origin vertex (*i.e.,* $t_0 = e_0$), given a later starting time t_0 cannot lead to a shorter overall travel time. We summarize in Table 1.1 the main variants of the TD-TSPTW and the differences between these problems.

Problem	Time-dependent travel times	TW constraints	Variable starting time	Objective function
TD-TSPTW$_m$	✓	✓		makespan
TD-TSPTW$_d$	✓	✓	✓	duration
TD-TSPTW$_{\Sigma t}$	✓	✓	✓	travel time
TD-TSP$_m$	✓			makespan
TD-TSP$_d$	✓		✓	duration $\iff$ travel time
TSPTW$_m$		✓		makespan
TSPTW$_d$		✓	✓	duration
TSPTW$_{\Sigma t}$		✓		travel time
TSP				makespan $\iff$ duration $\iff$ travel time

Table 1.1: Summary of TD-TSPTW variants and special cases. Starting time t_0 is either variable (*i.e.,* $t_0 \in [e_0, l_0]$) or fixed (*i.e.,* $t_0 = e_0$).

As we shall see in Section 1.3, precedence relations between vertices can often be inferred in routing problems with TWs. They are typically exploited in solvers to discard partial solutions that cannot lead to feasible solutions due to TW constraints. The TSPTW is therefore related to the Sequential Orienteering Problem (SOP) [Asc+93], in which the set of feasible solutions is constrained by precedence relations between vertices instead of TWs.

Also note that the TSPTW$_m$ is equivalent to the single-machine scheduling problem with task-dependent setup times, release dates and due dates (denoted $1|r_j\overline{d_j}|C_{\max}$ in the notation of [Gra+79]).

Finally, it is worth mentionning that solving the TD-TSPTW is not only useful for finding optimal Hamiltonian paths in an urban context: for instance, a problem consisting in planning the actions of satellites observing the Earth has been formulated as a TD-TSPTW$_m$ in [Pra23].

Complexity. The decision variant of the TSP, *i.e.,* determining whether or not a tour no longer than $L \in \mathbb{N}$ exists, is an NP-Complete problem given it is more general than the Hamiltonian Cycle problem, one of Karp's twenty-one original NP-Complete problems [Kar72]. Consequently, the associated optimization problem is NP-Hard, similarly to each generalization of the TSP discussed above. Moreover, [Sav85] demonstrated that finding a feasible solution for the TSPTW (without any bound on the tour length) is a strongly NP-Complete problem, even in the special case where costs are symmetric.

Note however that the TD-TSPTW$_m$ may be solved in polynomial time when, given a constant k, at most k TWs overlap during the time horizon. In this case, the original problem instance of size n may be split into at most $\frac{n}{k}$ subproblems of size $n' \leq k$, which can be solved sequentially *e.g.,* in $O\left(k^2 \cdot 2^k \cdot \frac{n}{k}\right)$ time using Dynamic Programming (we describe how this paradigm may be used to solve the TSP and some of its variants in Chapter 2). Such algorithms are called fixed-parameter tractable (FPT) and are studied in the context of scheduling problems with TWs in [HM23].

Discussion. We have seen that the TD-TSPTW generalizes multiple classic problems, and that it has been considered with several objective functions. The difficulty of this problem stems from the fact that the number of possible Hamiltonian paths grows exponentially with respect to the number of vertices to visit. The presence of TWs however restricts the set of feasible solutions, but the size of this set is further increased when the starting time from the origin vertex needs to be optimized, *e.g.,* in the TD-TSPTW$_d$.

Routing problems constrained with TWs may have no feasible solution: in this case, one can consider *flexible TWs* [FA20], which consist in allowing TW violations to a certain extent and trying to minimize both the travel costs and the penalties incurred by these violations. *Orienteering problems* with TWs [TPF23] do not require all customers to be visited: resolution therefore consists in selecting a subset of customers to visit and finding the associated optimal visit order, while respecting TW constraints and maximizing rewards associated to each visited customer. [Kho+22] studied the TD Orienteering problem with TWs and *TD profits*, in which the reward collected at each customer is proportional to the duration of the visit.

Yet another option is to consider more general Vehicle Routing Problems (VRPs), in which planning is done for a fleet of vehicles instead of a single vehicle. In the TD-VRPTW [MD92], a demand is associated to each customer and vehicles have a fixed capacity: one must plan a route for each vehicle such that (i) all customers are visited within their TW, (ii) capacity constraints are respected, and (iii) the sum of each vehicle's makespan is minimal.

1.2 On Time Dependent travel time functions

In this thesis, we focus on solving the TD-TSPTW$_m$ and assume the availability of TD travel time functions $c_{i,j}(t)$ providing – for each pair of distinct vertices $(i,j) \in \mathcal{V}^2$ – the shortest travel time from vertex i to j when departing from i at time t.

Obtaining and modeling these functions is a research topic in itself. Nonetheless, we briefly describe how these travel time functions can be obtained from real-world traffic data, given it has been shown that finer-grained data lead to higher quality results [RCS20], but potentially to harder problem instances. Note that these considerations seem to rarely be taken into account, as many solvers are evaluated on purely artificial data which – as we shall see in Section 1.5 – sometimes make unrealistic assumptions.

1.2.1 Devising time-dependent travel time functions from realistic data

Customer graph and road network. Recall that a TSP instance is defined by a complete graph on a set of vertices $\mathcal{V}$. This graph is called a *customer graph*, as its vertices either represent customers or the starting and ending locations of the Hamiltonian path. When vertices of a TSP instance represent physical locations, travel costs may, for example, represent the distance, travel time, or the fuel consumption when traveling between pairs of vertices. In this case, travel costs in the customer graph can be obtained by computing optimal paths in the underlying road network.

More precisely, the *road network* is a directed graph $G_{\text{RN}} = (\mathcal{V}_{\text{RN}}, \mathcal{E}_{\text{RN}}, c_{\text{RN}})$ in which $\mathcal{E}_{\text{RN}}$ is the set of (oriented) road segments, $\mathcal{V}_{\text{RN}}$ the set of road segment endpoints, and $c_{\text{RN}} : \mathcal{E} \to \mathbb{N}$ the cost of traveling on each of these segments. Given a set of vertices to visit $\mathcal{V} \subseteq \mathcal{V}_{\text{RN}}$, the travel cost between two vertices $\{i,j\} \subseteq \mathcal{V}$ is obtained by computing an optimal path from i to j in G_{RN}. This can be achieved in polynomial time using classic shortest path algorithms.

The customer graph is therefore an abstraction of the road network which allows one, when tackling the TSP, to focus on finding an optimal visit order.

Time-dependent travel times in the road network. A similar idea is used to model time-dependent variants of the TSP, although determining time-dependent travel times in the customer graph requires knowing time-dependent travel times on each arc of the road network.

In the real world, these time-dependent travel times can be estimated from physical sensors that measure the flow and density of traffic on each individual road segment. Figure 1.2 illustrates the distribution of these sensors in part of the road network of Lyon, France: notice that the sensors' spatial coverage is relatively low – less than 8% of road links are equipped with sensors – and traffic information is therefore unavailable for entire neighborhoods.

In [ALS15], this lack of data was compensated by interpolating sensor information between neighboring links, taking into account streets directions. In [RCS20], authors went further and

Figure 1.2: Spatial distribution of traffic sensors on the road network of Villeurbanne and two districts (the third and the sixth) of Lyon (figure courtesy of [RCS20]). Yellow dots represent the locations of traffic sensors.

estimated missing information through fine-grained simulation of traffic flows, based on the available sensor data and estimations of transportation demands in the area.

Once the road network time-dependent travel times c_{RN} are known, travel times in the customer graph of a TD-TSP instance are obtained – as we have seen for the TSP – by computing optimal paths between each relevant pair of vertices in the road network. [ALS15] generalized this step to time-dependent travel times by computing time-dependent *fastest paths* for each pair of locations and each possible departure time. This allows one to model the fact that the fastest path may depend on the departure time, *i.e.,* that path optimality may depend on traffic conditions.

These shortest paths can be computed efficiently (*i.e.,* in polynomial time) using an adaptation of Dijkstra's algorithm (*e.g.,* see [Agu16]), provided that time-dependent travel time functions satisfy the FIFO property. When this property is not verified, this problem becomes NP-Hard [KS93] as waiting for an arbitrary amount of time on vertices may be beneficial.

1.2.2 Modeling time-dependent travel time functions

In this section, we describe common ways to model time-dependent travel time functions, whether in the context of the customer graph or when considering a road network.

The two models we consider rely on piecewise constant functions defined over a time horizon $\mathcal{H} = [0, T[$ partitioned into K intervals (or time-steps) $\mathcal{K} = [0, K-1]$, *i.e.,*

$$\mathcal{H} = [0, T[= \bigcup_{k \in \mathcal{K}} [T_k, T_{k+1}[\tag{1.4}$$

We note $\kappa : \mathcal{H} \to \mathcal{K}$ the function that determines the time-step k containing a given time t. Without loss of generality, we assume that time-steps have uniform lengths, *i.e.,* $T_k = k * \frac{T}{K}$, which

allows one to compute κ in constant time, *i.e.*, $\kappa(t) = \left\lfloor \frac{t}{T} * K \right\rfloor$ (when it is not the case, it can be done efficiently through binary search, or by mapping each time to its associated time-step).

Piecewise constant travel time functions

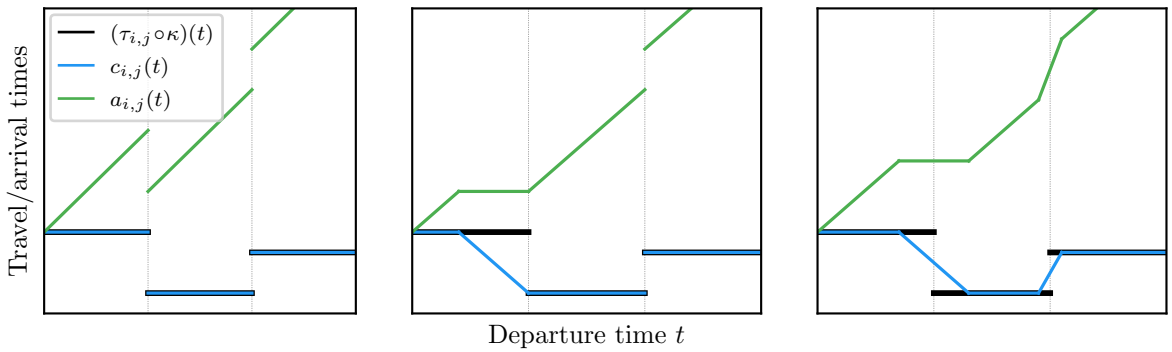
Early works on TD problems, whether in the context of shortest paths [CH66] or routing problems [Mal89], modeled TD travel time functions as piecewise constant functions $\tau_{i,j}$ associating a travel time to each time-step. This model has recently been used by [ALS15] and [RCS20], both for representing time-dependent travel times in the road network and in the customer graph.

Of course, as illustrated in Figure 1.3a, the TD travel time function $\tau_{i,j} \circ \kappa$ obtained from the piecewise constant function $\tau_{i,j}$ is likely not to verify the FIFO property due to discretization. A travel time function $c_{i,j}(t)$ satisfying the FIFO property is therefore obtained by applying transformations to $\tau_{i,j} \circ \kappa$ when necessary, *i.e.*, by replacing constant parts of these functions by linear pieces in order to guarantee a minimum slope of -1 in $c_{i,j}(t)$.

[MD92] obtains a travel time function $c_{i,j}$ verifying the FIFO property from $\tau_{i,j} \circ \kappa$ using a minimal set of changes: transitions in travel time between time-steps are smoothed only when the travel time decreases (*i.e.*, $\tau_{i,j}(k) > \tau_{i,j}(k+1)$) by considering a linear piece with a slope of -1 instead of the original constant piece during the last $\tau_{i,j}(k) - \tau_{i,j}(k+1)$ time units of time-step k, as illustrated in Figure 1.3b, *i.e.*,

$$c_{i,j}(t) = \begin{cases} \min(\tau_{i,j}(k), \tau_{i,j}(k+1) + T_{k+1} - t) & \text{if } \tau_{i,j}(k) > \tau_{i,j}(k+1) \\ \tau_{i,j}(k) & \text{otherwise.} \end{cases} \tag{1.5}$$

Where $k = \kappa(t)$ is the time-step containing time t, and $T_{k+1} - t$ denotes the time left before the beginning of time-step $k+1$.



(a) No transformation **(b)** Transformation from [MD92] **(c)** Transformation from [FGG04]

Figure 1.3: Illustration of FIFO transformations on a piecewise constant TD travel time function $\tau_{i,j}$ with three time-steps. Transitions between time-steps are represented by grey vertical lines; $\tau_{i,j} \circ \kappa$ is the original piecewise constant function, $c_{i,j}$ the TD travel time function obtained after transformation, and $a_{i,j}$ the arrival time function associated to $c_{i,j}$. Horizontal and vertical scales are identical.

[FGG04] later proposed an algorithm to make these functions both FIFO and continuous by smoothing transitions in travel times between each pair of contiguous time-steps, both on increases and on decreases in travel times, as illustrated in Figure 1.3c. It consists in using a linear piece with an arbitrarily large but finite slope when travel time increases, and a slope greater than or equal to -1 on decreases. Half of the linear piece from time-step k to $k+1$ lies on the current time-step k, and the other half on time-step $k+1$.

Piecewise constant travel speed functions

The *Ichoua, Gendreau, Potvin (IGP)* model [IGP03] allows one to compactly represent TD travel time functions that are both continuous and verify the FIFO property. A TD travel time function is computed from a constant travel distance $d_{i,j}$ and a piecewise constant TD speed function $v_{i,j}$ associating a travel speed to each time-step. In this model, travel may happen over multiple time-steps, and thus be performed at different speeds. Each unit of time traveling during time-step k allows one to reduce the remaining distance of $v_{i,j}(k)$ units. The IGP algorithm introduced in [IGP03] iteratively computes TD travel time $c_{i,j}(t)$ in linear time with respect to the number of time-steps, *i.e.,* in $O(K)$ time.

[Agu16] avoids running the IGP algorithm for each possible departure time by precomputing a piecewise linear travel time function. This allows one to compute TD travel times more efficiently, and it is achieved by determining a set of breakpoints between which travel times can be linearly interpolated.

Triangle inequality

Provided ideal TD travel time functions, triangle inequality holds in the customer graph. In practice, however, this property may not always be satisfied, *e.g.,* due to discretization and FIFO transformation when considering piecewise constant TD travel times, or because of rounding.

Although this property is trivial to ensure in the case of constant travel times (*i.e.,* by computing shortest paths between all pairs of vertices in polynomial time), it is not the case when considering TD travel times: it would indeed be expensive – both in space and time – to compute and memorize TD fastest paths between each pair of vertices and for each possible departure time. Another possibility consists in computing TD fastest paths on each query to the travel time function $c_{i,j}(t)$: this option does not require additional space but is likely to lead to redundant computations.

In the next section, we describe a set of rules commonly used to, amongst others, infer precedence relations between vertices in order to obtain a tighter but equivalent problem formulation (*i.e.,* containing the same set of feasible solutions). Some of these rules can be implemented efficiently provided triangle inequality holds: when it is not the case, they may lead to mistakenly excluding feasible solutions; we discuss this issue more in depth in Section 5.2.

Discussion

In this section, we have seen how travel time functions may be obtained from real world data that may, for instance, come from traffic data collected by physical sensors placed on road networks. We have also described two ways to model TD travel time functions efficiently both in terms of space and time, namely (i) a model based on piecewise constant travel times – including transformations to ensure the FIFO property – and (ii) the IGP model, which assumes piecewise constant travel speeds and constant distances.

The main motivation behind the IGP model is to provide continuous travel time functions, given that in reality, travel times fluctuate continuously instead of making discrete "jumps". One could argue that a piecewise constant function with a fine enough temporal granularity (*i.e.,* with a large enough number of time-steps) would tend to minimize the individual heights of these "jumps".

However, the IGP model assumes constant distances between vertices, which seems unrealistic when representing travel times in a customer graph: it indeed fails to model the fact that the sequence of road segments used to travel between vertices (*i.e.,* a fastest path in the underlying road network) may change according to traffic conditions. On the other hand, piecewise constant travel time functions are able to take this consideration into account, and therefore seems more suited than the IGP model to represent real-world data, as we shall see in Section 1.5 when describing classic TD-TSPTW$_m$ benchmarks. Note that in [Ben+21], authors managed to model such alternative paths by using the IGP model to represent TD travel times at the scale of the road network: the problem is also tackled at this scale, which tends to cause scalability issues in real-life networks.

1.3 Propagation of TW constraints

Most existing approaches for solving routing problems constrained with TWs use a set of preprocessing rules. This preprocessing aims to produce a "tighter" but equivalent formulation of the original problem, *i.e.,* both formulations contain the exact same sets of feasible solutions.

This is achieved using a set of local rules that tighten TWs [DDS92], recognize that some arcs cannot belong to a feasible solution [Lan+93] and infer precedence constraints between vertices [Des+95]. Their usefulness is studied in [AFG01] and they have been generalized to the TD-TSPTW in [MMM17].

These rules operate on two sets $\mathcal{E}$ and $\mathcal{R}$:

- $\mathcal{E}$ denotes the set of edges that can potentially be used to travel from a vertex to its successor in a feasible solution. We note $G_{\text{UA}} = (\mathcal{V}, \mathcal{E})$ the digraph of *usable arcs*. Initially, $\mathcal{E} = \{(0,i), (i,n), (i,j) : i,j \in \mathcal{C} \wedge i \neq j\}$.
- $\mathcal{R}$ is the set of precedence relations: it contains couples (i,j) such that vertex i must be visited before vertex j in a feasible solution (intermediate vertices may be visited between i and j). We note $G_{\text{PR}} = (\mathcal{V}, \mathcal{R})$ the *precedence digraph*. Initially, $\mathcal{R} = \{(0,i), (i,n) : i \in \mathcal{C}\}$.

These rules seek to (i) remove edges from $\mathcal{E}$, (ii) add edges to $\mathcal{R}$ and (iii) increase release times e_i and decrease deadlines l_i of vertices $i \in \mathcal{V}$. More precisely, these rules can remove an arc (i,j) from $\mathcal{E}$ when they manage to prove that all solutions visiting vertex j immediately after vertex i are infeasible. Similarly, they can add an arc (i,j) to $\mathcal{R}$ when they prove that all solutions visiting vertex j before vertex i are infeasible.

Arc removal and inference of precedence relations. A first set of rules is used to prove that some arcs cannot be used in a feasible solution (*i.e.,* by removing arcs from $\mathcal{E}$), or to infer precedence relations between vertices (*i.e.,* by adding arcs to $\mathcal{R}$). We name these two rules $\mathbf{PFR}_1$ and $\mathbf{PFR}_2$ given they are based on *path feasibility*, and define them as:

- $\mathbf{PFR}_1$: given a pair of distinct vertices $i,j \in \mathcal{V}$, if $a_{i,j}(e_i) > l_j$ then vertex j cannot be reached on time when departing as early as possible from vertex i: consequently, arc (i,j) is removed from $\mathcal{E}$ and arc (j,i) is added to $\mathcal{R}$.
- $\mathbf{PFR}_2$: a subpath $\langle i,j,k \rangle$ is infeasible if vertices j or k cannot be reached on time when departing as early as possible from i, *i.e.,* if $a_{i,j}(e_i) > l_j$ or $a_{j,k}(\max\{e_j, a_{i,j}(e_i)\}) > l_k$. Given a pair of distinct vertices $i,j \in \mathcal{V}$, if there exists a vertex $k \in \mathcal{V} \setminus \{i,j\}$ such that subpaths $\langle i,j,k \rangle$ and $\langle k,i,j \rangle$ are both infeasible, then arc (i,j) is removed from $\mathcal{E}$. If subpath $\langle i,k,j \rangle$ is also infeasible, then arc (j,i) is added to $\mathcal{R}$.

The second set of rules ensures the transitive closure of $\mathcal{R}$ and exploits precedence relations in $\mathcal{R}$ to filter $\mathcal{E}$:

- Given three distinct vertices $i,j,k \in \mathcal{V}$, if $\{(i,j),(j,k)\} \subseteq \mathcal{R}$, then arc (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 $\mathcal{E}$.

Tightening of TWs. The last set of rules is used to tighten TWs by trying to increase the release time e_i and to decrease the deadline l_i of vertices $i \in \mathcal{V}$, based on (i) TWs of other vertices, (ii) travel times, and (iii) the set of usable arcs $\mathcal{E}$. We name these *Tightening Rules* $\mathbf{TR}_1$ through $\mathbf{TR}_4$, illustrate them in Figure 1.4, and define them as:

- $\mathbf{TR}_1$: if a vertex $k \in \mathcal{V} \setminus \{0\}$ can only be reached past its release time e_k when departing as early as possible from each of its predecessors, then e_k can be increased, *i.e.,*

$$e_k \leftarrow \max\left\{e_k, \min_{(i,k)\in\mathcal{E}} a_{i,k}(e_i)\right\}, \forall k \in \mathcal{V} \setminus \{0\} \tag{1.6}$$

- $\mathbf{TR}_2$: if departing as early as possible from vertex $k \in \mathcal{V} \setminus \{n\}$ leads to arriving too early at each of its successors, then its release time e_k can be increased, *i.e.,*

$$e_k \leftarrow \max\left\{e_k, \min_{(k,i)\in\mathcal{E}} a^{-1}_{k,i}(e_i)\right\}, \forall k \in \mathcal{V} \setminus \{n\} \tag{1.7}$$

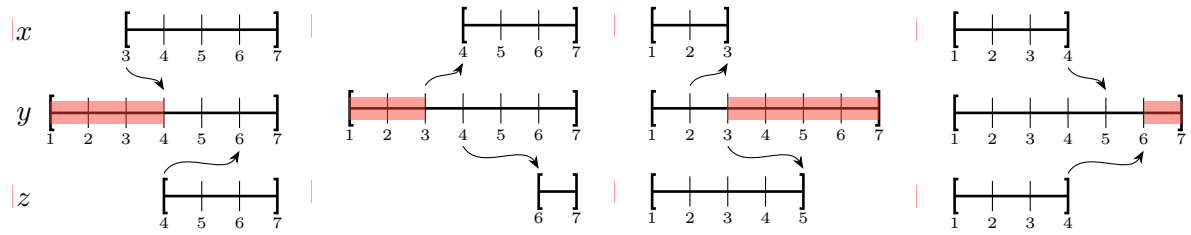
- $\mathbf{TR}_3$: if departing at late as possible from vertex $k \in \mathcal{V} \setminus \{n\}$ leads to arriving too late at each of its successors, then its deadline l_k can be decreased, *i.e.,*

$$l_k \leftarrow \min\left\{l_k, \max_{(k,i)\in\mathcal{E}} a^{-1}_{k,i}(l_i)\right\}, \forall k \in \mathcal{V} \setminus \{n\} \tag{1.8}$$

- $\mathbf{TR}_4$: if a vertex $k \in \mathcal{V} \setminus \{0\}$ can only be reached before its deadline l_k when departing as late as possible from each of its predecessors, then l_k can be decreased, *i.e.,*

$$l_k \leftarrow \min\left\{l_k, \max_{(i,k)\in\mathcal{E}} a_{i,k}(l_i)\right\}, \forall k \in \mathcal{V} \setminus \{0\} \tag{1.9}$$

All of these rules are applied until reaching a fixed point in which no more rule can be applied. They may prove that a given instance has no feasible solution, *e.g.,* when a vertex in $\mathcal{V} \setminus \{n\}$ has no outgoing edge in $\mathcal{E}$, when a vertex in $\mathcal{V} \setminus \{0\}$ has no incoming edge in $\mathcal{E}$, or when $\mathcal{R}$ contains a cycle.



(a) $\mathbf{TR}_1$: $e_y \leftarrow \min(a_{x,y}(e_x), a_{z,y}(e_z))$

(b) $\mathbf{TR}_2$: $e_y \leftarrow \min(a^{-1}_{y,x}(e_x), a^{-1}_{y,z}(e_z))$

(c) $\mathbf{TR}_3$: $l_y \leftarrow \max(a^{-1}_{y,x}(l_x), a^{-1}_{y,z}(l_z))$

(d) $\mathbf{TR}_4$: $l_y \leftarrow \max(a_{x,y}(l_x), a_{z,y}(l_z))$

Figure 1.4: Illustration of the four TW tigthening rules. TW of vertex y is tightened using TWs of x and z and constant travel times $c_{x,y} = c_{y,x} = 1$, $c_{y,z} = c_{z,y} = 2$. In **(a)** and **(d)**, y has two incoming arcs in $\mathcal{E}$: (x, y) and (z, y). In **(b)** and **(c)**, y has two outgoing arcs in $\mathcal{E}$: (y, x) and (y, z). Discarded parts of TWs are represented in red.

As we have seen in Section 1.2, TD travel time functions are imperfect and may not satisfy triangle inequality due to discretization or rounding of travel times. Also recall that it would be impractical to ensure it. Rules $\mathbf{PFR}_1$ and $\mathbf{PFR}_2$ assume that this property is satisfied: when it is not the case, they may erroneously rule out feasible solutions. We therefore provide an adaptation of these rules in Section 5.2.

1.4 State-of-the-art solving approaches

Since its introduction in 1992 by [MD92], different approaches have been proposed to solve the TD-TSP (or its variants) and a review may be found in [GGG15]. Many approaches are based on metaheuristics such as, for example, Ant Colony Optimization [Don+08], Tabu Search [IGP03; Gmi+21], or Large Neighborhood Search [Sun+20]. In this section, we focus on exact approaches, *i.e.,* approaches that are able to prove the optimality of the computed solutions.

Constraint Programming. [ALS15] 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.

Integer Linear Programming. State-of-the-art exact approaches for the TSP are usually based on ILP [Coo+11]. 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). [BS19] 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+20] for solving the TD-TSPTW under both the makespan and duration objectives. It performs best on instances with very tight TWs, and it dominates the Branch and Cut approach of [MMM17]. A stronger relaxation was proposed by [VHV22] to improve performance on the TD-TSPTW$_d$.

[CGG14] represents time-dependent travel times using the *Ichoua, Gendreau, Potvin (IGP)* model, introduced by [IGP03] and presented in Subsection 1.2.2. Recall that 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 travel speeds are time-dependent and are defined by piecewise-constant functions: during time-step $k \in \mathcal{K}$, the travel speed from i to j is constant and denoted v_{ijk}. Authors propose to decompose v_{ijk} in three factors $v_{ijk} = u_{ij} b_k \delta_{ijk}$, where:

- u_{ij} is the maximum travel speed from i to j during the whole time horizon (*i.e.,* $u_{ij} = \max_{k \in \mathcal{K}} v_{ijk}$),
- b_k is the best congestion factor over all arcs during time-step k (*i.e.,* $b_k = \max_{i,j \in \mathcal{V}} v_{ijk}/u_{ij}$),

- δ_{ijk} represents the degradation of the congestion factor of arc (i, j) during time-step k (*i.e.,* $\delta_{ijk} = v_{ijk}/(u_{ij}b_k)$).

A key parameter is Δ, the smallest value of all δ_{ijk} values (*i.e.,* $\Delta = \min_{i,j \in \mathcal{V}, k \in \mathcal{K}} \delta_{ijk}$): When $\Delta = 1$, all arcs (i, j) have the same congestion factor b_k for all time-steps $k \in \mathcal{K}$. In this case, the TD-TSP can be solved as a classic asymmetric TSP with constant travel times. When $\Delta < 1$, the optimal solution computed when assuming $\Delta = 1$ provides a lower bound which is used in the branch-and-bound algorithm of [Ari+18]. [Ari+19] tackles the TD-TSPTW$_m$ by enhancing this algorithm's branching strategy 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.

Dynamic Programming. The Dynamic Programming (DP) approach proposed by [Bel62] for the TSP has been extended to handle TD cost functions by [MD96] and TWs by [CMT81]. It has also been extended to *Vehicle Routing Problems* (VRPs) in [Hoo16] and to TD-VRPs in [RCS20]. Because the solving approach we propose for the TD-TSPTW$_m$ in Chapters 4 and 5 is based on DP, we only briefly list relevant approaches in this section and study DP more in depth in the context of the TSP in Chapter 2.

As the number of states explored by DP for the TSP is in $O(n \cdot 2^n)$, 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 [CMT81] for the TSPTW$_m$, [BMR12] for the TSPTW$_{\Sigma t}$, and [TI17] for the TSPTW$_d$. In this case, the optimal solution in the relaxed state space provides a lower bound. [LMS22] extended this approach to time-dependent travel time functions – considering both the makespan and duration objectives – and introduced the *ti-tour* relaxation which outperforms ILP approaches of [Ari+19] and [Vu+20]. It is not anytime as it does not yield approximate solutions during search: it either provides the optimal solution – given enough time and memory – or no solution at all.

Another possibility is to use *Restricted* DP (RDP), introduced by [MD96], 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.

More recently, [Pra23] proposed to solve the TD-TSPTW$_m$ by combining DP with Large Neighborhood Search (LNS). Starting from an initial solution that may violate constraints, it consists in iteratively performing a sequence of destroy and repair operations (*i.e.,* removing and re-inserting customers) to try to find better solutions. Solutions are repaired by exploring all possible reinsertions using DP, allowing one to obtain locally optimal paths from solutions in which customers have been removed. In the destroy phase, customer removals are constrained in such a way that the repair phase has bounded space and time complexities, exploiting the fact that hard TW constraints reduce the size of the DP search space. Diversity is increased through the use of perturbations and restarts. Note that this approach is not purely exact, although it manages to close instances with relatively tight TWs when it finds a solution whose cost is equal to a lower bound computed at the beginning of resolution. We nonetheless present this approach given it is

the state-of-the-art for quickly finding optimal or close-to-optimal solutions both when considering the TD-TSPTW$_m$ and the TSPTW$_m$.

Table 1.2 summarizes the approaches we experimentally compare to our solver in Chapter 7.

Ref.	Name	Type	Objective
[Ari+19]	ARI19	LP - Relaxation to the TSPTW$_m$	makespan
[Vu+20]	VU20	LP - Dynamic time discretization	makespan, duration
[LMS22]	LER22	DP - State Space Relaxations	makespan, duration
[Pra23]	PRA23	DP - Large Neighborhood Search	makespan

Table 1.2: TD-TSPTW$_m$ solvers experimentally evaluated in Chapter 7.

1.5 Classic benchmarks

In this section, we introduce benchmarks commonly used to evaluate recent TD-TSPTW$_m$ solving approaches. Because these benchmarks are typically based on randomly generated TD travel time functions, we highlight their weaknesses and also present the benchmark of [RCS20] which aims to provide realistic TD travel time functions.

The hardness of a TD-TSPTW instance depends on multiple factors, including the number of vertices $|\mathcal{V}|$, the tightness of TWs, as well as the granularity (*i.e.,* the number of time-steps $|\mathcal{K}|$) and variability of TD travel time functions.

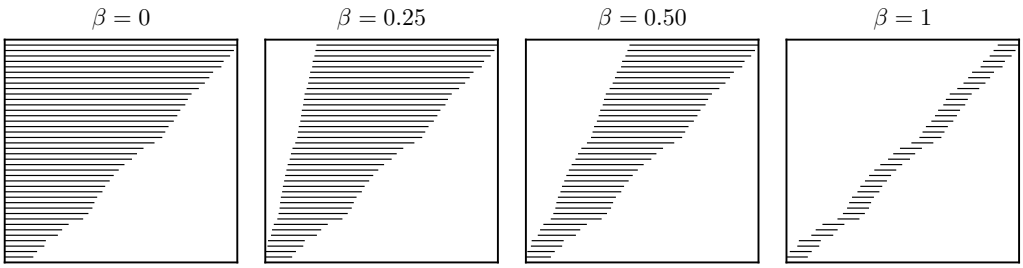
In order to compare the tightness of instances generated with different models, we compute the percentage of customer pairs that have *Overlapping TWs* and note it OTW, *i.e.,*

$$\text{OTW} = 100 * \frac{|\{\{i,j\} \subseteq \mathcal{C} : [e_i, l_i] \cap [e_j, l_j] \neq \emptyset\}|}{|\{\{i,j\} \subseteq \mathcal{C}\}|} \tag{1.10}$$

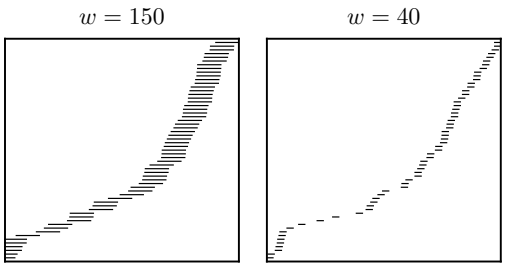
We compare the variability of TD travel time functions through value $0 \leq \Delta \leq 1$. Recall from Section 1.4 that the solving approach for the TD-TSPTW$_m$ of [Ari+19] manages to compute tight lower bounds when Δ is close to 1, *i.e.,* when TD travel speeds of all arcs vary rather uniformly throughout the time horizon.

Benchmark B_{ARI19}. This benchmark has been used by [Ari+19] to evaluate ARI19. It has been randomly generated according to the following parameters: the number of vertices $n \in \{16, 21, 31, 41\}$, the congestion factor $\Delta \in \{.70, .80, .90, .95, .98\}$, the traffic pattern $P \in \{B_1, B_2\}$ and the TW tightness $\beta \in \{0, 0.25, 0.50, 1\}$. There are 30 instances for each combination (n, β, Δ, P), leading to a total of 4800 instances.

TWs are generated in a way that guarantees instance feasibility: starting from a randomly generated tour in which vertex $i \in \mathcal{V}$ is visited at time v_i, TWs are placed around v_i and the release time e_i is reduced by a factor of β to widen TWs. More formally, TWs are obtained by setting $e_i = \max(0, \beta \cdot (v_i - 40))$ and $l_i = v_i + 40$, $\forall i \in \mathcal{V}$. Therefore, the smaller β, the wider



(a) Benchmark B_{Ari19} $(n = 41)$



(b) Benchmark B_{Vu20} $(n = 60)$

Figure 1.5: Distribution of customers' TWs on sample instances. The horizontal axis represents the time horizon; each horizontal line represents the TW of a customer. Customer TWs are presented, from bottom to top on the vertical axis, in increasing lexicographical order (l_i, e_i). We do not present a sample TW distribution for B_{Rif20} as TWs are generated using the same model as B_{Ari19}.

the TWs: the resulting layout is illustrated in Figure 1.5a on a sample instance, and the average OTW for instances with $\beta = 0$ (resp. 0.25, 0.50, and 1) is equal to 100 (resp. 90, 67, and 15).

TD travel times of this benchmark have been randomly generated according to a rather simple model: customers are randomly distributed in three concentric circular zones Z_1, Z_2, and Z_3. Euclidean distances between vertices in this plane are used together with TD travel speed functions to compute TD travel times using the IGP model described in Subsection 1.2.2: a single TD travel speed function $s_{Z_i}(k)$ is defined for each of the three zones, *i.e.,* all arcs originating from a vertex in zone z share the same travel speed function $s_z(k)$.

These three functions are obtained, for all instances, from a single and arbitrary piecewise constant function $s_{\text{base}}(k)$ composed of 73 time-steps by increasing congestion (*i.e.,* decreasing speeds) at certain times. In practice, congestion exclusively happens during morning and evening hours (*i.e.,* in the first 24 and last 25 time-steps, respectively), parameter P determines which two zones become congested, and parameter Δ defines the degree of congestion: if zone Z_1 is the most congested zone according to P, then $s_{Z_1}(k) = \delta_{Z_1}(k) \cdot s_{\text{base}}(k)$, where $\delta_{Z_1}(k) \in \{\Delta, 1\}, \forall k \in \mathcal{K}$. We illustrate in Figure 1.6 the TD travel speeds used in instance class $P = B_1, \Delta = 0.70$ (*i.e.,* instances with maximal congestion) and refer the reader to [Ari+19] for more details.

Some of the weaknesses of such TD travel time functions are inherent to the IGP model, *i.e.,* distances are assumed to be constant, which leads to assuming that the fastest path between two vertices in the road network does not change throughout the day. In this specific benchmark, (i) distances are symmetric, which disregards the potential presence of one-way streets and, more importantly, (ii) speeds vary rather homogeneously on all arcs throughout the planning horizon.

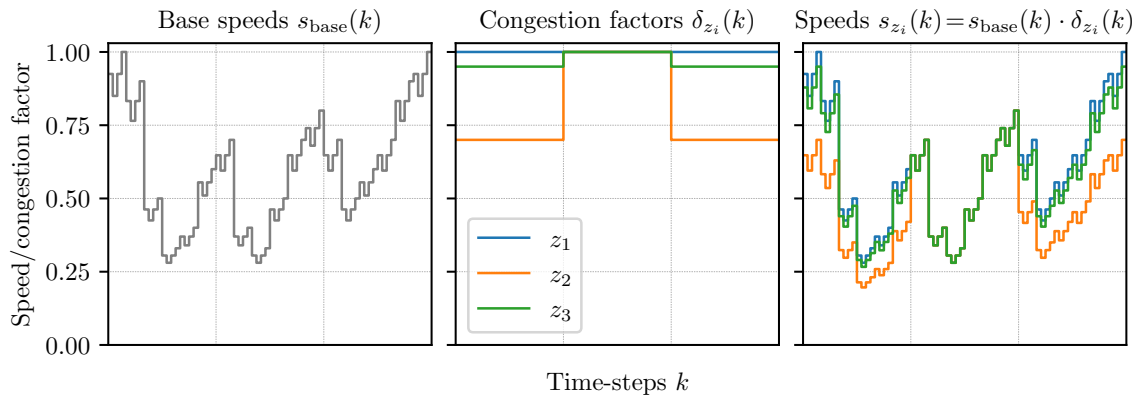


Figure 1.6: Travel time functions of B_{ARI19} instances when $P = B_1$ and $\Delta = 0.70$. The base travel speed function (left) is identical for all instances of B_{ARI19}; congestion factors (center) depend exclusively on P and Δ and are used to decrease travel speeds (right) in specific zones (here, z_2 and z_3) during the morning and evening hours.

Benchmark B_{VU20}. [Vu+20] introduced an extension of B_{ARI19} in which TD travel time functions are generated using the same model, but the number of vertices has been increased to $n \in \{60, 80, 100\}$ and only four values of Δ are considered, *i.e.,* $\Delta \in \{0.70, 0.80, 0.90, 0.98\}$.

Also, TWs are obtained in a different way: instead of determining TW widths with parameter β, TWs have a fixed width of w time units, where $w \in \{40, 60, 80, 100, 120, 150\}$, *i.e.,* parameter w is used to set $e_i = \max\left(0, v_i - \frac{w}{2}\right)$ and $l_i = v_i + \frac{w}{2}$, where v_i is the visit time at vertex i in a random tour. There are 10 instances for each combination of (n, w, Δ), leading to a total of 720 instances. As illustrated in Figure 1.5b, most TWs of this benchmark are much tighter than those of B_{ARI19}: 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}. In order to evaluate our approach on more realistic TD cost functions, we consider the benchmark introduced by [RCS20] and refer to it as B_{RIF20}. As we have seen in Section 1.2, 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 piecewise-constant TD cost functions are available: they depend on two parameters σ and l that respectively determine the spatial and temporal granularities of traffic data. In this thesis, we consider the finest granularities, *i.e.,* ideal spatial coverage ($\sigma = 100$) and 6-minute time-steps ($l = 6$). TD travel time functions are defined over a time horizon of twelve hours and are therefore composed of 120 time-steps.

Because we are interested in comparing the performance of various TD-TSPTW$_m$ solvers on benchmarks with both artificial and realistic TD travel time functions while keeping most other parameters equal, we do not use the original TWs of B_{RIF20} but instead generated new TWs using a model similar to the one used for B_{ARI19}. We therefore consider instances with $n \in \{21, 31, 41\}$ and TW tightness is controlled by $\beta \in \{0, 0.25, 0.50, 1\}$. 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. 0.25, 0.50, and 1) is equal to 100 (resp. 86, 62, and 16).

Note that, in B_{RIF20}, value Δ cannot be controlled. Moreover, unlike in B_{ARI19} and B_{VU20} where three rather similar speed functions are shared amongst all arcs, each arc has its own TD travel time function which may vary independently from the others: consequently, increasing the number of vertices also increases the number of distinct speed functions, which tends to decrease the value of Δ. In Figure 1.7, we illustrate this phenomenon and also show that Δ decreases when considering finer time-steps.

In the subset of instances we consider (*i.e.,* $|\mathcal{K}| = 120$ and $n \in \{21, 31, 41\}$), Δ is always smaller than 0.23 and has an average value of 0.09, which is significantly smaller than the lowest value of 0.70 considered in both B_{ARI19} and B_{VU20}. More generally, Figure 1.7 shows us that Δ is lower than 0.70 even for small instances with few time-steps.

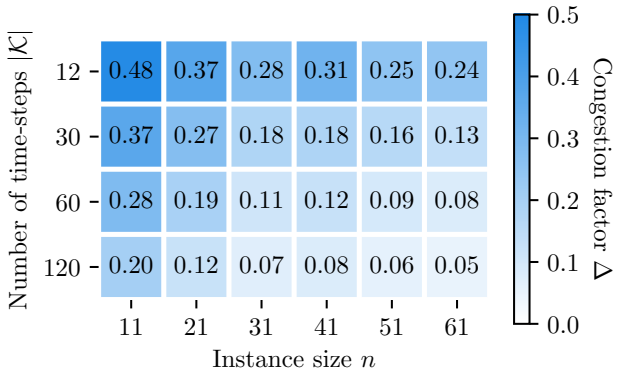


Figure 1.7: Mean Δ values on TD instances of B_{RIF20} with ideal spatial granularity (*i.e.,* , $\sigma = 100$ and $|\mathcal{K}| > 1$). Instances are grouped by size n and number of time-steps in TD travel time functions $|\mathcal{K}|$.

Other notable benchmarks. [Agu16] proposed a TD-TSPTW benchmark which consists in TD travel time functions composed of 6-minute time-steps that originate from the same sensor data as B_{RIF20}. [Agu16] estimated missing sensor information through interpolation, whereas B_{RIF20} aims to provide more realistic information by obtaining this data from a traffic simulator. Instances may contain extra precedence constraints, associate either one or two TWs to each vertex and sizes range in $n \in \{10, 20, 30, 50, 100\}$.

Original TWs of B_{RIF20} are obtained by dividing the time horizon into 2, 4, or 6 disjoint TWs and uniformly assigning them to customers [RCS20]. Unlike many TW generation models, TWs are not generated from the visit times in a TD-TSP solution and instance feasibility is not guaranteed. This model appears to be sensible when considering some real world use-case in which customers can choose a desired time range amongst a limited set of disjoint time slots.

More recently, [Pra23] introduced a TD-TSPTW$_m$ benchmark which does not consider vehicle routing on a road network, but instead consists in planning observations of satellites orbiting around the Earth. In this context, TD travel times and TWs depend on the altitude of satellites and on their (time-dependent) relative position to the observed areas.

Discussion. In this section, we have introduced TD benchmarks commonly used to evaluate TD-TSPTW solvers. Because of weaknesses in TD travel time functions in some of these benchmarks, we also consider an additional benchmark designed to provide realistic TD travel times. These benchmarks differ both in instance sizes, TW widths, and TD travel time functions: we provide a summary of their features in Table 1.3, and consider these three benchmarks when comparing TD-TSPTW$_m$ solvers in Chapter 7.

Note that TW generation schemes generally guarantee instance feasibility, but this assumption is unlikely to always hold when considering real-world problem instances. Solving infeasible instances require proving that no solution satisfying TW constraints exist. Such a proof can only be achieved by exact solving approaches: it may be trivial to obtain (*e.g.,* using the TW constraint propagation rules described in Section 1.3), or may instead require enumerating an exponential number of partial solutions.

Finally, we compared TD travel time functions through metric Δ, which has been used by [Ari+19] to generate artificial travel time functions for which relatively tight TD-TSPTW$_m$ lower bounds can be computed. This metric allows one to compare how homogeneously TD travel speeds vary over the time horizon. It however has limits: indeed, it considers that travel is possible on all arcs during the entire time horizon, failing to recognize that TWs reduce the set of times when travel is possible.

			TD travel times		Δ(%)			TWs	
Name	Ref.	Inst. sizes n	Model	$\lvert\mathcal{K}\rvert$	Min.	Mean	Max.	Width	$\overline{\text{OTW}}$ (%)
B_{Ari19}	[Ari+19]	$\{16, 21, 31, 41\}$	IGP	72	70	87	98	$\beta = 0$	100
								$\beta = 0.25$	90
								$\beta = 0.50$	67
								$\beta = 1$	15
B_{Vu20}	[Vu+20]	$\{60, 80, 100\}$	IGP	72	70	85	98	$w = 150$	20
								$w = 120$	16
								$w = 100$	14
								$w = 80$	11
								$w = 60$	8
								$w = 40$	5
B_{Rif20}	[RCS20]	$\{21, 31, 41\}$	Piecewise const.	120	1	9	23	$\beta = 0$	100
								$\beta = 0.25$	86
								$\beta = 0.50$	62
								$\beta = 1$	13

Table 1.3: Overview of the three TD-TSPTW benchmarks considered for experimental evaluation in Chapter 7. Column $\lvert\mathcal{K}\rvert$ contains the number of time-steps in time-dependent travel time functions. Column $\overline{\text{OTW}}$ contains the mean OTW over all instances, in percentage, for each benchmark and TW width.

1.6 Discussion

In this chapter, we have formalized the TD-TSPTW under different objective functions and described special cases of this problem. We then studied how realistic TD travel time functions may be obtained and modeled, and presented a set of rules commonly used to obtain a tighter problem formulation from travel times and TW constraints. We finally presented state-of-the-art solving approaches for the TD-TSPTW$_m$ and benchmarks commonly used to evaluate them.

In Chapter 2, we show how Dynamic Programming (DP) may be used to solve the TSP and its TD or TW-constrained counterparts. While DP is both exact and flexible, its scalability is limited: we therefore discuss of scalable algorithms that can be used to solve such planning problems in Chapter 3, before presenting our solving approach for the TD-TSPTW$_m$ in Chapters 4 and 5.

Chapter

2 Dynamic Programming

Contents

Dynamic Programming (DP) is an exact method that can be used to solve discrete optimization problems such as shortest path problems and routing problems, including the TD-TSPTW. Similarly to the Divide-and-Conquer paradigm, DP consists in (i) recursively decomposing the problem to solve into smaller subproblems, and (ii) combining their solutions to solve the original problem. Quick-Sort is an example of a Divide-and-Conquer sorting algorithm: it consists in partitioning the values to be sorted into two parts in such a way that the values of the first part are smaller than the values of the second part, before recursively sorting both parts separately.

DP however differs from Divide-and-Conquer because it exploits the fact that some subproblems overlap, *i.e.,* that the solution of a single subproblem is often a prerequisite to solve multiple larger subproblems. These redundant computations are avoided by memorizing the solution to each subproblem the first time it is solved, and reusing the memorized solution whenever the same subproblem is encountered again.

This chapter focuses on DP because this paradigm is used by multiple state-of-the-art solving approaches for the TD-TSPTW, including the one we propose in Chapters 4 and 5. Moreover, this paradigm has the advantages of being both exact and flexible, which may explain why original formulations of the TD-TSP and the TSPTW were based on DP.

We first describe a DP model to solve the TSP in Section 2.1. We then illustrate DP's flexibility in Section 2.2 by showing how it can be extended to handle generalizations of the TSP by considering time-dependent travel times and TW constraints. Because DP requires exponential space and time to solve these problems, we describe classic and problem-independent techniques used to improve its scalability in Section 2.3. We finally present recent declarative frameworks that are based on or borrow ideas from this paradigm in Section 2.4.

2.1 Dynamic Programming for the TSP

Recall from Chapter 1 that the Traveling Salesman Problem (TSP) is an NP-Hard discrete optimization problem in which the goal is to find a shortest tour that visits a set of locations and returns to its starting point. For self-containedness, we provide a TSP definition that is broad enough to later be extended to handle time-dependent travel times and time window constraints in Section 2.2.

A TSP instance is defined by a complete directed graph with vertices $\mathcal{V}$ and non-negative travel costs $c_{i,j}$ associated to each arc $(i,j) \in \mathcal{V}^2$, which are not necessarily symmetric. In vertex set $\mathcal{V} = \{0, \dots, n\}$, we distinguish two special vertices, 0 and n, which respectively denote the origin and destination vertices. In practice, 0 and n may refer to the same location: in this case, the origin vertex may be chosen arbitrarily, as solutions to the TSP are cycles. For convenience, we note $\mathcal{C}$ the set of customer vertices $\mathcal{C} = \mathcal{V} \setminus \{0, n\}$. A solution to this problem is a shortest path from 0 to n that visits each customer vertex exactly once, *i.e.,* a permutation of the customer set $\mathcal{C}$ leading to a Hamiltonian path of minimal cost.

A naive way to solve this problem is to perform an exhaustive search in $O(|\mathcal{C}|!)$ time, *i.e.,* to enumerate every permutation of the customer set $\mathcal{C}$ in order to find an optimal solution.

DP formulation. The DP formulation for solving the TSP was independently proposed in 1962 by Bellman [Bel62] and by Held and Karp [HK62], and is often referred to as the Held-Karp algorithm.

This formulation consists in defining a subproblem as finding a shortest path from the origin vertex 0 to a vertex $i \in \mathcal{V}$ while visiting exactly once each vertex of a set $\mathcal{S} \subseteq \mathcal{V}$. A subproblem is therefore defined as a couple $(i, \mathcal{S}) \in \mathcal{V} \times 2^{\mathcal{V}}$, such that $\{0, i\} \subseteq \mathcal{S}$. Subproblem $(i, \mathcal{S})$ is also called a *state*, and i and $\mathcal{S}$ are called *state variables*. We note $p(i, \mathcal{S})$ the *optimal value* associated to state $(i, \mathcal{S})$, *i.e.,* the cost of a shortest path from 0 to i that visits each vertex of $\mathcal{S}$ exactly once. Recall that solving the TSP requires determining the cost of a shortest path from vertex 0 to vertex n visiting each vertex of $\mathcal{V}$ once, *i.e.,* it requires computing $p(n, \mathcal{V})$.

Bellman equations or *recurrence equations* define how to recursively compute the optimal value associated to any given state. They exploit the fact that the TSP – using this state definition – has an optimal substructure, *i.e.,* a subproblem can be solved by combining the optimal solutions of simpler subproblems. These equations may be formulated as:

$$p(i,\mathcal{S}) = \begin{cases} c_{0,i} & \text{if } \mathcal{S} = \{0,i\} \quad (2.1a) \\ \min\limits_{j \in \mathcal{S}\setminus\{0,i\}} \left(p(j,\mathcal{S}\setminus\{i\}) + c_{j,i}\right) & \text{otherwise.} \quad (2.1b) \end{cases}$$

Equation 2.1a describes the base case of the recurrence relation by providing the value of trivial subproblems, *i.e.,* subproblems $(i,\mathcal{S})$ for which a single vertex has been visited after departing from the origin vertex (*i.e.,* $\mathcal{S} = \{0,i\}$). Optimal values associated to these subproblems are equal to the travel cost from the origin vertex 0 to a vertex $i \in \mathcal{V}$, *i.e.,* $c_{0,i}$.

Equation 2.1b – the induction case – solves a given subproblem by splitting it into smaller subproblems and combining their solutions. More precisely, it considers all possible ways to obtain an optimal path for a given state $(i,\mathcal{S})$ by decomposing it into $|\mathcal{S}| - 2$ smaller subproblems that end on a vertex $j \in \mathcal{S}\setminus\{0,i\}$ and visit intermediate vertices $\mathcal{S}\setminus\{i\}$. The solutions to these subproblems are first extended by using one more arc from j to i of cost $c_{j,i}$, and then combined by only retaining a least-cost alternative.

Resolution. A solving algorithm is obtained straightforwardly from the recurrence equations by using *memoization*, as described in Algorithm 2.1. This algorithm starts from the main problem $(n,\mathcal{V})$ and recursively breaks it down into smaller subproblems until reaching the base cases. Memoization consists in memorizing the value associated to each state the first time it is computed (Lines 2-4) and reusing this value in subsequent calls. Memoization is necessary in order to avoid performing redundant computations: it allows one to exploit the fact that subproblems overlap, *i.e.,* that the solution to a given subproblem is usually required to solve multiple larger subproblems.

Algorithm 2.1: Recursive computation of $p(i,\mathcal{S})$

input : A vertex $i \in \mathcal{V}$ and a subset of vertices $\mathcal{S} \subseteq \mathcal{V}$ such that $\{0,i\} \subseteq \mathcal{S}$
output : $p(i,\mathcal{S}) \in \mathbb{R}_0^+$, the cost of a shortest path from 0 to i visiting each vertex of $\mathcal{S}$ once
global variable : an initially empty map m associating a real value to every state $(i',\mathcal{S}')$ encountered during resolution

```
1 function p-rec(i, S):
2     if (i, S) ∉ m then
3         if S = {0, i} then set m(i, S) to c_{0,i};
4         else set m(i, S) to min_{j∈S\{0,i}} (p-rec(j, S \ {i}) + c_{j,i});
5     return m(i, S);
```

Relation to computing shortest paths. The recursive formulation of function p defines a weighted Directed Acyclic Graph (DAG), in which each node represents a state $(i, \mathcal{S})$. There exists an edge with weight $c_{i,i'}$ from subproblem $s = (i, \mathcal{S})$ to subproblem $s' = (i', \mathcal{S}')$ if subproblem s is a direct prerequisite to solve s', *i.e.,* iff `p-rec`$(i', \mathcal{S}')$ calls `p-rec`$(i, \mathcal{S})$.

This graph represents dependencies between subproblems, and also models weighted transitions between states. We can see the DP resolution of the TSP as a shortest path problem in this graph, provided we consider an extra initial state $s_0 = (0, \{0\})$ that represents the state in which no customer has yet been visited and the salesman is located at the origin vertex 0. This initial state is linked to the states defined in the base case of the recurrence equations. Solving this DP problem is then akin to finding a shortest path form initial state s_0 to final state $s_\mathrm{f} = (n, \mathcal{V})$ in this *state transition graph*.

More formally, we define this state transition graph as a layered and weighted DAG $G_\mathrm{ST} = (\mathcal{N}, \mathcal{A}, w)$. The set of states $\mathcal{N}$ is partitioned into $n+1$ layers (or *stages*): we note $\mathcal{N}^l$ the set of states that belong to layer $l \in [1, n+1]$. More precisely, set $\mathcal{N}^l$ contains all states $(i, \mathcal{S})$ such that $|\mathcal{S}| = l$, *i.e.,* states are grouped according to the number of visited vertices $|\mathcal{S}|$, and $\mathcal{N} = \bigcup_{l \in [1,n+1]} \mathcal{N}^l$.

The first and last layers each contain a single state, *i.e.,* $\mathcal{N}^1 = \{s_0\}$ and $\mathcal{N}^{n+1} = \{s_\mathrm{f}\}$. Intermediate layers $l \in [2, n]$ contain all states $(i, \mathcal{S})$ such that $i \in \mathcal{C}$, and $\mathcal{S}$ has cardinality l, is a subset of $\mathcal{V} \setminus \{n\}$ and contains both 0 and i, *i.e.,*

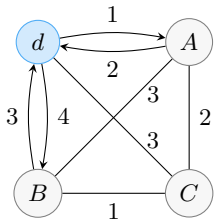
$$\mathcal{N}^l = \left\{(i, \mathcal{S}) \in \mathcal{C} \times 2^{\mathcal{V} \setminus \{n\}} : \{0, i\} \subseteq \mathcal{S} \wedge |\mathcal{S}| = l\right\}, \forall l \in [2, n] \tag{2.2}$$

Transitions only exist between states of consecutive layers: more specifically, given a layer $l \in [1, n]$, there exists an edge from $s = (i, \mathcal{S}) \in \mathcal{N}^l$ to $s' = (i', \mathcal{S}') \in \mathcal{N}^{l+1}$ with weight $w(s, s') = c_{i,i'}$ iff $\mathcal{S} \cup \{i'\} = \mathcal{S}'$, *i.e.,*

$$\mathcal{A} = \bigcup_{l \in [1,n]} \left\{((i, \mathcal{S}), (i', \mathcal{S}')) \in \mathcal{N}^l \times \mathcal{N}^{l+1} : \mathcal{S} \cup \{i'\} = \mathcal{S}'\right\} \tag{2.3}$$

Note that $p(i, \mathcal{S})$ denotes the cost of a shortest path from the initial state s_0 to $(i, \mathcal{S})$ in this DAG: consequently, solving the problem is equivalent to finding a shortest path from s_0 to $s_\mathrm{f} = (n, \mathcal{V})$ in G_ST. We illustrate in Figure 2.2 the state transition graph G_ST associated to the asymmetric TSP instance of Figure 2.1.

Also notice that the size of this state transition graph grows exponentially with respect to the number of vertices to visit, given states exist for each subset of $\mathcal{V}$. Algorithm 2.1 is just one way to compute such shortest paths; optimal values associated to each state may be computed in any topological ordering of G_ST. We study shortest path algorithms more in depth in Chapter 3, including algorithms designed for scalability in the context of limited space and time resources.



(a) Graph representation

$$c = \begin{array}{c} \\ d \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{cccc} d & A & B & C \end{array} \\ \begin{bmatrix} 0 & 1 & 4 & 3 \\ 2 & 0 & 3 & 2 \\ 3 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} \end{array}$$

(b) Cost matrix c

Figure 2.1: Sample TSP instance with three customer vertices $\mathcal{C} = \{A, B, C\}$. Vertex d represents both the origin and the destination. The optimal solution is $\langle d, A, C, B, d \rangle$ and has a cost of 7 units.

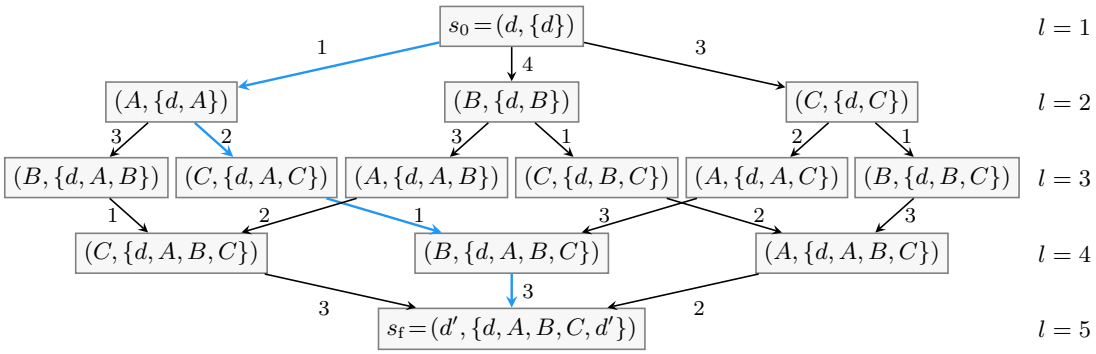


Figure 2.2: DP state transition graph for the asymmetric TSP instance of Figure 2.1. Vertices represent subproblems and arcs represent weighted transitions between states. Paths start from and end at vertex d; we refer to the destination vertex as d'. The shortest path from s_0 to s_f is represented in blue.

Computing optimal visit orders. For the sake of simplicity, we have so far focused on computing the cost of an optimal solution and disregarded the computation of an optimal vertex ordering. Algorithm 2.2 describes how to compute this ordering for any given state – after having computed its optimal value – by recursively tracing back the decisions that lead to it.

Algorithm 2.2: Recursive computation of an optimal path after resolution

input : A subproblem $(i, \mathcal{S})$ for which the solution needs to be computed
A map m associating to each state its optimal value
output : An optimal solution to subproblem $(i, \mathcal{S})$

```
1 function TSP-path(i, S):
2     if S = {0, i} then
3         return ⟨0, i⟩;
4     else
5         foreach vertex j ∈ S \ {0, i} do
6             if m(j, S \ {i}) + c_{j,i} = m(i, S) then
7                 return TSP-path(j, S \ {i}) + ⟨i⟩;
```

Complexity analysis. The space and time complexities of Algorithm 2.1 naturally flow from the state definition and the recurrence equations. Because $i \in \mathcal{V}$ and $\mathcal{S} \subseteq \mathcal{V}$, there are $O\big(|\mathcal{V}| \cdot 2^{|\mathcal{V}|}\big)$ possible states for which value $p(i, \mathcal{S})$ must be computed and memorized. The induction case of the recurrence relation combines the solutions of $|\mathcal{S}| - 2 \leq |\mathcal{V}|$ subproblems. Consequently, this algorithm requires $O\big(|\mathcal{V}| \cdot 2^{|\mathcal{V}|}\big)$ space and $O\Big(|\mathcal{V}|^2 \cdot 2^{|\mathcal{V}|}\Big)$ time.

Discussion. In this section, we have seen that DP is faster than exhaustive search to solve the TSP. These gains in running time is achieved by memorizing solutions to overlapping subproblems, and therefore comes at the expense of an exponential space complexity. Consequently, DP alone only allows one to solve relatively small problem instances: it fails to provide an optimal solution within reasonable space and time constraints otherwise. Note that DP is not an anytime approach (unlike, *e.g.,* Branch-and-Bound approaches): in other words, it does not output a sequence of solutions of increasing quality, but instead outputs a single optimal solution at the end of resolution.

We have also shown that solving this problem using DP is equivalent to computing shortest paths in a state transition graph; in Chapter 3, we discuss of shortest paths algorithms that aim to be more scalable by (i) avoiding to consider provably suboptimal paths, and (ii) by seeking to provide a sequence of solutions of increasing quality in order to provide the best possible solution under space and time constraints.

2.2 DP formulations for generalizations of the TSP

In this section, we illustrate DP's flexibility by adapting our model to solve generalizations of the TSP: we separately model the main variants of the TSP which are relevant to later tackle the TD-TSPTW$_m$ in Chapter 4. Similarly to the TSP, these variants can be expressed using only state variables $(i, \mathcal{S})$: we therefore model them by providing alternative definitions of $p(i, \mathcal{S})$.

2.2.1 Time-Dependent travel times

As presented in Chapter 1, the Time-Dependent TSP (TD-TSP) is a generalization of the TSP in which the travel time between vertices i and j is a function of the departure time from i, *i.e.,* $c_{i,j}(t)$ denotes the travel time from i to j when departing from i at time t. More precisely, we consider the TD-TSP$_m$ in which the departure time t_0 from the origin vertex 0 is fixed.

The DP model for solving the TD-TSP$_m$ was first introduced by [MD96]. When the FIFO property is satisfied (see Section 1.1), TD travel times are trivially integrated into the recurrence equations of the TSP. In this case, $p(i, \mathcal{S})$ denotes the earliest arrival time at vertex i when departing from the origin vertex 0 at time t_0 and visiting each vertex of $\mathcal{S}$ once. It may be defined as follows:

$$p(i, \mathcal{S}) = \begin{cases} t_0 + c_{0,i}(t_0) & \text{if } \mathcal{S} = \{0, i\} \quad (2.4a) \\ \min\limits_{j \in \mathcal{S} \setminus \{0,i\}} \left[p(j, \mathcal{S} \setminus \{i\}) + c_{j,i}(p(j, \mathcal{S} \setminus \{i\})) \right] & \text{otherwise.} \quad (2.4b) \end{cases}$$

When it is possible to wait before departing from a vertex and TD travel time functions do not satisfy the FIFO property, the principle of optimality does not hold [MD96] and these equations are no longer valid, given it may be possible to arrive sooner by waiting. In this thesis, we assume TD travel time functions satisfy the FIFO property.

2.2.2 Time Window constraints

Recall from Chapter 1 that the TSP with Time Windows (TSP-TW) generalizes the TSP by restricting the visit time at each vertex $i \in \mathcal{V}$ to belong to a time interval $[e_i; l_i]$. Also recall that we note $t_{\uparrow[e_i;l_i]}$ the TW-aware visiting time at vertex i, which includes a waiting time until e_i in case of early arrival (*i.e.,* $t < e_i$), returns infinity when arriving too late (*i.e.,* $t > l_i$), and returns t when arriving on time (*i.e.,* $e_i \leq t \leq l_i$).

In this thesis, we consider the *makespan* objective: the goal is to find a Hamiltonian path departing from the origin vertex 0 at time e_0 such that (i) TW constraints are satisfied and (ii) the arrival time at the destination vertex n is minimal. In this context, $p(i, \mathcal{S})$ denotes the earliest visit time at vertex i when departing from vertex 0 at time e_0 and visiting each vertex of $\mathcal{S}$ during its TW. The DP formulation for solving the TSPTW$_m$ was originally introduced by [CMT81] and consists in defining $p(i, \mathcal{S})$ as:

$$p(i,\mathcal{S}) = \begin{cases} (e_0 + c_{0,i})_{\uparrow[e_i,l_i]} & \text{if } \mathcal{S} = \{0,i\} \quad (2.5a) \\ \min\limits_{j \in \mathcal{S}\setminus\{0,i\}} (p(j, \mathcal{S}\setminus\{i\}) + c_{j,i})_{\uparrow[e_i,l_i]} & \text{otherwise.} \quad (2.5b) \end{cases}$$

The Bellman equations of the TSP are generalized relatively easily to the TSPTW$_m$ because TW constraints involve visiting times – *i.e.,* the objective function – and can be verified using state variable i.

They are however not as straightforward to extend to the TSPTW$_{\Sigma t}$, *i.e.,* the *travel time* objective, in which waiting times are excluded from the objective function. In this case, for any given state $(i, \mathcal{S})$, it is both desirable to minimize the arrival time – in order to maximize the slack time, *i.e.,* the time left to visit the remaining vertices $\mathcal{V} \setminus \mathcal{S}$ within their TWs – and the travel time. Greedily minimizing either of these objectives is not sufficient as they may be conflicting: it is therefore necessary to associate to each subproblem $(i, \mathcal{S})$ a set of Pareto-optimal solutions with respect to both these objectives. The recurrence equations to solve this problem involve an extra state variable t (see, *e.g.,* [BMR12]): in this case, the problem is solved by computing $\min_{t \in [e_n, l_n]} p(n, \mathcal{V}, t)$, where $p(i, \mathcal{S}, t)$ represents the minimum travel time to visit vertex i at time t when departing from the origin vertex 0 at time e_0 and visiting each vertex of $\mathcal{S}$ during its TW. A similar reasoning may be used to obtain a DP formulation for the TSPTW$_d$, in which the departure time from the origin vertex is a decision variable.

2.2.3 Precedence constraints

When considering routing problems in which visit times are constrained by TWs, precedence constraints between vertices can often be inferred from TWs and travel times, as we have seen in Section 1.3. More generally, some applications involve solving an asymmetric TSP with precedence constraints. This problem is called the Sequential Ordering Problem (SOP) [Asc+93]. These precedence constraints may be represented by a directed graph $G_{\text{PR}} = (\mathcal{V}, \mathcal{R})$ in which $(i, j) \in \mathcal{R}$ requires vertex i to be visited before vertex j. We assume that $\mathcal{R}$ is transitively closed, *i.e.,* $(i, j) \in \mathcal{R} \land (j, k) \in \mathcal{R} \implies (i, k) \in \mathcal{R}$.

We model this problem using DP by setting $p(i, \mathcal{S}) = \infty$ whenever constraints are violated, *i.e.,* when any of i's predecessors have not yet been visited. More formally, we note $pred(i)$ the set of predecessors of vertex i, *i.e.,* $pred(i) = \{j \in \mathcal{V} : (j, i) \in \mathcal{R}\}$, and set $p(i, \mathcal{S}) = \infty$ whenever $pred(i) \not\subseteq \mathcal{S}$.

2.2.4 Discussion

We have seen that it is relatively straightforward to extend the DP model of the TSP to handle generalizations of this problem when they can be expressed using only state variables i and $\mathcal{S}$. Indeed, time-dependent travel times are handled seamlessly and without increasing the size of the state space. Hard TW constraints are also straightforward to integrate: such hard constraints allow DP to solve larger problem instances since they typically reduce the number of states, given states which have no feasible solution can be discarded. On the other hand, note that modeling some TSP variants require using additional state variables, which tends to increase the size of the state space. We advise the reader to refer to [Hoo16; RCS20] for DP formulations of other TSP variants, including Vehicle Routing Problems.

In Chapter 4, we present a single DP model to solve the TD-TSPTW under the makespan objective, *i.e.,* a model that handles both time-dependent travel times and TW constraints. In the next section, we discuss of generic frameworks which leverage DP's flexibility and allow one to solve DP problems in a declarative way.

2.3 Improving DP's scalability

When considering NP-Hard optimization problems such as the TSP, DP does not manage to solve large problem instances due to its exponential space and time complexities. Richard Bellman coined this problem the *curse of dimensionality* [Bel57], which is sometimes referred to as *combinatorial explosion*. When considering the TSP, it is based on the observation that each additional customer to visit roughly doubles the time and space requirements for resolution.

In this section, we present general techniques which allow one to compute upper and lower bounds

using DP. We discuss bounding more in depth in Chapter 3 when tackling optimization problems by looking for optimal paths in state space graphs, and use it in our solving approach for the TD-TSPTW$_m$ in Chapter 4.

2.3.1 Restricted Dynamic Programming

Restricted Dynamic Programming (RDP) has been introduced by [MD96] in order to find approximate solutions to Time-Dependent TSP instances too large to be solved optimally using DP, due to its exponential space and time requirements. RDP consists in limiting the number of states considered at each level of the state space to the $H \in \mathbb{N}^+$ most promising ones. In this application, authors consider as most promising the states with the lowest value, assuming they are more likely to lead to higher-quality solutions.

Of course, RDP is not exact and only allows one to compute an upper bound on the optimal solution cost, *i.e.,* it can provide suboptimal solutions as it typically discards (or *prunes*) some states. Note that RDP may fail to find a feasible solution in presence of constraints. More generally, RDP may be used to compute an upper bound on the cost of the shortest path from any state s to the final stat s_f in the state transition graph G_ST. Parameter H allows one to obtain a trade-off between computation time and solution quality: when $H = \infty$, RDP is equivalent to the optimal DP algorithm in which no states are discarded; when $H = 1$, it is purely greedy and has a quadratic time complexity.

Notice that the criterion used to rank the states from most to least promising is inexpensive to compute but relatively near-sighted: in Chapter 3, we present generalizations of this algorithm which prioritize states not only according to their value (*i.e.,* their *cost-so-far*) but also according to an estimate of the remaining cost to obtain a complete Hamiltonian path (*i.e.,* their estimated *cost-to-go* to reach the final state s_f).

2.3.2 Computing lower bounds through relaxed state spaces

State Space Relaxations (SSRs) [CMT81] consist in mapping the original state space to another state space of smaller size, in such a way that solving the relaxed problem provides lower bounds in the original problem. Recall from Section 1.4 that [LMS22] prunes the TD-TSPTW search space using lower bounds computed through SSRs.

This mapping is defined by a function translating the original state variables to one or more relaxed state variables. A single relaxed state typically represents several states in the original search space; there exist transitions between relaxed states whenever a transition exists between the original states they represent. When multiple transitions exist between relaxed states, only the one of minimal cost is retained.

Relaxed state spaces are often obtained by ignoring some constraints to make the problem easier

to solve. A relaxed state space overapproximates the set of feasible solutions in the original problem, *i.e.,* a feasible solution in the relaxed state space is not necessarily a feasible solution in the original state space, but the converse is true.

For example, a simple relaxation for the TSP would be the *n-path* relaxation [CMT81]: instead of considering the original state variables $(i, \mathcal{S})$, only the cardinality of set $\mathcal{S}$ is considered, *i.e.,* state variables $(i, |\mathcal{S}|)$ are used. Using this relaxed state definition, the optimal value associated to state $(i, |\mathcal{S}|)$ denotes the cost of the shortest path from 0 to i that visits $|\mathcal{S}|$ vertices, which relaxes the constraint that each customer must be visited exactly once. This relaxed state space is considerably smaller than the original one since it has a polynomial size of $O\left(|\mathcal{V}|^2\right)$. Of course, using a mapping that leads to a larger relaxed state space tends to lead to tighter bounds, but also to higher computational costs.

More generally, solving the problem in such a relaxed state space allows one to compute a lower bound on the cost of the shortest path from a given state s to the final state s_{f} in the state transition graph G_{ST}.

2.4 Frameworks related to DP

In this section, we describe two problem-independent frameworks used to solve optimization problems that are related to or based on Dynamic Programming. These frameworks are exact and achieve scalability by being anytime, *i.e.,* they provide a sequence of solutions of increasing quality before finding the optimal solution and proving its optimality. Intermediate solutions provide upper bounds on the cost of the optimal solution and are used together with lower bounds in order to prune the search space. These frameworks have been used to tackle routing problems and share similarities with our solving approach for the TD-TSPTW$_m$ described in Chapter 4.

2.4.1 Multivalued Decision Diagrams

[Ber+16] provides a review on Multivalued Decision Diagrams (MDDs), a general framework for solving combinatorial optimization problems, which resembles DP on several aspects (see [Hoo13] for an analysis of the relationship between these two paradigms). MDDs represent a problem as a set of variables, domains associated to each of these variables, as well as a set of constraints. Exact MDDs, similarly to DP, consist in compactly representing the set of feasible solutions to a given problem as a layered DAG. Each vertex of this DAG represents an assignment of values to a subset of variables such that constraints are respected. A value is assigned to a new variable at each layer of this graph. The first layer contains a single vertex – the *root* r – in which no variables have been assigned, and the last layer a single vertex t – the *target* – in which all variables have been assigned. The set of feasible solutions is therefore encoded by paths from vertex r to vertex t. An objective function can be handled by associating a weight to each arc of the DAG, which represents the influence of a given assignment on the value of the objective

function. This effectively encodes the set of optimal solutions as the set of optimal paths from r to t.

Similarly to DP, exact MDDs suffer from the curse of dimensionality when the size of the state space grows exponentially with respect to the problem instance size. Consequently, two MDDs variants are used: *Relaxed* MDDs are computed by merging states through a user-provided procedure in order to compute lower bounds by overapproximating the solution set. *Restricted* MDDs consist in limiting the number of states considered at each layer of the DAG, which result in an underapproximation of the solution set and therefore upper bounds. Note that computing Restricted MDDs is analogous to Restricted DP, and that the concept of Relaxed MDDs is similar to performing DP in a relaxed state space.

Exact, restricted and relaxed MDDs are then combined into a generic branch-and-bound solving algorithm which is both exact and anytime, *i.e.*, which outputs a sequence of solutions of increasing quality until proving optimality (given enough time and memory).

This approach was improved in [Gil+21] by computing new bounds (some of them being problem-specific), in [GS22] by using Large Neighborhood Search to try to improve the solutions found, and in [RCR23] by reusing MDDs generated in previous iterations of the algorithm. Results are presented for several problems – including the TSPTW – for each of these extensions.

2.4.2 Domain-Independent Dynamic Programming

[KB23a] recently introduced Domain-Independent Dynamic Programming (DIDP), a DP-based discrete optimization framework. As it names implies, it is a problem-agnostic approach to model and solve problems that can be formulated using DP. It aims to be declarative, *i.e.,* to decouple modeling and resolution, similarly to other model-based approaches such as Constraint Programming (CP) and Mixed-Integer Programming (MIP).

DIDP provides a new modeling language which allows one to define state variables, initial and target states, constraints, as well as transitions between states and their cost. The problem is solved by finding an optimal path in a state transition graph; this search may be guided by a user-provided function associating to each state a lower bound on the remaining cost to reach a target state. DIDP originally used a generalization of A* able to handle different cost algebras, outperforming MIP and CP models on various problems, including the $\text{TSPTW}_{\Sigma t}$ and the Capacitated Vehicle Routing Problem (*i.e.,* a routing problem with multiple vehicles and capacity constraints). The framework was also extended to solve these problems using exact and anytime search algorithms in [KB23c], and augmented with Large Neighborhood Search in order to converge faster towards quality solutions in [KB23b].

2.5 Discussion

In this chapter, we have seen that DP is a paradigm that can be used to solve discrete optimization problems such as the TSP faster than brute-force approaches: this gain in efficiency is achieved by recursively decomposing problems into simpler subproblems and combining their solutions, while taking advantage of the fact that subproblems often overlap. We also defined a state transition graph which allows one to see the DP resolution of the TSP as a shortest path problem. However, when tackling NP-Hard problems such as the TSP, DP does not scale well because it requires solving and memorizing the solution to a number of subproblems which grows exponentially with respect to the instance size. Consequently, DP fails to provide a solution on large enough instance sizes given reasonable space and time constraints.

To improve the scalability of DP models, we have studied general techniques to compute lower and upper bounds, which may be used – as we shall see in the next chapter – to prune the state space. We also illustrated the flexibility of DP by generalizing a TSP model to handle time-dependent travel times – without increasing the size of the search space – and hard TW constraints, which tend to decrease the size of the search space and thus to improve scalability. We then presented problem-independent frameworks related to DP, which achieve scalability by tackling problems in an anytime manner, *i.e.,* by first providing approximate solutions before finding an optimal solution and proving its optimality.

In Chapter 3, we study how discrete optimization problems such as the TSP can be solved by searching for optimal paths in state transition graphs: we first consider classic algorithms and then study Exact and Anytime Search (EAS) algorithms for improved scalability. In Chapter 4, we (i) present a DP model for the TD-TSPTW$_m$ and formalize its associated state transition graph, (ii) instantiate four EAS algorithms which seek to provide the best possible solution under time and memory constraints and (iii) describe three lower bounds for the TD-TSPTW$_m$, which are used by EAS algorithms to guide search and to prune the search space.

Chapter

3 Planning Problems

Contents

Planning is an important field of Artificial Intelligence (AI) which consists in, given a description of an environment, devising a sequence of actions to reach a goal. Many constrained optimization problems can be modeled as planning problems, *e.g.,* the Traveling Salesman Problem (TSP) and its variants. A planning problem can be solved by searching for a best path in its associated state transition graph, using classic algorithms such as Dijkstra's algorithm or A*.

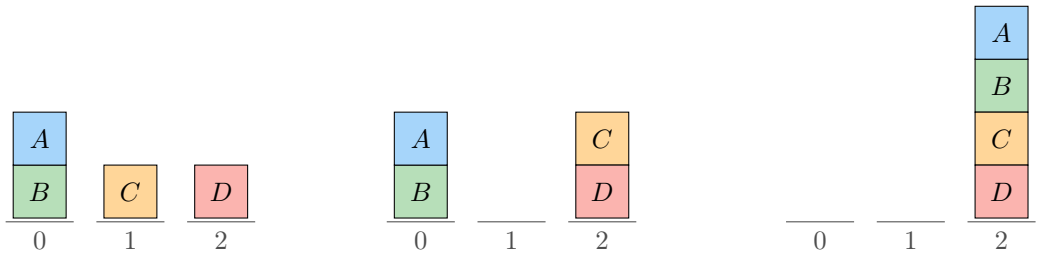
However, for some problems (*e.g.,* the TSP), the size of the state transition graph grows exponentially with respect to the instance size. In this context, classic shortest path algorithms require exponential amounts of time and memory and therefore do not scale well enough to solve large problem instances. We therefore consider *anytime* shortest paths algorithms: these algorithms seek to quickly find an approximate solution and keep searching for better solutions as long as space and time resources are available. We focus on anytime algorithms which are also *exact*, *i.e.,* which are able to prove that a given solution is optimal, or that no solution satisfying the constraints exists.

In this chapter, we first define planning problems and state transition graphs in Section 3.1. We then present uninformed graph search algorithms which are able to tackle small instances of these problems in Section 3.2. Finally, in Section 3.3, we consider (i) informed search and (ii) Exact and Anytime Search (EAS) algorithms, for improved scalability.

3.1 Definition of Planning Problems

Planning, *i.e.,* determining an action plan to reach one's goal, is a fundamental field of AI. Given a formal description of (i) the initial state, (ii) a goal state to reach, and (iii) the actions available in a given state as well as their outcome, a planner outputs a sequence of actions. This sequence of actions, when performed in an environment matching the initial state description, will change the environment to reach the goal state. One can seek any plan, *i.e.,* any sequence of actions leading to the goal, or an optimal plan according to some objective function (*e.g.,* the number of actions). This class of problems may be solved by finding an optimal path in a graph where nodes represent states, and edges between them represent actions.

We illustrate the main concepts studied in this chapter through examples on the *blocks world* problem: this problem consists in rearranging a set of n cubic blocks distributed on k stacks. Blocks are stacked vertically: a block is either located on top of a single other block, or at the bottom of a stack. A robot has the goal of changing the initial blocks configuration by moving them to reach an ideal configuration. The robot can only move a single block at a time, *i.e.,* it is only able to take a block located on the top of its stack, and has to immediately place it on top of a (possibly empty) different stack. In Figure 3.1, we illustrate a sample instance of this problem: Figure 3.1a introduces an arbitrary initial state. Figure 3.1b represents this state after moving block C, previously on top of stack 1, to the top of stack 2. Figure 3.1c represents the desired state in which all blocks are placed on stack 2 in alphabetical order, from top to bottom.



(a) Initial state s_0 **(b)** Intermediate state s_1 **(c)** Goal state s_f

Figure 3.1: Sample states of the blocks world problem with $n = 4$ blocks (A, B, C and D) and $k = 3$ stacks (0, 1 and 2).

Planning Problems. A planning problem is defined by a tuple $(\mathcal{S}_P, \mathcal{A}_P, s_0, s_f, F, \tau, c_P)$, where $\mathcal{S}_P$ and $\mathcal{A}_P$ respectively represent the set of states and the set of actions. States $s_0, s_f \in \mathcal{S}_P$ respectively denote the initial and final (also known as *terminal* or *goal*) states. Function $F : \mathcal{S}_P \to 2^{\mathcal{A}_P}$ associates to every state a set of feasible actions. Given a state $s \in \mathcal{S}_P \setminus s_f$ and a feasible action $a \in F(s)$, the transition function $\tau : \mathcal{S}_P \times \mathcal{A}_P \to \mathcal{S}_P$ determines the resulting state, and the cost function $c_P : \mathcal{S}_P \times \mathcal{A}_P \to \mathbb{R}_0^+$ determines the cost of this transition.

Without loss of generality, we assume (i) a single initial state s_0 and a single final state s_f, and (ii) that in any given state, each action leads to a different outcome, *i.e.,* $|\{\tau(s,a) : a \in F(s)\}| = |F(s)|, \forall s \in \mathcal{S}_P$.

In the blocks world, a state is defined as a permutation of n blocks, partitioned into k (possibly empty) subsequences. Because actions consist in moving a block from one stack to another, we define an action as a pair of stacks (i, j) such that $i, j \in [0, k-1]$ and $i \neq j$. A feasible action (i, j) consists in moving the top block of a non-empty stack i to the top of stack j.

In our example from Figure 3.1, six actions are feasible in state s_0 since no stack is empty. State s_1 is obtained from s_0 by performing action $(1, 2)$, *i.e.,* $\tau(s_0, (1, 2)) = s_1$. When considering state s_1 from Figure 3.1b, stack 1 is empty so only four actions are possible, *i.e.,* $F(s_1) = \{(0, 1), (0, 2), (2, 0), (2, 1)\}$.

A solution to a given planning problem is an ordered sequence of actions $\pi = \langle a_1, \ldots, a_k \rangle$ which leads to final state s_f when iteratively applied to the initial state s_0. Starting from s_0, the sequence of states obtained is $\langle s_1, \ldots, s_k = s_\mathrm{f} \rangle$, such that all actions are feasible (*i.e.,* $a_i \in F(s_{i-1}) \wedge s_i = \tau(s_{i-1}, a_i), \forall i \in [1, k]$). Solution π has length k. In this thesis, we assume that the cost of a solution is equal to the sum of the costs associated to each action taken, *i.e.,* $\sum_{i=1}^{k} c_P(s_{i-1}, a_i)$.

When considering our example from Figure 3.1, a sequence of actions allowing one to reach s_f from s_0 is $\pi = \langle (1, 2), (0, 1), (0, 2), (1, 2) \rangle$. This solution has a minimal length of four actions.

State transition graph. One may solve a given planning problem by looking for best paths in its associated *state transition graph* (or *state space graph*). More precisely, the state transition graph associated to a planning problem $P = (\mathcal{S}_P, \mathcal{A}_P, s_0, s_\mathrm{f}, F, \tau, c_P)$ is the weighted digraph $G^P_{\mathrm{ST}} = (\mathcal{N}, \mathcal{A}, w)$ such that $\mathcal{N} = \mathcal{S}_P$ and edge set $\mathcal{A} \subseteq \mathcal{N} \times \mathcal{N}$ represents directed transitions between states. Edge (s, s') belongs to $\mathcal{A}$ if and only if there exists an action $a \in F(s)$ such that $\tau(s, a) = s'$. The cost function c_P is used to associate a weight w to edge (s, s'), *i.e.,* $w(s, s') = c_P(s, a)$.

In the state transition graph G^P_{ST}, a solution $p = \langle s_0, \ldots, s_k = s_\mathrm{f} \rangle$ to problem P is a path from s_0 to s_f such that $(s_{i-1}, s_i) \in \mathcal{A}, \forall i \in [1, k]$. The cost of this path is the sum of its edges' weights, *i.e.,* $\sum_{i=1}^{k} w(s_{i-1}, s_i)$.

When considering the blocks world problem, actions are reversible: in a given state s, performing action (i, j) immediately after action (j, i) leads us back to state s. State transition graphs modelling instances of this problem are therefore not directed. Also note that the number of nodes in this graph grows exponentially with respect to the input parameters n and k.

We consider two main classes of planning problems:

- *Decision* planning problems are solved by finding a feasible solution, or by proving that no such solution exists. They can be solved by looking for a path from s_0 to s_f in a state transition graph.
- *Optimization* planning problems consist in finding an optimal solution, which can be achieved by looking for an optimal path from s_0 to s_f in a state transition graph. They are

a generalization of decision planning problems.

Note that finding a feasible solution can be difficult depending on the problem tackled: for instance, recall from Chapter 1 that it is trivial to find a solution to the TSP (any ordering of the customer set is a feasible solution), but it is NP-Complete when considering the TSPTW, *i.e.,* when hard constraints on visit times are associated to each customer.

Modeling the TSP as a planning problem. We may formulate the TSP as an optimization planning problem $P = (\mathcal{S}_P, \mathcal{A}_P, s_0, s_\mathrm{f}, F, \tau, c_P)$. Recall from the DP formulation of the TSP presented in Chapter 2 that (i) states are pairs $(i, \mathcal{S})$ such that $\{0, i\} \subseteq \mathcal{S} \subseteq \mathcal{V}$, (ii) the initial state is $s_0 = (0, \{0\})$ and (iii) the final state is $s_\mathrm{f} = (n, \mathcal{V})$. Therefore, the set of states is $\mathcal{S}_P \subseteq \mathcal{V} \times 2^\mathcal{V}$, and an action consists in visiting a yet unvisited vertex, *i.e.,* $\mathcal{A}_P = \mathcal{V} \setminus \{0\}$. More precisely, the destination vertex n can only be reached if all customer vertices $\mathcal{C}$ have been visited: therefore, the set of feasible actions given a state $(i, \mathcal{S})$ is $F(i, \mathcal{S}) = \{n\}$ when $\mathcal{C} = \mathcal{S} \setminus \{0\}$, and $F(i, \mathcal{S}) = \mathcal{C} \setminus \mathcal{S}$ otherwise. Given a state $s = (i, \mathcal{S})$ and a feasible action $a \in F(i, \mathcal{S})$, applying action a to state s has cost $c_P((i, \mathcal{S}), a) = c_{i,a}$ and leads to state $\tau((i, \mathcal{S}), a) = (a, \mathcal{S} \cup \{a\})$.

Problem P may be solved by finding an optimal path from s_0 to s_f in its associated state transition graph $G^P_\mathrm{ST} = (\mathcal{N}, \mathcal{A}, w)$. Note that G^P_ST is equivalent to the graph G_ST previously defined in Section 2.1.

3.2 Uninformed search algorithms

We first consider uninformed search algorithms which allow one to compute paths between the nodes of a graph. These algorithms are able to solve a planning problem P by finding a path between s_0 and s_f in its associated state transition graph G^P_ST. Recall that the size of this graph may grow exponentially with respect to the input parameters (*e.g.,* in the blocks world, $|\mathcal{N}|$ grows exponentially with respect to n and k) and may therefore not fit in memory. Also note that it may not be necessary to consider all nodes of the graph to solve the problem. Consequently, the graph is represented implicitly, as opposed to using an explicit representation such as an adjacency matrix or list. In other words, the graph is not stored in memory before starting the search, but it is computed gradually instead: initially, only one node is known, *i.e.,* the initial state s_0. Once a node is known, it may be *expanded*: expanding a node means generating all of its successors s' such that $(s, s') \in \mathcal{A}$ and memorizing those that are reached for the first time.

Search algorithms introduced below gradually explore the search space, expanding one node at a time, starting from the initial state s_0. During search, the set of nodes $\mathcal{N}$ is partitioned into three sets *open*, *closed* and *unseen*:

- The *open* set contains the nodes that are known but not yet expanded. This set represents candidate nodes for future expansions.
- The *closed* set contains the nodes that have already been expanded.

- The *unseen* set represents the remaining nodes, *i.e.,* $\mathcal{N} \setminus (open \cup closed)$. These nodes are not direct successors of *closed* nodes (*i.e.,* $s' \in closed \wedge s \in unseen \implies (s', s) \notin \mathcal{A}$). They are either reachable from *open* nodes (directly or indirectly), or not reachable from the initial state s_0.

Initially, we have $open = \{s_0\}$ and $closed = \emptyset$. Because *open*, *closed* and *unseen* form a partition of the set of states $\mathcal{N}$, we implicitly represent the *unseen* set, *i.e., open* and *closed* are memorized and used to determine whether or not a given node s belongs to *unseen* (*i.e.,* $s \in unseen \iff s \notin open \cup closed$). Also, when expanding node s, we compute all of its successors s' at once, given it may be expensive to compute its set of feasible actions $F(s)$.

Given a state $s \in \mathcal{N}$, we note $g^*(s)$ the cost of the shortest path from s_0 to s in G^P_{ST} (when no such path exists, $g^*(s) = \infty$). The cost of the optimal solution is therefore $g^*(s_\text{f})$. Also, we note $g(s)$ an upper bound on $g^*(s)$, *i.e.,* $g^*(s) \leq g(s) \leq \infty$. When computing shortest paths, $g(s)$ represents the cost of the best-known path to reach s from s_0 (*i.e.,* the best-known *cost-so-far* to reach s). For all nodes $s \in open \cup closed$, $g(s)$ is memorized explicitly. However, in the case of an *unseen* node s, $g(s)$ is equal to infinity (*i.e.,* no path from s_0 to s has yet been found) so this value is not memorized.

Algorithm 3.1 introduces a generic graph search algorithm. Initially, the *open* set only contains the initial state s_0, the *closed* set is empty, and $g(s_0)$ is set to 0 (Lines 2-5). At each iteration of loop L7-14, an *open* node s is expanded by (i) moving s from *open* to *closed*, and (ii) considering each of its *unseen* successors for later expansion by moving them to *open* and memorizing their g-value. This process is repeated until (i) the *open* set becomes empty, or (ii) the goal node s_f becomes *closed*. This algorithm proves that no path from s_0 to s_f exists if the *open* set becomes empty before finding s_f, *i.e.,* it proves that the problem as no solution.

At any given time, nodes belonging to $open \cup closed$ are necessarily reachable from s_0 (*i.e.,* $s \in open \cup closed \iff g(s) < \infty$). There is no such guarantee for *unseen* nodes since no path from s_0 to each node $s \in unseen$ has been discovered yet (*i.e.,* $s \in unseen \iff g(s) = \infty$). However, if such a path does exist, it necessarily goes through at least one node of *open*. Because *open* nodes are potential stepping stones for reaching *unseen* (or *undiscovered*) nodes from *closed* nodes, this set is often referred to as the *frontier* or *fringe*.

In order to be able to compute the resulting path, we use a map $\pi : \mathcal{N} \to \mathcal{N}$ which associates s to s' when first generating s' in order to remember that s leads to it (L14). Map π allows us to compute a path from s_0 to any node $s \in open \cup closed$ by recursively looking at its predecessor until reaching s_0 (see function `path` defined in Algorithm 3.1).

This algorithm is generic because it does not specify in which order *open* nodes should be expanded at L8. In the following sections, we introduce Depth-First Search and Breadth-First Search, two special cases of this algorithm obtained by defining this expansion order. Depending on this order and on the properties of the graph, sets *open* and *closed* may provide optimality guarantees for the g-values associated to the states they contain.

Algorithm 3.1: Generic graph search algorithm

```
1  function search(G^P_ST, s_0, s_f):
       input     : a state transition graph G^P_ST = (N, A, w) with initial state s_0 and final state s_f
       output    : a couple (c, p)
       postrel.  : if there exists a path from s_0 to s_f in G^P_ST, then p is a path from s_0 to s_f and c is the
                   cost of path p. Otherwise, p is an empty sequence and c = ∞.
2      let open be a set of states initialized to {s_0};
3      let closed be a set of states initialized to ∅;
4      let g be a map associating a real value to each state in open ∪ closed;
5      set g(s_0) to 0;
6      let π be a map associating a state to each state in (open ∪ closed) \ {s_0};
7      while open ≠ ∅ do
8          remove a state s from open and add it to closed;
9          if s = s_f then return (g(s), path(s, π, s_0));
10         foreach state s' such that (s, s') ∈ A do
11             if s' ∉ open ∪ closed then
12                 set g(s') to g(s) + w(s, s');
13                 add s' to open;
14                 set π(s') to s;
15     return (∞, ⟨⟩);
16 function path(s, π, s_0):
       input     : a state s, map π : N → N and initial state s_0
       precond.  : s is reachable from s_0, i.e., s ∈ open ∪ closed
       output    : a sequence of states ⟨s_0, s_1, ..., s_k⟩ such that s_k = s and s_i = π(s_{i+1}), ∀i ∈ [0, k − 1]
17     if s = s_0 then return ⟨s_0⟩;
18     return path(π(s), π, s_0)+⟨s⟩;
```

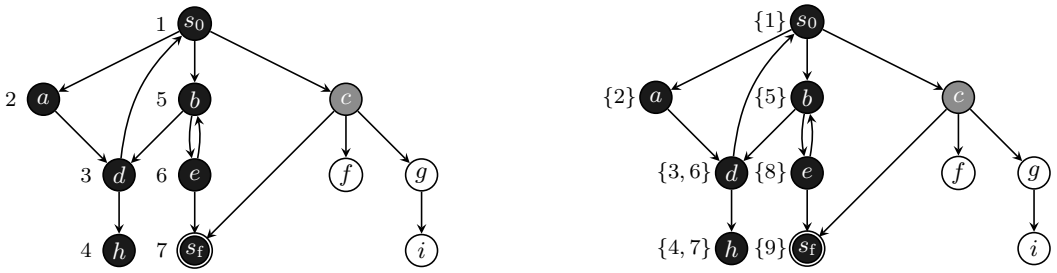
3.2.1 Depth-First Search

Depth-First Search (DFS) is a special case of the generic Algorithm 3.1 in which the *open* set behaves like a stack, *i.e.,* DFS iteratively expands the state most recently added to *open*. When considering a graph with unit weights, DFS expands, at each iteration, an *open* node with maximal g-value.

As its name implies, its strategy is to go as deep as possible in the graph. DFS recursively expands all *unseen* successors of a given node s, starting from the initial state s_0. When all successors of s have been expanded, DFS *backtracks* and recursively expands *open* successors of s's predecessor, *i.e.,* $\pi(s)$.

Figure 3.2a illustrates the behavior of DFS on an example graph. DFS first expands nodes s_0, a, d and h before reaching a dead-end. It therefore backtracks to node s_0 (*i.e.,* the most recently expanded node with *open* successors), and reaches the goal state through path $\langle s_0, b, e, s_f \rangle$.

This implementation of DFS performs *duplicate detection*, *i.e.,* it guarantees that each node is expanded at most once. This is achieved by memorizing both the *open* and *closed* sets in order to only add *unseen* nodes to the *open* set upon expansion. DFS with duplicate detection has a time complexity of $O(|\mathcal{N}| + |\mathcal{A}|)$ and a space complexity of $O(|\mathcal{N}|)$.



(a) With duplicate detection. **(b)** Without duplicate detection.

Figure 3.2: Illustration of two variants of DFS on a sample graph. Edges are considered from left to right. Nodes belonging to the *open*, *closed* and *unseen* sets are respectively represented in grey, black and white. Left: closed nodes are memorized, so each node is expanded at most once (the iteration in which a node is expanded is displayed on its left). Right: closed nodes are not memorized, so a given node i is expanded each time a new elementary path from s_0 to i is found (the set of iterations in which a node is expanded is displayed on its left). This graph has two elementary solutions $\langle s_0, b, e, s_f \rangle$ and $\langle s_0, c, s_f \rangle$.

Alternatively, DFS can be implemented without duplicate detection in order to reduce its space complexity. In this case, the *open* set is represented implicitly on the call stack, and the *closed* set, g-values and π-values are not memorized. This may lead DFS to expand multiple times the same state. In addition to the implicit *open* set, this implementation of DFS memorizes the current path in order to avoid expanding nodes that would introduce a cycle in it. This guarantees termination in finite graphs and allows one to compute the path found from s_0 to s_f (which was previously computed using π-values). Its space complexity is in $O(bd)$, where d is the depth of the graph (*i.e.,* the length in arcs of the longest elementary path from the root), and b is its branching factor (*i.e.,* the maximum number of direct successors of any node in the graph). Figure 3.2b illustrates the behavior of this variant of DFS and the associated node re-expansions: nodes d and h are expanded twice because (i) d can be reached both through nodes a and b from the initial state s_0, and (ii) h is a successor of d.

While performing duplicate detection is not necessary in the special case of trees, it may have a significant impact on time complexity in the general case, especially on graphs which contain many *redundant paths*, *i.e.,* when nodes can be reached from s_0 through multiple paths. Recall from Chapter 2 how using Dynamic Programming reduces the time complexity of solving a TSP instance with n customers from $O(n!)$ to $O\big(n^2 \cdot 2^n\big)$, at the expense of an exponential space complexity (*i.e.,* $O(n \cdot 2^n)$). In this case, not performing duplicate detection reduces the space complexity but cancels out the benefits of memoization, leading us back to a factortial time complexity.

3.2.2 Breadth-First Search

Breadth-First Search (BFS, also known as *uniform cost search*) is a graph search algorithm which computes shortest paths in uniformly weighted graphs (*i.e.*, $w(i,j) = c, \forall (i,j) \in \mathcal{A} \wedge c > 0$). Consequently, BFS solves optimization planning problems by providing an optimal solution with respect to the number of actions it contains. When considering BFS, we assume all arcs have unit weight, *i.e.*, $w(i,j) = 1, \forall (i,j) \in \mathcal{A}$. We shall relax this assumption in the next subsection, when considering Dijkstra's algorithm.

BFS is obtained from Algorithm 3.1 by expanding the state which was least recently added to *open*, *i.e.*, by representing the *open* set as a FIFO (First In, First Out) queue. Because weights are assumed to be uniform, BFS iteratively expands an *open* node with minimal g-value, until finding an optimal path to s_f.

Consequently, BFS guarantees that:

- all nodes located at a distance k of the initial state s_0 are expanded before any node located at a distance $k+1$ of s_0,
- the best-known path from s_0 to every discovered node is optimal, *i.e.*, $s \in open \cup closed \implies g(s) = g^*(s)$,
- at any given time, the difference in g-value between two *open* states never exceeds one (*i.e.*, $\max_{s \in open} g(s) - \min_{s \in open} g(s) \leq 1$).

Also, in BFS, we note $\underline{g}$ the minimum g-value of any state in *open*, *i.e.*, $\underline{g} = \min_{s \in open} \{g(s)\}$. Notice that, at the beginning of each iteration, $\underline{g}$ provides a lower bound on $g^*(s_\mathrm{f})$, the cost of the optimal path from s_0 to s_f.

Similarly to DFS with duplicate detection, BFS has a time complexity of $O(|\mathcal{N}| + |\mathcal{A}|)$ and a space complexity in $O(|\mathcal{N}|)$ as both *open* and *closed* sets must be memorized. The behavior of BFS is illustrated on a sample graph in Figure 3.3.

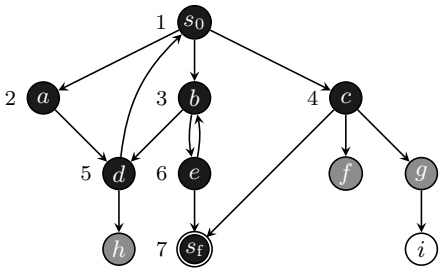


Figure 3.3: Illustration of BFS on a sample graph with uniform weights and two elementary solutions – $\langle s_0, c, s_\mathrm{f} \rangle$ and $\langle s_0, b, e, s_\mathrm{f} \rangle$ – of respective lengths 2 and 3. The iteration in which a closed node is expanded is displayed on its left. BFS finds the optimal solution $\langle s_0, c, s_\mathrm{f} \rangle$.

3.2.3 Dijkstra's shortest path algorithm

Dijkstra's algorithm was introduced in [Dij59] and generalizes BFS to graphs with non-uniform and non-negative weights. It is based on Bellman's optimality principle, *i.e.,* if the optimal path from node A to node C is known to go through node B, then the subpath from A to B is also optimal. Similarly to BFS, Dijkstra's algorithm expands at each iteration an *open* state s with minimal $g(s)$. However, because costs are not necessarily uniform, Dijkstra's algorithm (i) relies on a *priority queue* instead of a queue, and (ii) only guarantees that $g(s)$ – the cost of best-known path from s_0 to s – is optimal once s is *closed* (whereas BFS guarantees this for both *open* and *closed* nodes due to weight uniformity).

Because g-values are not guaranteed to be optimal for *open* states, the condition at Lines 11-14 of Algorithm 3.1 must be adapted. It is sufficient for DFS and BFS, because the former does not seek shortest paths, and the latter assumed unit weights. Dijkstra's algorithm however requires a more general *arc relaxation procedure*.

In the context of shortest paths algorithms, *relaxing* an arc $(i, j) \in \mathcal{A}$ means to consider using arc (i, j) to reach j from s_0, *i.e.,* it consists in trying to improve the current best-known path to j by appending arc (i, j) to the current best-known path to i. If the best-known cost-so-far to reach j is improved when going through i (*i.e.,* $g(j) > g(i) + w(i, j)$), then it is updated by setting $g(j)$ to $g(i) + w(i, j)$.

When expanding node s, Dijkstra's algorithm relaxes all arcs $(s, s') \in \mathcal{A}$ using Algorithm 3.2. Condition of Line 2 is always true when $s' \in \mathit{unseen}$ (given $g(s) = \infty, \forall s \in \mathit{unseen}$) and it is always false when $s' \in \mathit{closed}$ (because $s \in \mathit{closed} \implies g(s) = g^*(s)$). States $s' \in \mathit{open}$ are only considered if going through s leads to an improvement of the current best-known path to s'. At Lines 3-5, we update $g(s')$, ensure that s' belongs to *open*, and memorize that s led to s'.

Algorithm 3.2: Relaxation of arc (s, s') in Dijkstra's algorithm

```
1 procedure dijkstra-relax(s, s', w, open, g, π):
      input        : arc (s, s') ∈ A and weights w : A → ℝ₀⁺
      input/output : the open set, maps g : N → ℝ₀⁺ and π : N → N
2     if g(s) + w(s, s') < g(s') then
          // s' ∉ closed
3         set g(s') to g(s) + w(s, s');
4         ensure that s' belongs to the open set;
5         set π(s') to s;
```

Dijkstra's algorithm relies on a priority queue to represent the *open* set. This abstract data structure allows one to (i) insert new nodes, (ii) update the g-value of an existing node, (iii) extract a node with minimal g-value. These three operations are respectively known as *insert*, *decrease-key*, and *extract-min*.

Because Dijkstra's algorithm performs $O(|\mathcal{A}|)$ *insert* and *decrease-key* operations, and $O(|\mathcal{N}|)$ *extract-min* operations, the time complexity of this algorithm strongly depends on the priority

queue implementation. Common implementations consist in:

- Using a hash map associating to each state its (i) g-value and (ii) whether or not it belongs to *open*. The *insert* and *decrease-key* operations are performed in $O(1)$ time, but *extract-min* requires a linear search in $O(|\mathcal{N}|)$ time to find the *open* state with minimal g-value. In this case, Dijkstra's algorithm runs in $O\left(|\mathcal{A}| + |\mathcal{N}|^2\right) = O\left(|\mathcal{N}|^2\right)$ time.
- Using a *Binary Heap* [Wil64] allows one to perform all three operations in logarithmic time, which results in a $O((|\mathcal{A}| + |\mathcal{N}|) \log |\mathcal{N}|)$ time complexity.
- Using a *Fibonacci Heap* [FT87] also performs *extract-min* in logarithmic time, but handles *insert* and *decrease-key* operations in amortized constant time, leading to an overall time complexity of $O(|\mathcal{A}| + |\mathcal{N}| \log |\mathcal{N}|)$.

Note that, for both types of heaps, efficiently performing the *decrease-key* operation on arbitrary nodes requires knowing the position of each node in the heap. This is usually achieved by storing and maintaining this information in an associative data structure (*e.g.,* a hash map).

The first alternative should be preferred for *dense* graphs (*i.e.,* graphs for which $|\mathcal{A}|$ is close to $|\mathcal{N}|^2$). Otherwise, binary heaps are generally used in practice because of the constant factors and programming complexity of Fibonacci Heaps [Cor+09].

3.3 Informed search

Informed search algorithms use problem-specific information to guide the search towards the goal, *i.e.,* to avoid the exploration of unpromising regions of the search space. This information is integrated to graph search algorithms in the form of a *heuristic function* h. The heuristic value $h(s)$ of a state $s \in \mathcal{N}$ is an estimate of the cost of reaching the goal state s_f from s.

We first introduce heuristic functions and their main properties in Subsection 3.3.1 before presenting the A* algorithm – a generalization of Dijkstra's algorithm which uses a heuristic function to guide the search – in Subsection 3.3.2. Finally, Subsection 3.3.3, we discuss of exact and anytime search algorithms which (i) are related to A* and (ii) aim to improve its scalability.

3.3.1 Heuristic functions

A heuristic function $h : \mathcal{N} \to \mathbb{R}_0^+$ associates to each state $s \in \mathcal{N}$ an estimate of the cost of the shortest path from s to s_f. For convenience, we note $h^*(s)$ the *perfect heuristic* (or *optimal heuristic*) which returns the cost of the shortest path from s to the final state s_f, and $h^0(s)$ the *null heuristic* which always returns 0. Two main properties may be associated to heuristic functions: *admissibility* and *consistency*.

A heuristic function h is *admissible* if it never overestimates the cost of the shortest path between

s and the final state s_f, *i.e.,* $h(s) \leq h^*(s), \forall s \in \mathcal{N}$. Because of this relation to h^*, admissible heuristics are often referred to as *lower bounds*.

A heuristic function h is *consistent* or *monotone* when $h(s_f) = 0$ and $h(s) \leq w(s, s') + h(s')$ for all arcs $(s, s') \in \mathcal{A}$. All consistent heuristics are admissible, but the converse is not true (*e.g.,* setting $h(s_0) = h^*(s_0)$ and $h(s) = 0, \forall s \in \mathcal{N} \setminus \{s_0\}$ in a graph where weights are strictly positive, $g^*(s_f) < \infty$ and $(s_0, s_f) \notin \mathcal{A}$ leads to an admissible but inconsistent heuristic).

Admissible and consistent heuristics may be derived by solving a relaxed problem: this simpler problem is usually (i) less constrained and (ii) solved in polynomial time.

To illustrate the concept of admissible heuristic functions when searching, let us consider our example of the blocks world: we may define a heuristic function $h_1(s)$ which returns the number of misplaced blocks in s, *i.e.,* the number of blocks which are not on the same stack as in the goal state. This heuristic function is a lower bound on $h^*(s)$ because (i) at least one action is required to move each of these misplaced blocks, and (ii) the ordering constraint on the stacks of the goal state is relaxed. When considering our sample states from Figure 3.1, h_1-values for states s_0, s_1 and s_f are respectively 3, 2 and 0 (their respective h^*-values are 4, 3 and 0).

3.3.2 A* algorithm

A* is an *informed search* algorithm introduced by [HNR68]. It belongs to the family of Best-First Search algorithms, as it iteratively expands the most promising *open* node and thus tries to avoid exploring unpromising regions of the graph. A* uses a *guide* (or *evaluation* function) f to rank nodes from most to least promising. The guide function is defined as $f(s) = g(s) + h(s)$, where $g(s)$ is the best-known cost-so-far from s_0 to s, and $h(s)$ is an admissible heuristic function which estimates the cost-to-go to reach s_f from node s. Provided an admissible heuristic h, A* is guaranteed to return the optimal solution. Consequently, in the rest of this dissertation, we always assume the heuristic function h is admissible.

A* is obtained from Algorithm 3.1 by (i) expanding a node with minimal f-value at each iteration, and (ii) relaxing arcs using Algorithm 3.3. In the special case where the heuristic function h is consistent, A* can be seen as a generalization of Dijkstra's algorithm which iteratively expands an *open* node with minimal f-value instead of the one with minimal g-value.

However, when guiding search with an admissible but inconsistent heuristic, A* may have to *re-expand* nodes: indeed, it does not guarantee that the g-value of *closed* nodes is optimal, *i.e.,* it may expand *open* nodes with suboptimal g-values. Finding an improved path to a *closed* node s requires expanding it again to propagate this improvement in g-value to all of its successors. Therefore, when finding an improved path to an already expanded node s', s' is removed from the *closed* set and inserted back into *open* for later consideration (L4-6 of Algorithm 3.3). In Figure 3.4, we provide a sample graph and an admissible but inconsistent heuristic which leads A* to expand the same node twice.

Algorithm 3.3: Relaxation of arc (s, s') in A*

```
1 procedure A*-relax(s, s', w, h, open, closed, g, π):
      input        : arc (s, s') ∈ A, weights w : A → ℝ₀⁺ and an heuristic function h : N → ℝ₀⁺
      input/output : the open and closed sets, maps g : N → ℝ₀⁺ and π : N → N
2     if g(s') > g(s) + w(s, s') then
3         set g(s') to g(s) + w(s, s');
4         if h is inconsistent and s' ∈ closed then
5             remove s' from the closed set;
6         ensure that s' belongs to the open set;
7         set π(s') to s;
```

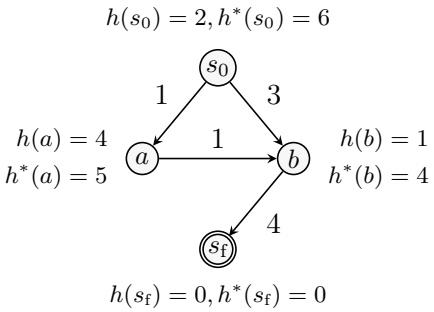


Figure 3.4: Example of an admissible but inconsistent heuristic. Heuristic h is admissible because $h(s) \leq h^*(s)$ for each state $s \in \{s_0, a, b, s_f\}$, but it is not consistent given $h(a) \leq w(a, b) + h(b)$ does not hold. In this case, A* expands node b twice.

Recall that Dijkstra's algorithm and BFS can compute a lower bound $\underline{g}$ based on the minimum g-value amongst *open* states. Similarly in A*, $\underline{f} = \min_{s \in open} \{f(s)\}$ provides, at the beginning of each iteration, a lower bound on $g^*(s_f)$, *i.e.,* the cost of the optimal path from s_0 to s_f.

In Figure 3.5, we schematically compare the set of states expanded by informed and uninformed search algorithms. Uninformed search has to consider all states with $g(s) < g^*(s_f)$, whereas informed search only considers the subset of these nodes for which $f(s) < g^*(s_f)$. While uninformed search explores the search space uniformly by expanding nodes in increasing order of g, informed search is biased towards nodes expected to be close to the goal state s_f. In Figure 3.5, state s_i is expanded by both types of algorithms, state s_j is not considered by informed search because $f(s_j) > g^*(s_t)$, and state s_k is considered by neither approaches given $g(s_k) > g^*(s_t)$.

Although A* tends to expand less nodes than Dijkstra's algorithm because it guides the search using external information, it has the same space and time complexities as it degenerates into Dijkstra's algorithm when using the null heuristic h^0. Additionally, A* requires $O(|\mathcal{N}|)$ calls to the heuristic function. Note that using h_1, a more informed heuristic than h_2 (*i.e.,* $h_1(s) \geq h_2(s), \forall s \in \mathcal{N}$, and $\exists s \in \mathcal{N}$ such that $h_1(s) > h_2(s)$), does not necessarily guarantee faster solving times, since there is a trade-off between the time required to compute the heuristic function and the number of node expansions it avoids. It has also been proven that, given a consistent heuristic h, no exact algorithm – starting search from s_0 and using the same heuristic information – can expand fewer nodes than A* to solve the problem [DP85].

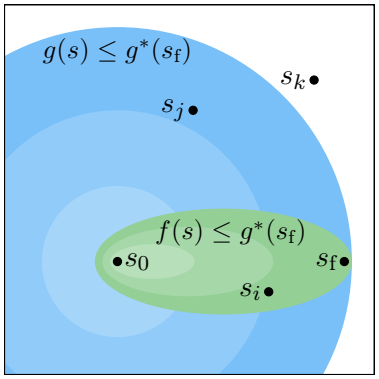


Figure 3.5: Schematic comparison of Dijkstra's algorithm (in blue) and A* using an admissible heuristic (in green) when looking for a shortest path from s_0 to s_f. Light shades of blue (resp. green) denote states with low g-values (resp f-values), and darker shades denote higher values.

Many strategies have been developped to improve the scalability of A*, such as:

- Bidirectional search [Poh69] and, more specifically, Bidirectional A* [Poh71; KK97] consists in starting a backward search from the final node s_f in addition to the regular forward search from the initial node s_0; a solution is found when both search frontiers intersect. Note that this strategy can only be applied when the search operators are reversible, *i.e.,* when it is possible to compute inverse costs and transitions.

- Iterative Deepening A* (IDA*) [Kor85] iteratively performs partial depth-first searches. These searches are limited by a f-value threshold which is increased after each unsuccessful iteration, *i.e.,* IDA* trades node re-expansions for memory.

- Recursive Best-First Search (RBFS) [Kor93] is space optimized variant of A* which only uses $O(bd)$ space (*i.e.,* linear space with respect to the depth of the graph). This is achieved by backtracking and re-expanding nodes instead of explicitly maintaining *open* and *closed* sets. RBFS is therefore not able to perform duplicate detection: as we have seen in Subsection 3.2.1 for DFS, this may lead to exponential increases in time complexity in the general case.

- Enhanced A* (EA*) [II99], consists in computing an upper bound on $g^*(s_f)$ before starting the search and discarding states which are known to be unable to improve it, thereby reducing the size of the *open* set (this technique is called *bounding* and will be covered more in depth when considering exact and anytime search algorithms in Subsection 3.3.3).

- Partial Expansion A* (PEA*) [YMI00] also seeks to limit the number of *open* nodes, but achieves it at the price of node re-expansions. When expanding a node s, PEA* only adds its most promising successors to *open*. Completeness is ensured by adding s back to *open* to be fully expanded later, and its cutoff f-value is stored to determine which of its successors have not yet been considered.

Although these approaches do improve on drawbacks of A*, they still require exponential space or time to provide a solution. Consequently, we do not describe them in depth but instead consider exact and anytime search algorithms, which are less likely to fail to provide a solution in large

enough search spaces because of space or time constraints.

3.3.3 Exact and Anytime Search (EAS) algorithms related to A*

We have seen that the space and time complexities of A* depend on the size of the state transition graph on which it operates. However, in some problems such as the TSP or the blocks world, the size of this graph grows exponentially with respect to the instance size. In this context, A* requires exponential time and memory to find the optimal solution, *i.e.,* it may have to expand and store an exponential number of nodes. In other words, A* does not scale well, and it may – on large enough problem instances – either (i) not terminate within a reasonable time, or (ii) run out of memory before having found any solution.

Anytime algorithms seek to decrease the probability of such a failure by trying to output a sequence of solutions of increasing quality instead of – like A* – a single but optimal solution. Of course, they still may fail to find a solution under the time and memory limits. Anytime solvers are well suited to real-world planning problems in which computation time is limited and potentially uncertain [LGT03; HZ07].

We do not consider non-exact anytime algorithms, *i.e.,* algorithms which do not provide optimality guarantees. We instead focus on *exact* (or *complete*) anytime algorithms, *i.e.,* algorithms which are also able to prove (i) that the last solution found is optimal, or (ii) that no solution exists. Unlike A*, which both finds the optimal solution and proves its optimality at the same time, Exact and Anytime Search (EAS) algorithms first find a sequence of solutions of increasing quality before proving the optimality of the last solution found.

We do not consider *online* planning algorithms either as they are more suited to unknown or stochastic environments in which individual decisions must be taken quickly and have uncertain outcomes [HZ07; RN10]. Unlike *offline* planning algorithms, which compute a complete solution before executing any "real-world" actions, online planning algorithms interleave planning and action execution, and therefore typically expand only one path at a time. A notable example of online planning algorithm is Learning Real-Time A* (LRTA*) [Kor90]. It assumes the goal node s_{f} can be reached from every node of the graph, and iteratively refines heuristic values h from experience by performing a series of *trials*.

General principles

EAS algorithms related to A* have recently received a lot of attention [TR10; Lib20; KB23c]. In particular, [LF21] won the 2018 ROADEF/EURO challenge using such an algorithm. Like A*, these EAS algorithms (i) use an an admissible (but not necessarily consistent) heuristic function $h(s)$, (ii) guide the search using an evaluation function in the form of $f(s) = g(s) + h(s)$, and (iii) usually maintain *open* and *closed* sets, progressively refining the best-known g-value for each state they contain.

EAS algorithms typically expand states in a different order than A*. Indeed, approximate solutions can only be found if one allows expanding states with possibly suboptimal g-values. Consequently, some nodes may have to be expanded multiple times (similarly to A*, when guided by an admissible but inconsistent heuristic).

Generally, EAS algorithms rely on *pruning* in order to save time and memory by discarding nodes. We distinguish two main kinds of pruning strategies, *inadmissible pruning* and *admissible pruning*.

Inadmissible pruning (or *heuristic pruning*) consists in discarding states which may belong to the optimal path from s_0 to s_f in order to reduce the size of the search space and to potentially find approximate solutions quickly. EAS algorithms typically recover from this by repeatedly restarting the search using weaker pruning rules until the entire search space has been considered and optimality is proven. Adopting this strategy requires re-expanding nodes, which may lead to time overheads when it is expensive to compute the successors of a node or the heuristic h.

On the other hand, *admissible pruning* consists in discarding states which are guaranteed not to belong to the optimal path from s_0 to s_f. This type of pruning is said to be admissible because the algorithm remains exact. *Bounding* is a type of admissible pruning, which consists in discarding nodes that cannot improve the current best-known solution of cost $g(s_\mathrm{f})$, *i.e.,* nodes s for which $f(s) = g(s) + h(s) \geq g(s_\mathrm{f})$. Note that A* also avoids expanding a subset of these nodes (*i.e.,* the ones with $f(s) > g^*(s_\mathrm{f})$, as it expands states in increasing order of f-values), but it has to store all of them in *open* given no upper bound on $g^*(s_0)$ is known before finding the optimal solution (*i.e.,* $g(s_\mathrm{f}) = \infty$ until the final state s_f is expanded).

Most of these algorithms are able to compute some suboptimality bounds (*i.e.,* upper bounds on the relative error $\frac{g(s_\mathrm{f})}{g^*(s_\mathrm{f})}$), based on (i) the cost of the current best-known solution and (ii) the minimum f-value amongst the states which belong to the fringe.

EAS algorithms often have parameters which allow one to find a balance between (i) the time required to find a first solution and its quality, (ii) the frequency at which solutions can be found, (iii) the overall convergence and solving speeds. More generally, they allow one to balance *exploration* (or *diversification*) and *exploitation* (or *intensification*), *i.e.,* best-first and depth-first behaviors, respectively. As in A*, the heuristic function plays an important role on the overall speed and memory use.

Taxonomy

Table 3.1 presents a high-level comparison of EAS algorithms related to A*: they are listed in chronological order and compared in terms of (i) their expansion strategy, and (ii) the space and time trade-off they propose.

Recall that A* is purely best-first and is therefore not anytime. EAS algorithms use an alternative expansion strategy to introduce a depth-first behavior while trying to strike a balance between exploration and exploitation. We distinguish three kinds of expansion strategies, *i.e.,* the algorithms

Name	Ref.	Expansion strategy: Weighted heuristic	Expansion strategy: Breadth constraints	Expansion strategy: Depth constraints	Space-time trade-off: Duplicate detection	Space-time trade-off: Memory bounded
AWA*	[ZH02]	✓			✓	
ARA*	[LGT03]	✓			✓	
BSS	[ZH05]		✓			✓[a]
AWinA*	[ACK07]			✓	✓	
AWRBFS	[HZ07]	✓				✓
CBFS	[KSJ09]		✓		✓	
RWA*	[RTR10]	✓			✓[b]	
MAWinA*	[VAC11]			✓		✓[a]
ANA*	[Van+11]	✓			✓	
ACS	[Vad+12]		✓		✓	
APS	[VAC16]		✓		✓	
IBS	[Lib+20]		✓		✓[b]	
RandWA*	[BSZ21]	✓			✓	

Table 3.1: High-level comparison of EAS algorithms related to A*, in terms of expansion strategies as well as space and time trade-offs. Caveats: [a]Performs partial duplicate detection, [b]Re-expands nodes due to restarts.

considered either:

- rank the *open* states using a weighted evaluation function, *i.e.,* prioritize nodes with low h-values,
- use breadth constraints, *i.e.,* limit the number of nodes expanded in each layer of the search graph, or
- use depth constraints, *i.e.,* restrict node expansions based on their depth.

We also differentiate these algorithms according to the space and time trade-off they provide. On one side of this spectrum, algorithms have a bounded space complexity; on the other side of this spectrum, algorithms use memory to perform duplicate detection and thus avoid wasting time performing redundant node expansions. Detecting that multiple paths lead to the same state requires memorizing *open* and *closed* states as well as their g-values; it is therefore orthogonal to having a bounded space complexity. Note however that (i) some memory-bounded algorithms are able to perform partial duplicate detection, and (ii) some algorithms perform duplicate detection but re-expand nodes because of restarts.

Other desirable properties include (i) having a small number of parameters to tune, (ii) the ability to quickly provide a first solution, and (iii) the frequency at which solutions may be reported. We do not discuss these properties in details in this chapter as they are not trivial to evaluate in the general case (*e.g.,* they may depend on the topology of the state transition graph).

We now briefly introduce each of the three expansion strategies used by these EAS algorithms to make A* more depth-first.

Weighted heuristics

Weighted A* (WA*) [Poh70] is an incomplete extension of A* which prioritizes the *open* nodes using a weighted evaluation function $f_w(s)$. Given an admissible heuristic function $h(s)$ and a weight $w > 1$, the weighted evaluation function is defined as $f_w(s) = g(s) + w \cdot h(s)$. This weighted evaluation function artificially increases the heuristic estimate and therefore makes the nodes close to the goal look more appealing, *i.e.,* it makes A* more "depth-first". The weighted heuristic $w \cdot h$ is said to be w-admissible and guarantees that the solution found is at most w times worse than the optimal solution, *i.e.,* $g^*(s_f) \leq g(s_f) \leq w \cdot g^*(s_f)$. Because this heuristic is not necessarily consistent, WA* may have to re-expand nodes. WA* is purely greedy when w tends to ∞ and degenerates into A* when $w = 1$.

Anytime Weighted A* (AWA*) [ZH02; HZ07] is an exact and anytime extension of WA* which (i) continues the search after finding a solution, and (ii) proves optimality when the *open* set becomes empty. AWA* performs bounding and computes suboptimality bounds using the unweighted evaluation function f. Anytime Weighted RBFS (AWRBFS) [HZ07] adopts a similar strategy but only requires $O(bd)$ space through backtracking and node re-expansions.

Anytime Repairing A* (ARA*) [LGT03; Lik+08] and Restarting Weighted A* (RWA*) [RTR10] progressively decrease the weight w to converge faster and to obtain tighter suboptimality bounds. RWA* restarts the search after having found a solution in order to bring diversity, and memorizes g-values and h-values in order to reuse previous search effort.

More recently, approaches were proposed to avoid parameter tuning, *i.e.,* to avoid adjusting the weight w (and its decrease schedule, in the case of ARA* and RWA*). These include Anytime Nonparametric A* (ANA*) [Van+11], which adjusts the weight using the cost of the current best-known solution, and Randomized Weighted A* (RandWA*) [BSZ21], which operates with a set of weights and randomly chooses one before each expansion.

Breadth-constrained

Beam Search [Low76] is an incomplete search algorithm which only expands the B most promising nodes at each level of the state transition graph; the remaining nodes, if any, are inadmissibly pruned. This algorithm is very similar to the concept of Restricted Dynamic Programming discussed in Chapter 2. Many EAS algorithms related to A* use such breadth-constraints to obtain a more depth-first behavior.

For example, Cyclic Best First Search[1] (CBFS) [KSJ09] iteratively expands the single most promising node at each level of the state transition graph, and proves optimality when no state remains *open*. Unlike Beam Search, it does not inadmissibly prune the least promising nodes but stores them in an *open* set instead. Anytime Column Search (ACS) [Vad+12] generalizes CBFS by iteratively expanding the B most promising states at each level of the state transition graph.

[1] CBFS was originally named Distributed Best First Search (DBFS).

Iterative Beam Search (IBS) [Lib+20] repeatedly performs beam searches from the initial state s_0 and geometrically increases B between iterations to achieve completeness. Beam-Stack Search (BSS) [ZH05], unlike IBS, achieves completeness by backtracking instead of restarting search, and uses only $O(Bd)$ space. Anytime Pack Search (APS) [VAC16] consists in running a series of beam searches which start from the B most promising *open* nodes instead of the initial state s_0.

Depth-constrained

Anytime Window A* (AWinA*) [ACK07] is an anytime extension of A* which restricts node expansions based on node depths. It relies on an incomplete search procedure called Window A* (WinA*), which takes as parameter a depth tolerance *tol*. This procedure, like A*, iteratively expands a most promising *open* state. It however keeps track of the level $\overline{k}$ of the deepest state expanded yet (initially, $\overline{k} = -\infty$), and only considers *open* nodes located deeper than level $\overline{k}$, with a tolerance of *tol* levels. In other words, WinA* expands, at each iteration, a most promising *open* state such that its depth is greater than $\overline{k} - tol$.

AWinA* repeatedly calls this procedure with an increasing tolerance *tol* (*i.e.,* depth constraints are progressively weakened), and proves optimality when the *open* set becomes empty. The first WinA* call operates with no tolerance (*i.e.,* $tol = 0$) and the search is purely depth-first (as expanding a node at level k restricts later expansions to levels deeper than k). Tolerance *tol* is incremented after each call to the WinA* procedure to gradually make it more best-first. WinA* degenerates into A* when *tol* becomes equal to the depth of the goal node.

Memory bounded Anytime Window A* (MAWinA*) [VAC11] is a memory-bounded extension of AWinA*. It generates successor nodes one at a time, and enforces a limit on the size of the *open* and *closed* sets. When needed, MAWinA* "forgets" one of the least promising nodes and, if necessary, marks its parent as *open* to ensure that the forgotten node can be re-generated later.

3.4 Discussion

In this chapter, we have introduced the fundamentals of planning through state space search. We have seen that classic algorithms such as Dijkstra's algorithm or A* do not scale well enough for solving problems in which the size of the state transition graph grows exponentially with respect to the size of the input instance. We therefore presented and categorized various EAS algorithms related to A*: these algorithms are also guided by a heuristic function, but – unlike A* – they seek to provide the best possible solution given a limited amount of space and time.

In the next chapter, we tackle the TD-TSPTW$_m$ by defining a state transition graph based on its DP formulation and proposing three heuristic functions. We also describe ACS, AWA*, AWinA* and IBS more in depth by instantiating these algorithms to solve the TD-TSPTW$_m$. Additionally, we highlight similarities and differences between these EAS algorithms before comparing them experimentally in Chapter 6.

Part

II

Proposed solving approach

Chapter

4 Exact and Anytime Search (EAS) for the TD-TSPTW$_m$

Contents

We have seen in Chapter 2 that DP can be used to solve the TSP and some of its variants. Also recall that such a problem can be solved by computing a shortest path in a state transition graph, *e.g.,* using Exact and Anytime Search (EAS) algorithms (see Chapter 3). In this chapter, we propose to use EAS algorithms to solve the TD-TSPTW$_m$ by looking for shortest paths in the state transition graph associated to its DP formulation.

We first provide in Section 4.1 a DP formulation for the TD-TSPTW$_m$, define the associated state transition graph and provide a simple solving algorithm which is exact but not anytime. Then, in

Section 4.2, we instantiate four EAS algorithms to the TD-TSPTW$_m$, study their similarities and differences, and propose a new EAS algorithm by hybridizing two of them. In Section 4.3, we discuss major implementation decisions and propose novel and efficient ways to implement some of these algorithms. Finally, in Section 4.4, we introduce three lower bounding functions which are used by EAS algorithms to guide search and to prune the state space.

In the next chapter, we describe how we combine EAS algorithms with (i) TW constraint propagation, in order to reduce the size of the search space, and (ii) a local search procedure which tries to improve the solutions found.

4.1 Dynamic Programming model and state transition graph

In this section, we present a DP model for the TD-TSPTW$_m$, define the state transition graph associated to it and show how it may be used to solve the problem using a simple shortest path algorithm which is exact but not anytime.

DP formulation. Recall from Chapter 1 that a TD-TSPTW$_m$ instance is defined as a tuple $(\mathcal{V}, c, e, l)$ in which $\mathcal{V}$ is the set of vertices and $c : \mathcal{V} \times \mathcal{V} \times \mathbb{N} \to \mathbb{N}$ a matrix of TD travel time functions verifying the FIFO property. The visit time at vertex $i \in \mathcal{V}$ is constrained to time interval $[e_i, l_i]$. Special vertices $0 \in \mathcal{V}$ and $n \in \mathcal{V}$ denote the origin and destination vertices: solving the TD-TSPTW$_m$ requires finding a path that departs from 0 at time e_0 and ends on vertex n as early as possible while visiting each vertex of set $\mathcal{V}$ during its TW.

For convenience, we note $a : \mathcal{V} \times \mathcal{V} \times \mathbb{N} \to \mathbb{N}$ the arrival time function (*i.e.*, $a_{i,j}(t) = t + c_{i,j}(t)$) and $t_{\uparrow[e_i;l_i]}$ the TW-aware visiting time at vertex i, which includes a waiting time until e_i in case of early arrival (*i.e.*, $t < e_i$), returns infinity when arriving too late (*i.e.*, $t > l_i$), and returns t when arriving during the TW (*i.e.*, $e_i \leq t \leq l_i$).

Also recall that we note $G_{\text{PR}} = (\mathcal{V}, \mathcal{R})$ the precedence graph and $G_{\text{UA}} = (\mathcal{V}, \mathcal{E})$ the usable arcs graph associated to a TD-TSPTW$_m$ instance (see the TW constraint propagation rules defined in Section 1.3).

The Bellman equations for the TD-TSPTW$_m$ may be formulated as follows:

$$p(i, \mathcal{S}) = \begin{cases} a_{0,i}(e_0)_{\uparrow[e_i,l_i]} & \text{if } \mathcal{S} = \{0, i\} \\ \min_{j \in \mathcal{S} \setminus \{0,i\}} a_{j,i}(p(j, \mathcal{S} \setminus \{i\}))_{\uparrow[e_i,l_i]} & \text{otherwise.} \end{cases} \tag{4.1}$$

Value $p(i, \mathcal{S})$ denotes earliest visit time at vertex i when departing from the origin vertex 0 at time e_0 and visiting each vertex of $\mathcal{S}$ during its TW; when TW constraints cannot be satisfied, then $p(i, \mathcal{S}) = \infty$.

These equations do not take into account precedence constraints $\mathcal{R}$ and usable arcs $\mathcal{E}$. In the next paragraph, we define the state transition graph associated to these equations while taking $\mathcal{R}$ and

$\mathcal{E}$ into account.

State transition graph. Recall that in Chapter 2, we defined nodes of the TSP's DP state transition graph as couples $(i, \mathcal{S})$: arcs represented weighted transitions between nodes, *i.e.,* transition between nodes $(i, \mathcal{S})$ and $(i', \mathcal{S}')$ had weight $c_{i,i'}$. However, when considering TD travel times, it becomes necessary to know the departure time from vertex i in order to determine the travel time from i to i'. The departure time from vertex i is also required to handle TW constraints given it determines the arrival time on i': depending on this arrival time, (i) a waiting time may increase the transition cost or (ii) the transition may be infeasible. In other words, in the induction case of Equation 4.1, the transition cost from state $(j, \mathcal{S} \setminus \{i\})$ to state $(i, \mathcal{S})$ is a function of $p(j, \mathcal{S} \setminus \{i\})$, whereas it is the constant $c_{j,i}$ when considering the TSP.

Consequently, we define states as triples $(i, \mathcal{S}, t)$ such that $\{0, i\} \subseteq \mathcal{S} \subseteq \mathcal{V}$ and $t \in [e_i, l_i]$. A given state $(i, \mathcal{S}, t)$ belongs to the state transition graph if there exists a path that visits vertex i at time t when departing from vertex 0 at time e_0 and visiting each vertex of $\mathcal{S}$ during its TW. We now define the state transition graph more formally as a planning problem based on this state definition.

In the initial state, only the origin vertex 0 has been visited and it must be departed from at time e_0, *i.e.,* $s_0 = (0, \{0\}, e_0)$. Contrarily to the TSP, the TD-TSPTW$_m$ may have multiple final states, *i.e.,* a final state $(n, \mathcal{V}, t)$ exists for each time $t \in [e_n, l_n]$ such that there exists a path departing from vertex 0 at time e_0 and arriving on vertex n at time t while visiting each vertex of $\mathcal{V}$ during its TW. We therefore determine whether or not a state is final through predicate $final(i, \mathcal{S}, t) \iff i = n \wedge \mathcal{S} = \mathcal{V}$.

Given a state $s = (i, \mathcal{S}, t)$, an action consists in visiting an additional vertex $j \in \mathcal{V} \setminus \mathcal{S}$: the set of actions is therefore defined as $\mathcal{A}_P = \mathcal{V} \setminus \{0\}$. An action $j \in \mathcal{A}_P$ is feasible in state s if (i) vertex j has not yet been visited, (ii) arc (i, j) may be used in a feasible solution, (iii) all vertices that must precede j have already been visited and (iv) the TW constraint at j is satisfied, *i.e.,*

$$F(i, \mathcal{S}, t) = \{j \in \mathcal{V} \setminus \mathcal{S} : (i, j) \in \mathcal{E} \wedge pred(j) \subseteq \mathcal{S} \wedge a_{i,j}(t) \leq l_j\} \tag{4.2}$$

where $pred(j)$ denotes the set of vertices that must be visited before vertex j, *i.e.,* $pred(j) = \{k \in \mathcal{V} : (k, j) \in \mathcal{R}\}$. Note that precedence constraints implicitly prevent one from reaching the destination vertex n before having visited all other vertices.

Given a state $s = (i, \mathcal{S}, t)$ and a feasible action $j \in F(i, \mathcal{S}, t)$, transition function τ determines the state $s' = (j, \mathcal{S} \cup \{j\}, t')$ obtained when applying action j to state s. Value t' denotes the visit time at vertex j: it depends on both (i) the arrival time on j when departing from i at time t and (ii) the potential waiting time for the opening of j, *i.e.,*

$$\tau((i, \mathcal{S}, t), j) = \Big(j, \mathcal{S} \cup \{j\}, \max\left(e_j, a_{i,j}(t)\right)\Big) \tag{4.3}$$

Because we consider the makespan objective, the cost of this transition includes both the travel

time and the potential waiting time, *i.e.*,

$$c_P((i, \mathcal{S}, t), j) = \max(e_j, a_{i,j}(t)) - t \tag{4.4}$$

We call this planning problem $P = (\mathcal{S}_P, \mathcal{A}_P, s_0, final, F, \tau, c_P)$. As we have seen in Chapter 3, this problem may be solved by looking for a shortest path from the initial state s_0 to a final state in the state transition graph $G^P_{\text{ST}} = (\mathcal{N}, \mathcal{A}, w)$ associated to P.

We illustrate in Figure 4.2 the state transition graph associated to the TSPTW$_m$ instance of Figure 4.1 (for simplicity and without loss of generality, we consider constant travel times). This graph encodes the three feasible solutions of the instance: a single path exists from s_0 to final state $(n, \mathcal{V}, 7)$, and two optimal paths lead to final state $(n, \mathcal{V}, 6)$.

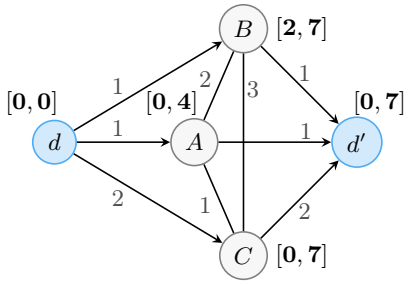


Figure 4.1: Sample TSPTW instance with three customer vertices $\mathcal{C} = \{A, B, C\}$. Vertices d and d' repectively represent the origin and the destination. This instance has three feasible solutions $\langle d, B, A, C, d' \rangle$, $\langle d, A, C, B, d' \rangle$ and $\langle d, C, A, B, d' \rangle$ with respective makespans 7, 6 and 6 time units.

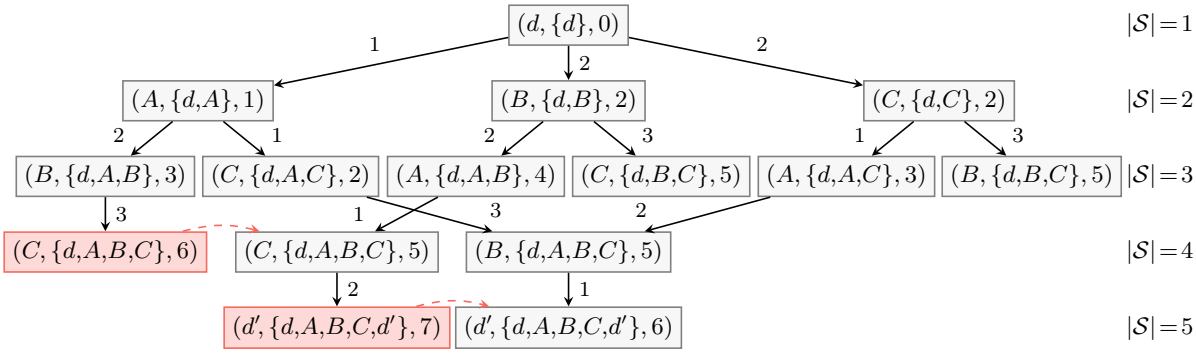


Figure 4.2: State transition graph associated to the TSPTW$_m$ instance of Figure 4.1. Dominated states are represented in red. A red arrow from state s to state s' indicates that s is dominated by s'.

Notice that in the graph of Figure 4.2, the cost of all paths from s_0 to a given state $(i, \mathcal{S}, t)$ is equal to t. More generally, all paths from s_0 to $(i, \mathcal{S}, t) \in \mathcal{N}$ have a cost of $t - e_0$: this comes from the relation between τ and c_P, *i.e.*, function τ could be alternatively defined as $\tau((i, \mathcal{S}, t), j) = (j, \mathcal{S} \cup \{j\}, t + c_P((i, \mathcal{S}, t), j))$.

This is specific to the makespan objective, in which (i) the departure time from the origin vertex is fixed and (ii) waiting times are included in the objective function. In other words, given a state $(i, \mathcal{S}, t)$, the visit time t at vertex i is the objective to minimize, and it is also used to compute transition costs and transition feasibility.

Dominance relation. Given two states $s, s' \in \mathcal{N}$ such that $s = (i, \mathcal{S}, t)$, $s' = (i, \mathcal{S}, t')$ and $t < t'$, it is not necessary to consider s' to compute an optimal solution, as formalized in Equation 4.1. Indeed, it is cheaper to reach s than s' from s_0, and reaching a final state from s (resp. s') requires finding a path departing from i at time t (resp. t') which arrives as early as possible on n while visiting each vertex of $\mathcal{V} \setminus \mathcal{S}$ during its TW: a later departure from i cannot allow one to visit n earlier (i) by the FIFO property and (ii) because less "slack time" is available to visit the remaining vertices within their TW. State s is said to *dominate* s', which we define as the predicate:

$$dominates((i, \mathcal{S}, t), (i', \mathcal{S}', t')) \iff i = i' \wedge \mathcal{S} = \mathcal{S}' \wedge t < t' \tag{4.5}$$

In Figure 4.2, state $s_1 = (C, \{d, A, B, C\}, 5)$ dominates $s'_1 = (C, \{d, A, B, C\}, 6)$. In both these states, only the destination vertex d' is left to be visited: while s_1 leads to a final state (*i.e.,* $d' \in F(s_1)$), d' cannot be reached on time in state s'_1 (*i.e.,* $d' \notin F(s'_1)$).

Note that, when seeking to obtain approximate solutions quickly (*e.g.,* in EAS algorithms), it is necessary to consider dominated states as long as no better alternative is known (*i.e.,* one may expand a dominated state s' provided that no state s such that $dominates(s, s')$ has yet been found). The reason for this is twofold:

- It takes exponential time to prove that a state $(i, \mathcal{S}, t)$ is not dominated (*i.e.,* it requires computing $p(i, \mathcal{S})$).
- In G^P_{ST}, the path associated to a suboptimal solution necessarily goes through dominated states (*e.g.,* the dominated state $(d', \{d, A, B, C, d'\}, 7)$ in Figure 4.2).

In the algorithms presented in this chapter, we handle dominance implicitly by (i) memorizing the best-known value associated to each subproblem $(i, \mathcal{S})$ in a map g and (ii) only expanding states $(i, \mathcal{S}, t)$ such that $t = g(i, \mathcal{S})$. Exact and non-anytime algorithms only expand states for which $g(i, \mathcal{S})$ is known to be optimal, *i.e.,* $g(i, \mathcal{S}) = p(i, \mathcal{S})$.

Basic solving algorithm. Algorithm 4.1 computes the optimal value associated to each subproblem $(i, \mathcal{S})$ by expanding all non-dominated states of G^P_{ST} in breadth-first order (see Subsection 3.2.2), starting from the initial state. More precisely, it considers subproblems $(i, \mathcal{S})$ by increasing order of size $|\mathcal{S}|$ while memorizing in a map g the best-known value associated to each feasible subproblem. Subproblems $(i, \mathcal{S})$ for which a solution is known are stored either in the *open* or in the *closed* set. A given subproblem belongs to *closed* if and only if its associated state has already been expanded.

Initially, only the optimal value of the trivial subproblem $(0, \{0\})$ is known. At Lines 3-4, this subproblem is added to the *open* set and its optimal value e_0 is stored in g. Each iteration of loop L5-11 uses the optimal value of feasible subproblems of size $k \in [1, |\mathcal{V}| - 1]$ to solve subproblems of size $k + 1$. More precisely, loop L6-11 considers all *open* subproblems of size k: each iteration removes from *open* and adds to *closed* a subproblem $(i, \mathcal{S})$ such that $|\mathcal{S}| = k$. Then, the associated state $s = (i, \mathcal{S}, t)$ such that $t = g(i, \mathcal{S})$ is expanded (L8-11): for each successor $s' = (i', \mathcal{S}', t')$ of

Algorithm 4.1: Layer-wise computation of the optimal value associated to each subproblem $(i, \mathcal{S})$

input : Transitions $\mathcal{A}$ of G^P_{ST}, vertex set $\mathcal{V}$, the departure time e_0 from the origin vertex $0 \in \mathcal{V}$

```
1  let g be an empty map associating to each subproblem (i, S) its best-known value;
2  let open and closed be empty sets of subproblems;
3  add subproblem (0, {0}) to open;
4  set g(0, {0}) to e_0;
5  foreach layer k ∈ [1, |V| − 1] do
6      foreach subproblem (i, S) ∈ open such that |S| = k do
7          remove (i, S) from open and add it to closed;
8          foreach state s' = (i', S', t') such that (s, s') ∈ A, with s = (i, S, g(i, S)) do
9              if (i', S') ∉ open or t' < g(i', S') then
10                 ensure that (i', S') belongs to open;
11                 set g(i', S') to t';
```

state s, we ensure that $(i', \mathcal{S}')$ belongs to *open* and update the best-known value associated to $(i', \mathcal{S}')$ by setting $g(i', \mathcal{S}')$ to t' if (i) no solution to this subproblem was previously known (*i.e.,* $(i', \mathcal{S}') \notin$ *open*) or (ii) a new best-known solution has been found (*i.e.,* $t' < g(i', \mathcal{S}')$).

This algorithm exploits the layered structure of the graph by expanding states in a topological order of G^P_{ST}, *i.e.,* it takes advantage of the fact that subproblems of a given size $k \in [2, \mathcal{V}]$ only directly depend on subproblems of size $k - 1$. Consequently, value $g(i, \mathcal{S})$ is guaranteed to be optimal (*i.e.,* $g(i, \mathcal{S}) = p(i, \mathcal{S})$) for all *closed* subproblems and, more generally, for all subproblems of size $[1, k + 1]$ at the end of each iteration of loop L5-11 (if such a subproblem $(i, \mathcal{S})$ does not belong to *open* $\cup$ *closed*, then it has no feasible solution, *i.e.,* $p(i, \mathcal{S}) = \infty$).

If a feasible solution exists, this algorithm provides us with the optimal makespan $g(n, \mathcal{V})$: an actual customer ordering of optimal makespan can be reconstructed in $O\left(|\mathcal{V}|^2\right)$ time from the optimal values stored in map g using the method presented in Section 2.1.

Implementation. Throughout this chapter, we discuss of major implementation decisions given these choices have an impact on the overall space and time requirements. In particular, each algorithm we consider needs to perform a specific set of operations efficiently on map g and on the *open* and *closed* sets. We first describe the data structures used to implement Algorithm 4.1, given it shares similarities with the EAS algorithms we shall consider in Section 4.2, and also propose an efficient way to compute transitions $\mathcal{A}$.

We store feasible subproblems of each layer $k \in [1, |\mathcal{V}|]$ in a hash map $m[k]$. Map $m[k]$ associates to each subproblem $(i, \mathcal{S}) \in$ *open* $\cup$ *closed* of size k its best-known value. This allows us to:

- iterate in linear time over all subproblems stored in $m[k]$,
- determine in average constant time whether or not a given subproblem $(i, \mathcal{S})$ belongs to *open* $\cup$ *closed* and to read and update its associated best-known value, and
- insert a new subproblem and its value in amortized constant time.

Also, we use bitsets to compactly represent set $\mathcal{S}$ in a subproblem $(i, \mathcal{S})$. Apart from being

space-efficient, bitsets also allow one to efficiently perform set operations. For instance, assuming integers encoded on 64 bits and provided $\mathcal{V}$ contains at most 64 elements, the set of feasible actions $F(i, \mathcal{S}, t)$ may be computed in linear time (F is defined in Equation 4.2 and implicitly used at Line 8 of Algorithm 4.1 to obtain successor states).

More precisely, set $\mathcal{V}$ is represented as a bitset and the adjacency matrix $\mathcal{E}$ as an array of $|\mathcal{V}|$ bitsets (we note $\mathcal{E}[i]$ the i^{th} bitset of this array, such that $\mathcal{E}[i] = \{j \in \mathcal{V} : (i, j) \in \mathcal{E}\}$). We compute in constant time a superset $F^+(i, \mathcal{S}, t)$ of the feasible actions in a state $(i, \mathcal{S}, t)$, *i.e.,* $F^+(i, \mathcal{S}, t) = \{j \in \mathcal{V} \setminus \mathcal{S} : (i, j) \in \mathcal{E}\} = (\mathcal{V} \setminus \mathcal{S}) \cap \mathcal{E}[i]$. We also encode $pred(j) = \{i \in \mathcal{V} : (i, j) \in \mathcal{R}\}$ as a bitset for each vertex $j \in \mathcal{V}$, and obtain $F(i, \mathcal{S}, t)$ in linear time with respect to $|F^+(i, \mathcal{S}, t)|$ by verifying that $pred(j) \subseteq \mathcal{S}$ and that the TW at j is respected. More generally, when $\mathcal{V}$ contains more than 64 elements, bitsets are represented by arrays of $\left\lceil \frac{|\mathcal{V}|}{64} \right\rceil$ integers, and each set operation is performed in $O(|\mathcal{V}|)$ time.

Discussion. Algorithm 4.1 is exact but has exponential space and time complexities. Consequently, it may fail to provide a solution when considering large enough graphs. In the next section, we instantiate EAS algorithms to solve the TD-TSPTW$_m$. Contrarily to this algorithm, they may expand dominated states (*i.e.,* states $(i, \mathcal{S}, t)$ for which $t = g(i, \mathcal{S}) > p(i, \mathcal{S})$), which enables them to provide approximate solutions. The cost of these approximate solutions are upper bounds on the optimal solution cost, which are used to perform bounding, *i.e.,* prune the search space using admissible heuristics.

Bounding may be incorporated to Algorithm 4.1 by noticing that l_n is a trivial upper bound on the optimal makespan, if a feasible solution exists. Provided an admissible heuristic h associating to each state $s = (i, \mathcal{S}, t)$ a lower bound on the cost of the shortest path from s to a final state in G^P_{ST}, one may tighten the condition of L9 by also verifying that state $s' = (i', \mathcal{S}', t')$ may lead to a feasible solution, *i.e.,* that $t' + h(i', \mathcal{S}', t') \leq l_n$.

Note that the set of feasible actions F defined in Equation 4.2 could be tightened by noticing that a state $(i, \mathcal{S}, t)$ is guaranteed not to lead to a feasible solution if there exists an unvisited vertex $j \in \mathcal{V} \setminus \mathcal{S}$ such that $a_{i,j}(t) > l_j$, provided triangle inequality holds (see, *e.g.,* [Dum+95]). We handle this case when computing lower bounds $h(i, \mathcal{S}, t)$ (covered in Section 4.4) by returning ∞ whenever a state is proven not to lead to a feasible solution.

4.2 Instantiation of EAS algorithms

In this section, we consider four of the A*-related EAS algorithms presented in Chapter 3 and instantiate them to solve the TD-TSPTW$_m$. We consider algorithms based on each expansion strategy from the taxonomy we provided in Subsection 3.3.3: Anytime Weighted A* (AWA*) based on weighted heuristics, Anytime Column Search (ACS) based on breadth constraints, and Anytime Window A* (AWinA*) based on depth constraints. We also consider Iterative Beam Search (IBS), a second algorithm based on breadth constraints: we study how it relates to ACS

and propose an improved version of IBS which re-expands fewer nodes.

We do not consider memory-bounded algorithms as detecting duplicate states is crucial in graphs with many redundant paths: this is the case when considering the DP state transition graph of the TD-TSPTW$_m$, given each subproblem $(i, \mathcal{S})$ has $O(|\mathcal{S}|!)$ solutions.

In Chapter 6, we experimentally compare these EAS algorithms and tune their parameters. We also study the impact of different implementation decisions, which highlights the computational cost of node re-expansions (*e.g.,* when restarting search after having inadmissibly pruned nodes).

Before describing the four EAS algorithms we consider, we first remind the reader of the main principles on which they rely by presenting them in the particular context of the TD-TSPTW$_m$.

Specifics of the TD-TSPTW$_m$. EAS algorithms require an admissible heuristic function h. Given a state s, $h(s)$ is a lower bound on $h^*(s)$, the cost of an optimal path from s to a final state in G^P_{ST}. Of course, state s may not lead to a final state due to TW constraints, *i.e.,* $h^*(s) = \infty$: admissible heuristics may detect this, and in turn return ∞.

The evaluation function f associated to a subproblem $(i, \mathcal{S})$ is obtained by summing the best-known value associated to $(i, \mathcal{S})$ to the heuristic value of state $s = (i, \mathcal{S}, t)$ such that $t = g(i, \mathcal{S})$. More formally, f is defined as $f(i, \mathcal{S}) = g(i, \mathcal{S}) + h(s)$: $f(i, \mathcal{S})$ is a lower bound on the cost of an optimal path from s_0 to a final state which goes through the current best-known state associated to subproblem $(i, \mathcal{S})$. We propose and describe three consistent heuristics h for the TD-TSPTW$_m$ in Section 4.4.

The evaluation function is used (i) to determine which state should be expanded next and (ii) to bound states, provided an upper bound ub on the optimal makespan: initially, $ub = l_n + 1$ and subproblems $(i, \mathcal{S})$ for which $f(i, \mathcal{S}) \geq ub$ may be pruned. Each time a solution of makespan $m < ub$ is found, ub is set to m in order to prevent considering paths leading to solutions worse than or equivalent to the current best-known solution.

Similarly to Algorithm 4.1, the majority of these algorithms operate on *open* and *closed* sets. The *closed* set contains subproblems $(i, \mathcal{S})$ such that state $(i, \mathcal{S}, g(i, \mathcal{S}))$ has already been expanded, although $g(i, \mathcal{S})$ is no longer guaranteed to be optimal. The *open* set contains subproblems $(i, \mathcal{S})$ such that state $(i, \mathcal{S}, g(i, \mathcal{S}))$ (i) needs to be expanded, and (ii) may lead to a solution of makespan lower than ub, *i.e.,* $f(i, \mathcal{S}) < ub$.

We start by instantiating Anytime Weighted A* to solve the TD-TSPTW$_m$ because this algorithm is very similar to A* and thus relatively simple.

4.2.1 Anytime Weighted A*

Anytime Weighted A* (AWA*) [ZH02; HZ07] is a generalization of A*: provided a heuristic function h and a heuristic weight $w \in \mathbb{R}$ such that $w \geq 1$, AWA* determines the most promising *open* node to expand using a weighted evaluation function f_w, such that $f_w(s) = g(s) + w \cdot h(s)$.

When $w > 1$, the heuristic estimate is artificially increased to make the nodes with low h-values more appealing, which makes search more "depth-first". Of course, the weighted heuristic $w \cdot h$ is not necessarily admissible or consistent: therefore, AWA* may expand states with suboptimal g-values, which (i) allows it to provide approximate solutions, but (ii) leads to node re-expansions. Therefore, search continues until the *open* set becomes empty instead of – like A* – stopping once a solution is found. Bounding is performed using the admissible heuristic, *i.e.,* using the unweighted evaluation function f.

Algorithm 4.2: AWA* for the TD-TSPTW$_m$

input : Transitions $\mathcal{A}$ of G^P_{ST}, vertex set $\mathcal{V} = \{0, \ldots, n\}$,
the departure time e_0 from the origin vertex 0, the deadline l_n of the destination vertex n,
an admissible heuristic h and a heuristic weight w

```
1  let g, f and f_w be empty maps associating a value to each subproblem (i, S);
2  let open and closed be empty sets of subproblems;
3  ub ← l_n + 1;
4  set g(0, {0}) to e_0;
5  set f(0, {0}) to e_0 + h(0, {0}, e_0);
6  set f_w(0, {0}) to e_0 + w · h(0, {0}, e_0);
7  add (0, {0}) to open;
8  while open ≠ ∅ do
9      remove a subproblem (i, S) from open such that f_w(i, S) is minimal;
10     add (i, S) to closed;
11     foreach state s' = (i', S', t') such that (s, s') ∈ A, with s = (i, S, g(i, S)) do
12         if [(i', S') ∈ open ∪ closed and t' ≥ g(i', S')] or t' + h(s') ≥ ub then continue;
13         set g(i', S') to t';
14         if final(s') then
15             ub ← t'; // New solution found
16             remove from open each subproblem (i'', S'') such that f(i'', S'') ≥ ub;
17             ensure that (i', S') belongs to closed;
18         else
19             if (i', S') ∉ open then add (i', S') to open;
20             if (i', S') ∈ closed then remove (i', S') from closed;
21             set f(i', S') to t' + h(s') and f_w(i', S') to t' + w · h(s');
```

Instantiation to the TD-TSPTW$_m$. In our application, the weighted guide function f_w is defined as $f_w(i, \mathcal{S}) = g(i, \mathcal{S}) + w \cdot h(i, \mathcal{S}, g(i, \mathcal{S}))$.

We instantiate AWA* to the TD-TSPTW$_m$ in Algorithm 4.2: we highlight in blue the differences with A*. At Lines 4-7, maps g, f and f_w are initialized with the values associated to the initial state, which is added to *open*[1]. As long as the *open* set is not empty, each iteration of the main loop L8-21 considers a most promising *open* subproblem, *i.e.,* a subproblem with minimal f_w-value.

[1]For simplicity and without loss of generality, we assume in this section that the heuristic function h does not prove that no final state is reachable from the initial state $(0, \{0\}, e_0)$, *i.e.,* we assume that $e_0 + h(0, \{0\}, e_0) \leq l_n$.

The state s associated to this subproblem is expanded at L10-21: at L12, a successor s' of state s is discarded (i) if an equivalent or better solution to subproblem $(i', \mathcal{S}')$ is already known, or (ii) if s' is proven to lead to a solution of makespan greater than or equal to the upper bound ub. If s' is a final state, then a new best-known solution has been found: ub is updated to the new best-known makespan t', bounded subproblems (*i.e.,* subproblems $(i, \mathcal{S})$ for which $f(i, \mathcal{S}) \geq ub$) are removed from *open*, and s' is added to *closed* (L15-17). Otherwise, s' is added to *open* for later expansion: it is removed from *closed* if necessary (because the weighted heuristic is not necessarily consistent, a state dominated by s' may have already been expanded *i.e.,* a state $(i', \mathcal{S}', t'')$ such that $t'' > t'$), and its values in maps f and f_w are updated (L19-21).

Note that, contrarily to A* which checks whether or not a state is final when expanding it, AWA* tests this as soon as a state is generated, given it may allow one to improve the best-known solution faster [HZ07].

4.2.2 Iterative Beam Search

Iterative Beam Search (IBS) was originally proposed in [Lib+20] and allowed authors to close new instances of the Sequential Ordering Problem (SOP, see Section 1.1). More recently, it was applied to the Permutation Flowshop Problem in [Lib+22]. It was also used in the anytime and problem-independent DP-based solver of [KB23c]: experimental results have shown IBS to be more efficient for proving optimality than five other EAS algorithms when considering nine different problems, including the TSPTW$_{\Sigma t}$.

IBS consists in performing a series of Beam Searches (*i.e.,* breadth-limited breadth-first searches): only the B most promising nodes are considered at each layer of the graph, and the remaining nodes are inadmissibly pruned. Initially, the breadth-limit B is equal to one, and it is doubled at the end of each search iteration. Optimality is proven when a search iteration terminates without having inadmissibly pruned any node.

The main motivation behind the geometrical growth of B is to ensure that each Beam Search considers a significant number of new states compared to all of the previous iterations combined [Lib+20]. Authors note that B should not grow too fast in order not to waste time in the last iteration, which is incomplete when the algorithm is stopped before termination (this last iteration cannot provide a solution when considering either the SOP or the TD-TSPTW$_m$ because final states necessarily belong to the last layer of the state transition graph).

IBS can be seen as a special case of a general strategy called Complete Anytime Beam Search (CABS) [Zha98]: CABS consists in performing a series of searches (either breadth-first or depth-first) using inadmissible pruning rules which are weakened after each iteration. Note that IBS also resembles Restricted DP (see Subsection 2.3.1): IBS differs from RDP because IBS (i) prioritizes states by f-value, and (ii) performs multiple searches with increasing breadth-limits.

Algorithm 4.3: IBS for the TD-TSPTW$_m$

input : Transitions $\mathcal{A}$ of G^P_{ST}, vertex set $\mathcal{V} = \{0, \dots, n\}$,
the departure time e_0 from the origin vertex 0, the deadline l_n of the destination vertex n,
an admissible heuristic h

```
1  let g and f be empty maps associating a value to each subproblem (i, S);
2  let known, current and next be empty sets of subproblems;
3  ub ← l_n + 1;
4  set g(0, {0}) to e_0;
5  add (0, {0}) to known;
6  B ← 1;
7  do
8      inadmissible ← false;
9      current ← {(0, {0})};
10     while current ≠ ∅ do
11         next ← ∅;
12         foreach subproblem (i, S) ∈ current do
13             foreach state s' = (i', S', t') such that (s, s') ∈ A, with s = (i, S, g(i, S)) do
14                 if [(i', S') ∈ known and t' > g(i', S')] or t' + h(s') ≥ ub then continue;
15                 set g(i', S') to t';
16                 ensure that subproblem (i', S') belongs to known;
17                 if final(s') then
18                     ub ← t'; // New solution found
19                 else
20                     ensure that subproblem (i', S') belongs to next;
21                     set f(i', S') to t' + h(s');
22         if |next| ≤ B then
23             current ← next;
24         else
25             current ← the B least f-valued states from next;
26             inadmissible ← true;
27     B ← B · 2;
28 while inadmissible;
```

Instantiation to the TD-TSPTW$_m$. We formalize our instantiation of IBS in Algorithm 4.3 and highlight in blue the differences with AWA* (see Subsection 4.2.1). Because IBS may inadmissibly prune nodes, it has to restart search and consequently to re-expand nodes. Therefore, we do not describe it using *open* and *closed* sets, but instead use a set of *known* subproblems, which contains all subproblems encountered during search. We memorize the best-known value associated to each *known* subproblem in map g in order to avoid considering dominated states. Initially, *known* only contains the initial subproblem $(0, \{0\})$, its g-value is set to e_0, and the breadth-limit B is set to 1 (Lines 4-6).

Loop Lines 7-28 repeats Beam Searches with increasing breadth-limit B as long as states have been inadmissibly pruned. These searches use two additional sets of subproblems, *current* and *next*. Set *current* contains up to B subproblems of the current layer, and *next* their successors. Initially, at L9, *current* only contains the initial subproblem $(0, \{0\})$. Loop L10-26 expands *current* subproblems and stores their successors in *next*: at L22-26, up to B of the most promising subproblems from *next* become the *current* set of the following iteration. If there are more than B subproblems in *next*, the $|next| - B$ least promising subproblems are discarded and the current

search iteration is marked as inadmissible. A Beam Search terminates once set *next* becomes empty.

More precisely, each iteration of loop L12-21 expands the state associated to a subproblem $(i, \mathcal{S})$ from the *current* set. In IBS, states are expanded in a similar way as in AWA*, *i.e.,* a successor $s' = (i', \mathcal{S}', t')$ is discarded if it is dominated or bounded; otherwise, $(i', \mathcal{S}')$ is added to *known* if necessary and its g-value is memorized. A new solution has been found if $(i', \mathcal{S}')$ is a final state; otherwise, $(i', \mathcal{S}')$ is added to *next*.

However, IBS differs from AWA* in an important way: while AWA* can discard a successor state $(i', \mathcal{S}', t')$ if t' is greater than or equal to $g(i', \mathcal{S}')$, IBS – at L14 – can only discard it if t' is *strictly* greater than $g(i', \mathcal{S}')$. This is due to the fact that AWA* guarantees that all successors of $(i', \mathcal{S}')$ have been or will be considered (*i.e.,* $(i', \mathcal{S}')$ either belongs to *open* or to *closed*). On the other hand, IBS cannot provide such a guarantee due to inadmissible pruning (*i.e.,* $(i', \mathcal{S}')$ may have been inadmissibly pruned in previous iterations).

4.2.3 Anytime Column Search

Anytime Column Search (ACS) [Vad+12], like A*, maintains *open* and *closed* sets of states: ACS iteratively expands the B most promising *open* states at each layer of the state transition graph, and proves optimality when the *open* set becomes empty. Recall from Subsection 3.3.3 that ACS is a generalization of Cyclic Best First Search [KSJ09] in which the breadth-limit B is equal to 1.

Instantiation to the TD-TSPTW$_m$. We formalize our instantiation of ACS in Algorithm 4.4 and highlight in blue the differences with AWA* (see Subsection 4.2.1). These two algorithms are very similar, as they only differ in the way they determine the next state to expand.

Instead of maintaining a single *open* set, ACS maintains an *open* set for each layer $k \in [1, |\mathcal{V}| - 1]$ of G^P_{ST}. We note $open[k]$ the set containing *open* subproblems of size k, *i.e.,* subproblems $(i, \mathcal{S})$ such that $|\mathcal{S}| = k$. These sets are initialized at L4-5: $open[1]$ only contains the initial subproblem $(0, \{0\})$, and the remaining *open* sets are empty.

Loop L8-25 is repeated as long as there remain *open* subproblems. Loop L9-25 considers each layer $k \in [1, |\mathcal{V}| - 1]$ such that $open[k]$ is not empty: at L10, up to B of the most promising subproblems in $open[k]$ are placed in a *candidates* set for expansion. Loop L11-25 then expands each subproblem from *candidates* in non-decreasing order of f-values; if necessary, their successors are (i) added to $open[k+1]$ and (ii) removed from *closed*.

Algorithm 4.4: ACS for the TD-TSPTW$_m$

input : Transitions $\mathcal{A}$ of G^P_{ST}, vertex set $\mathcal{V} = \{0, \dots, n\}$, the departure time e_0 from the origin vertex 0, the deadline l_n of the destination vertex n, an admissible heuristic h and a breadth-limit B

```
1  ub ← l_n + 1;
2  let g and f be empty maps associating a value to each subproblem (i, S);
3  let closed be an empty set of subproblems;
4  foreach layer k ∈ [1, |V| − 1] do let open[k] be an empty set of subproblems;
5  add (0, {0}) to open[1];
6  set g(0, {0}) to e_0;
7  set f(0, {0}) to e_0 + h(0, {0}, e_0);
8  while ⋃_{k∈[1,|V|−1]} open[k] ≠ ∅ do
9      foreach layer k ∈ [1, |V| − 1] such that open[k] ≠ ∅ do
10         candidates ← the min (B, |open[k]|) least f-valued subproblems (i, S) from open[k];
11         foreach subproblem (i, S) ∈ candidates, sorted in non-decreasing order of f-values do
12             remove (i, S) from open[k] and add it to closed;
13             foreach state s' = (i', S', t') such that (s, s') ∈ A, with s = (i, S, g(i, S)) do
14                 if [(i', S') ∈ open[k + 1] ∪ closed and t' ≥ g(i', S')] or t' + h(s') ≥ ub then
15                     continue;
16                 g(i', S') ← t';
17                 if final(s') then // ⟺ k = |V| − 1
18                     ub ← t'; // New solution found
19                     foreach layer k ∈ [1, |V| − 1] do
20                         remove from open[k] each subproblem (i'', S'') such that f(i'', S'') ≥ ub;
21                     ensure that (i', S') belongs to closed;
22                 else
23                     if (i', S') ∉ open[k + 1] then add (i', S') to open[k + 1];
24                     if (i', S') ∈ closed then remove (i', S') from closed;
25                     set f(i', S') to t' + h(s');
```

Relationship with IBS. ACS resembles IBS in the sense that it iteratively expands up to B states at each layer of the graph. However, IBS inadmissibly prunes the least promising states and restarts search with an increased breadth-limit for completeness. On the other hand, ACS memorizes these states in *open* sets for later consideration, and the breadth-limit remains constant.

Our formalization of ACS is general enough to also describe an improved version of IBS, which we note IBS$^+$. IBS$^+$, like ACS and contrarily to IBS, memorizes discarded states instead of restarting search, which decreases the costs associated to re-expansions (*i.e.,* the computation of the successor states and their heuristic value). It has the same space complexity as IBS (during the last iteration of IBS, the map which stores g-values contains all subproblems). Algorithm 4.4 formalizes IBS$^+$ when the breadth-limit B is (i) initially set to one, and (ii) doubled at the end of each iteration of loop L8-25. Additionally, because IBS does not require expanding the B most promising states of a given layer in non-decreasing order of f-values, IBS$^+$ expands subproblems from *candidates* in an arbitrary order.

Our improvement from IBS to IBS$^+$ is analogous to an improvement which leads to *Fringe Search* [Bjö+05] when applied to IDA* (see Subsection 3.3.2): instead of discarding the nodes with an f-value greater than the threshold, these nodes are memorized and used as the starting point for

the next iteration: this improves on IDA*'s efficiency by avoiding to start each search iteration from scratch.

In Section 4.3, we (i) describe an implementation suited for relatively low values of B (*i.e.,* for ACS) and (ii) propose an alternative implementation suited for large values of B (*i.e.,* for IBS$^+$).

4.2.4 Anytime Window A*

Anytime Window A* (AWinA*) [ACK07] restricts node expansions based on node depths. Authors defined it by introducing an incomplete search procedure called Window A* (WinA*). This procedure takes as parameter a depth tolerance *tol*, and iteratively expands a most promising *open* state while keeping track of the layer $\overline{k}$ containing the deepest state expanded yet. However, it only considers *open* states located deeper than layer $\overline{k}$, with a tolerance of *tol* layers. In other words, WinA* expands, at each iteration, a most promising *open* state such that its depth is strictly greater than $\overline{k} - tol$. When $tol = 0$, WinA* is purely depth-first, and when *tol* is equal to the depth of the deepest state, it behaves like A*. AWinA* repeatedly calls the WinA* procedure with an increasing tolerance *tol* and proves optimality when the *open* set becomes empty.

Instantiation to the TD-TSPTW$_m$. We instantiate AWinA* to the TD-TSPTW$_m$ in Algorithm 4.5: grey parts of the algorithm are identical to ACS (see Subsection 4.2.3).

Loop L8-30 runs the WA* procedure with increasing tolerance *tol*, ranging from 0 to $|\mathcal{V}| - 1$ (*i.e.,* the depth of the deepest state which needs to be expanded). At L9, $\overline{k}$ is initialized to $-\infty$ as no state has yet been expanded in the current iteration. We note $\underline{k}$ the shallowest layer from which a state may get expanded and initialize it to the layer containing the initial subproblem at L10.

Loop L11-30 iteratively expands a most promising *open* subproblem $(i, \mathcal{S})$ from layers $[\underline{k}, |\mathcal{V}| - 1]$ (ties are broken in favor of smaller depths $|\mathcal{S}|$). When expanding a state located at a depth k greater than $\overline{k}$, we set $\overline{k}$ to k and update $\underline{k}$ accordingly (L15-16): this prevents the current WinA* search from later expanding subproblems belonging to layer range $\left[1, \overline{k} - tol\right]$.

4.2.5 Discussion

In this section, we have instantiated four EAS algorithms (namely, AWA*, IBS, ACS and AWinA*) to the TD-TSPTW$_m$. We have also proposed an improved version of IBS inspired from ACS, which we named IBS$^+$. In Table 4.1, we provide a high-level view of these algorithms by recalling (i) their expansion strategy (see Subsection 3.3.3), (ii) whether or not they restart search, (iii) their parameters and (iv) their main behavior(s).

AWA* has a biased best-first behavior since it is guided by a weighted heuristic, *i.e.,* its behavior depends on the heuristic h used and on the heuristic weight w. ACS, IBS and IBS$^+$ are all based on breadth-constraints: the behavior of ACS depends on the value of its parameter, *i.e.,* the

Algorithm 4.5: AWinA* for the TD-TSPTW$_m$

input : Transitions $\mathcal{A}$ of G^P_{ST}, vertex set $\mathcal{V} = \{0, \dots, n\}$,
the departure time e_0 from the origin vertex 0, the deadline l_n of the destination vertex n,
an admissible heuristic h

$ub \leftarrow l_n + 1$;
let g and f be empty maps associating a value to each subproblem $(i, \mathcal{S})$;
let *closed* be an empty set of subproblems;
foreach layer $k \in [1, |\mathcal{V}| - 1]$ **do** let $open[k]$ be an empty set of subproblems;
add $(0, \{0\})$ to $open[1]$;
set $g(0, \{0\})$ to e_0;
set $f(0, \{0\})$ to $e_0 + h(0, \{0\}, e_0)$;
foreach $tol \in [0, |\mathcal{V}| - 1]$ **do**
  $\overline{k} \leftarrow -\infty$;
  $\underline{k} \leftarrow 1$;
  while $\bigcup_{k \in [\underline{k}, |\mathcal{V}|-1]} open[k] \neq \emptyset$ **do**
    let $(i, \mathcal{S})$ be a least f-valued subproblem from $\bigcup_{k \in [\underline{k}, |\mathcal{V}|-1]} open[k]$;
    $k \leftarrow |\mathcal{S}|$;
    if $k > \overline{k}$ **then**
      $\overline{k} \leftarrow k$;
      $\underline{k} \leftarrow \max\left(\overline{k} - tol + 1, 1\right)$;
    remove $(i, \mathcal{S})$ from $open[k]$ and add it to *closed*;
    foreach state $s' = (i', \mathcal{S}', t')$ such that $(s, s') \in \mathcal{A}$, with $s = (i, \mathcal{S}, g(i, \mathcal{S}))$ **do**
      if $[(i', \mathcal{S}') \in open[k+1] \cup closed$ **and** $t' \geq g(i', \mathcal{S}')]$ **or** $t' + h(s') \geq ub$ **then**
        continue;
      $g(i', \mathcal{S}') \leftarrow t'$;
      if $final(s')$ **then** // $\iff k = |\mathcal{V}| - 1$
        $ub \leftarrow t'$; `// New solution found`
        foreach layer $k \in [1, |\mathcal{V}| - 1]$ **do**
          remove from $open[k]$ each subproblem $(i'', \mathcal{S}'')$ such that $f(i'', \mathcal{S}'') \geq ub$;
        ensure that $(i', \mathcal{S}')$ belongs to *closed*;
      else
        if $(i', \mathcal{S}') \notin open[k+1]$ **then** add $(i', \mathcal{S}')$ to $open[k+1]$;
        if $(i', \mathcal{S}') \in closed$ **then** remove $(i', \mathcal{S}')$ from *closed*;
        set $f(i', \mathcal{S}')$ to $t' + h(s')$;

Name	Ref.	Expansion strategy	Restarts	Parameter	Behavior
AWA*	[ZH02]	Weighted heuristic	–	w	"Biased" Best-First
ACS	[Vad+12]	Breadth-constrained	–	B	$\begin{cases} B \text{ is close to 1: Depth-First} \\ B \text{ tends to } \infty\text{: Breadth-First} \end{cases}$
IBS	[Lib+20]	Breadth-constrained	✓	–	Depth-First $\rightarrow$ Breadth-First*
IBS$^+$	[This thesis]	Breadth-constrained	–	–	Depth-First $\rightarrow$ Breadth-First*
AWinA*	[ACK07]	Depth-constrained	–	–	Depth-First $\rightarrow$ Best-First*

Table 4.1: Overview of the EAS algorithms instantiated to the TD-TSPTW$_m$. *The first iteration of these algorithms is purely depth-first, then search becomes increasingly breadth-first (resp. best-first) in IBS and IBS$^+$ (resp. AWinA*).

breadth-limit B. On the other hand, in both IBS and IBS$^+$, this breadth-limit is not a parameter. Instead, the breadth-limit B is initially equal to one (leading to a depth-first behavior) and B is doubled at the end of each search iteration (*i.e.,* search progressively becomes more breadth-first). The key difference between IBS and IBS$^+$ is that IBS$^+$ avoids restarting search between each increase of B by memorizing states instead of inadmissibly pruning them: this leads to a decrease in the number of re-expansions. Finally, AWinA* is based on depth constraints: the first iteration of this algorithm is purely depth-first and search becomes increasingly best-first (recall that AWinA*, during its last iteration, behaves like A*).

Breadth-constrained and depth-constrained algorithms exploit the layered structure of the graph to obtain a depth-first behavior. Additionally, breadth-constrained algorithms (*i.e.,* ACS, IBS and IBS$^+$) only compare subproblems by f-value within a given layer: this leads to fairer comparisons, as authors of [Pea84] note that systematic heuristic error can be expected to be homogeneous within a layer. AWinA* also benefits from this during early search iterations (later during search, its behavior becomes more best-first). AWA*, on the other hand, uses a single *open* set and prioritizes subproblems using the weighted evaluation function only, *i.e.,* regardless of the layer they belong to.

In absence of TW constraints, ACS with $B = 1$, IBS, IBS$^+$ and AWinA* can provide a first solution after having expanded $|\mathcal{V}| - 1$ states. In AWA*, on the other hand, it depends on the heuristic function h and on the heuristic weight w. Also note that the behavior of ACS remains the same throughout search: each iteration expands $O(B \cdot |\mathcal{V}|)$ states, *i.e.,* each iteration performs a constant amount of work. This provides ACS with the ability to frequently report new solutions, depending on (i) the breadth-limit B and (ii) the instance size $|\mathcal{V}|$. On the other hand, in both variants of IBS and in AWinA*, the initial search behavior (*i.e.,* depth-first) favors convergence speed, and these algorithms later adopt a behavior favoring optimality proofs (either by becoming more breadth-first or more best-first).

4.3 Implementation of EAS algorithms

In this section, we discuss of key implementation decisions regarding the algorithms we instantiated to the TD-TSPTW$_m$ in Section 4.2. These decisions mainly involve data structures and algorithms: their scope of application is not limited to the TD-TSPTW$_m$, *i.e.,* they can be used when searching for optimal paths in any layered directed acyclic graph.

Note that such concerns are often disregarded: indeed, none of the publications introducing the algorithms considered in this chapter (see Table 4.1) discuss data structure choices. In this section, we show that (i) these choices have an impact on time complexity and (ii) data structures commonly used to implement A* (*i.e.,* min-heaps) are not necessarily the best suited to implement all of the EAS algorithms we consider. We experimentally study the influence of some of these decisions in Section 6.4.

We first discuss implementation choices of AWA*, ACS and AWinA*: these three algorithms – and their implementation – are very close to A* because (i) they maintain *open* and *closed* sets of states and (ii) they expand a fixed number of states at each iteration. We then propose an efficient implementation of IBS$^+$ and describe two possible implementations of the original IBS algorithm. Finally, we introduce an efficient way to determine the layer which contains the next state to expand in AWinA* according to depth constraints.

4.3.1 A*-like algorithms

A*-like algorithms require to efficiently (i) keep track of *open* and *closed* states, including their associated g-value, and (ii) keep track of the *open* states and their associated f-value.

In AWA*, ACS and AWinA*, we maintain the g-values of *open* and *closed* subproblems using hash maps. More precisely, we proceed in the same way as in the exact algorithm presented in Section 4.1, *i.e.,* we use a hash map $m[k]$ for each layer $k \in [1, |\mathcal{V}|]$ of the graph. Map $m[k]$ associates its g-value to each to each subproblem $(i, \mathcal{S}) \in \mathit{open} \cup \mathit{closed}$ such that $|\mathcal{S}| = k$. This allows one to efficiently perform accesses, insertions and updates (we discuss these operations and their time complexities more in depth when describing the implementation of Algorithm 4.1 in Section 4.1). We use one map for each layer of the graph instead of a single map because the size $|\mathcal{S}|$ of a given problem $(i, \mathcal{S})$ is a trivial and inexpensive way to discriminate between subproblems: this tends to result in fewer collisions and leads to maintaining smaller maps.

We now discuss the second key requirement of A*-like algorithms, *i.e.,* maintaining the *open* set. We first describe how it may be done in the context of A* using a min-heap (including "lazy" updates of this heap) and then detail the minor adaptations necessary to implement AWA*, ACS and AWinA*.

Representing of the *open* set in A*. A* expands a single *open* subproblem of minimal f-value at each iteration. A min-heap is therefore an obvious and common choice to represent the *open* set. More precisely, this min-heap contains tuples $(f, i, \mathcal{S})$ compared by f-value. The main operations performed on *open* are then achieved in $O(\log |\mathit{open}|)$ time, *i.e.,* extracting the most promising subproblem, inserting a new subproblem $(i, \mathcal{S})$ and its associated f-value, or decreasing the f-value of an existing subproblem.

Note that decreasing the f-value of an arbitrary element of the heap requires knowing its position, *i.e.,* using an *indexed min-heap*. This data structure combines a hash map and min-heap: the hash map associates to each *open* subproblem $(i, \mathcal{S})$ a reference r_{heap} to its associated heap entry. The heap contains a couple (f, r_{map}) for each *open* subproblem $(i, \mathcal{S})$, where r_{map} is a reference to the hash map entry associated to $(i, \mathcal{S})$. Then, one can update the f-value of an arbitrary *open* subproblem in $O(\log |\mathit{open}|)$ time. However, it requires maintaining references r_{heap} in the hash map every time a heap operation is performed. Finally, note that references r_{heap} can either be stored (i) in the map associating to each known subproblem is g-value, or (ii) in a separate map.

Lazy updates of f-values in the heap. We avoid maintaining these references each time heap elements are rearranged by adopting a lazy approach based on a min-heap containing tuples $(f, i, \mathcal{S}, t)$. Instead of updating the heap entry associated to $(i, \mathcal{S})$ when decreasing its g-value and therefore its f-value, we insert a new entry in the heap: in this entry, t is the best-known value associated to subproblem $(i, \mathcal{S})$ at the time of insertion. When extracting a tuple $(f, i, \mathcal{S}, t)$ from the heap, we discard obsolete entries and only expand state $s = (i, \mathcal{S}, t)$ if $t = g(i, \mathcal{S})$.

Proceeding this way leads to a simpler implementation as it eliminates the need of performing random accesses in the heap, and thus the need of memorizing and maintaining the position of the heap entry associated each *open* subproblem. This strategy is not specific to A*: it has been used to maintain the *open* set of Dijkstra's algorithm and is sometimes called *lazy deletion* (see, *e.g.,* [HHE18, Section 4.4.3] and [SW11, Section 4.4]). With this lazy implementation, the space requirements of the *open* set are in $O(\log |\mathcal{A}|)$ instead of $O(\log |\mathcal{N}|)$. The time complexity for heap operations does not change even when considering dense graphs, given $O\left(\log |\mathcal{N}|^2\right) = O(2 \log |\mathcal{N}|)$.

We conducted experiments to compare both the lazy and the classic implementations on the TD-TSPTW$_m$. Results did not show significant differences both in terms of space and time: while the lazy algorithm may maintain a larger *open* set, it has better memory locality and requires less time for bookkeeping than the classic implementation (these conclusions may differ when considering problems in which storing state variables require more space). Note that we compared these two implementations by considering Anytime Column Search (see Subsection 4.2.3), in which the bookkeeping overhead of the classic implementation is expected to be lower than in A*, given the *open* set is partitioned into multiple smaller min-heaps.

Lazy removal of bounded subproblems. Recall that in the algorithms described in Section 4.2, we ensured that the *open* set never contains bounded subproblems, *i.e.,* subproblems $(i, \mathcal{S})$ for which $f(i, \mathcal{S}) \geq ub$. When the *open* set is implemented as a heap, removing these subproblems requires filtering the heap and restoring the heap property (*i.e.,* $O(|open|)$ time) each time the upper bound ub is decreased. We proceed in the same way as authors of [HZ07], who avoid this operation by adopting a lazy approach: this approach consists in allowing bounded subproblems to belong to the heap, and discarding them only upon extraction from *open*.

Representation of *open* sets in AWA*, ACS and AWinA*. We implemented AWA*, ACS and AWinA* using min-heaps with lazy updates of f-values and lazy removal of bounded subproblems. We now describe how the implementation of these algorithms differs from the one of A*.

In AWA*, elements of the heap representing the *open* set contain an additional value f_w, which corresponds to the value of the weighted guide function. More precisely, heap elements are tuples $(f_w, f, i, \mathcal{S}, t)$ compared according to the value of f_w. Note that it is also necessary to store the value of the unweighted guide function f in these tuples in order to perform lazy deletion.

Recall that ACS and AWinA* do not use a single *open* set: instead, these algorithms maintain an *open* set for each layer $k \in [1, |\mathcal{V}| - 1]$ of the graph. Consequently, we implement them using

$|\mathcal{V}| - 1$ min-heaps which contain tuples $(f, i, \mathcal{S}, t)$.

Note that a last building block is missing to obtain an efficient of implementation of AWinA*: it consists in determining which *open* set contains the next state to expand according to the current depth constraints. We propose an efficient solution to this problem in Subsection 4.3.3.

4.3.2 Iterative Beam Search

In this section, we first propose an efficient way of implementing IBS$^+$, the non-restarting variant of IBS we proposed in Subsection 4.2.3. Then, we describe the original implementation of IBS and propose a more efficient alternative. We compare experimentally the resulting algorithms in Section 6.4, which shall also allow us to evaluate the performance impact of avoiding re-expansions.

Non-restarting variant IBS$^+$. Recall from Subsection 4.2.3 that the key difference between ACS and IBS$^+$ is that B is constant in ACS whereas it doubles after each iteration in IBS$^+$. We have seen in Subsection 4.3.1 that min-heaps are an obvious choice when few subproblems are expanded at each iteration. This is not the case when considering IBS$^+$: we propose to implement it efficiently by using a selection algorithm instead of min-heaps.

Quickselect [Hoa61] is a selection algorithm which allows one to compute the k^{th} smallest element of an array of size n in $O(n)$ average time [AHU74, Section 3.7]: this algorithm has a quadratic worst-case time complexity, which may be improved to $O(n)$ using an introspective selection algorithm [Mus97].

More concretely, we propose to implement IBS$^+$ using two hash maps for each level $k \in [1, |\mathcal{V}| - 1[$. The first hash map associates to each subproblem $(i, \mathcal{S}) \in \mathit{closed}$ its g-value. The second map associates to each subproblem $(i, \mathcal{S}) \in \mathit{open}$ a couple (g, f), *i.e.,* both its g-value and its f-value. We then use a selection algorithm to obtain in $O(|\mathit{open}[k]|)$ average time the B most promising *open* subproblems from layer k. Note that this operation rearranges the elements of the input array: consequently, we run it on a copy of $\mathit{open}[k]$, which requires $O(|\mathit{open}[k]|)$ extra space.

Using such hash maps also allows us, in average constant time, to (i) insert subproblems in *open*, (ii) move subproblems from *open* to *closed* (and the reverse) and (iii) update the g-value and f-value of a subproblem.

Recall that ACS represents *open* sets as min-heaps: in this case, all of these operations take $O(|\mathit{open}[k]|)$ time. Also, note that expanding $O(B)$ states from a given layer k require performing $O(B \cdot |\mathcal{V}|)$ operations on $\mathit{open}[k+1]$, given a state has $O(|\mathcal{V}|)$ successors.

We note IMPL-HEAP the heap-based implementation used in ACS, and IMPL-SELECT the implementation based on a selection algorithm. We summarize in Table 4.2 the time complexities of these two implementations. While both these implementations could be used for IBS$^+$, it is clear that (i) IMPL-SELECT is more efficient than IMPL-HEAP for inserting subproblems in $\mathit{open}[k+1]$ and (ii) IMPL-HEAP should be preferred when B is small, given the size of *open* sets is in $O(|\mathcal{N}|)$.

Note also that IMPL-SELECT can be expected to be more efficient than IMPL-HEAP for extracting B subproblems from $open[k]$ when $B > \frac{|open[k]|}{\log|open[k]|}$

Because B grows geometrically in IBS$^+$, we implement IBS$^+$ using IMPL-SELECT. In Section 6.4, we experimentally compare the performance of IBS$^+$ under both implementations and show that using a selection algorithm instead of min-heaps leads to improved performance.

Operation	IMPL-HEAP	IMPL-SELECT
Extract $O(B)$ subproblems from $open[k]$	$O(B \log \lvert open[k] \rvert)$	$O(\lvert open[k] \rvert)^{\text{a}}$
Add $O(B \cdot \lvert\mathcal{V}\rvert)$ subproblems to $open[k+1]$	$O(B \cdot \lvert\mathcal{V}\rvert \log \lvert open[k+1] \rvert)$	$O(B \cdot \lvert\mathcal{V}\rvert)^{\text{b}}$

Table 4.2: Time complexities of key operations of implementations IMPL-HEAP and IMPL-SELECT. Caveats: [a]average; [b]amortized.

Original IBS. We now describe two possible implementations of the original IBS algorithm (see Subsection 4.2.2). These implementations only vary in the way set *next* is represented. We name them IBS$_{\text{select}}$ and IBS$_{\text{heap}}$ because – similarly to what we have seen for IBS$^+$– they either use a selection algorithm or a min-heap: IBS$_{\text{select}}$ is an implementation we propose, and IBS$_{\text{heap}}$ is similar to the original implementation of [Lib+20]. We only briefly describe these implementations because (i) they are less efficient than IBS$^+$ due to restarts and (ii) they share many similarities with it. Also, in Section 6.4, we experimentally compare IBS$^+$ to IBS$_{\text{select}}$ and IBS$_{\text{heap}}$ to quantify the variations in performance between these three variants.

For both IBS$_{\text{select}}$ and IBS$_{\text{heap}}$, we use a hash map m_{known} associating to each *known* subproblem its g-value (this is akin to the implementation of A*-like algorithms and IBS$^+$, except that these algorithms operate on *open* and *closed* sets instead of a *known* set).

In IBS$_{\text{select}}$, we use a hash map m_{next} associating its f-value to each subproblem of set *next*. We insert elements (resp. update elements) of m_{next} in amortized constant (resp. average constant) time. As in IBS$^+$, we obtain the B most promising subproblems from *next* in $O(|next|)$ average time using a selection algorithm.

IBS$_{\text{heap}}$ corresponds to the original implementation of [Lib+20]. Authors do not provide implementation details but their solver is open source: set *next* is implemented as a priority queue of bounded size B, so each insertion requires $O(\log B)$ time (including the removal of the worst element when the queue is full). Such a choice reduces the space requirements of *next* to $O(B)$, but does not change the overall space complexity of the algorithm, since map m_{known} associating to each *known* subproblem its g-value requires $O(|\mathcal{N}|)$ space (note that set *next* also requires exponential space because of the exponential growth of B).

Note that in the implementation of [Lib+20], the priority queue may contain duplicate elements. To better understand the motivation behind this choice, it is necessary to study more in depth the solving approach. Authors propose to solve the SOP through tree search: the nodes of this tree are partial solutions, *i.e.,* permutations of subsets of vertices. This tree search is augmented with the concept of dominance from DP, *i.e.,* by pruning the worst of any two partial solutions

which visit the same set of vertices and end on the same vertex. Experimental results of [Lib+20] highlight the benefits of enhancing tree search with such dominance pruning: in this thesis, we go one step further by considering the DP state space, which is a DAG instead of a tree.

Discussion In this section, we have proposed an efficient implementation of IBS$^+$ which is not based on min-heaps but on a selection algorithm. We have also described two implementations of IBS (which, unlike IBS$^+$, has to restart search to ensure completeness): IBS$_{\text{heap}}$ models set *next* using a min-heap which may contain duplicate subproblems. In IBS$_{\text{select}}$ on the other hand, elements of *next* are unique and a selection algorithm is used to efficiently obtain B of its most promising elements.

We experimentally compare these algorithms and their implementations in Section 6.4: these results shall also allow us to study the importance of avoiding re-expansions.

4.3.3 Anytime Window A*

AWinA* requires determining which *open* subproblem is most promising amongst layers $k \in [\underline{k}, |\mathcal{V}| - 1]$ at L12 of Algorithm 4.5. We propose to do it efficiently by maintaining an additional min-heap q associating to each layer k the minimum f-value of subproblems in $open[k]$. More precisely, q contains couples $(f_{\min}, k)$ compared lexicographically for each layer $k \in [1, |\mathcal{V}| - 1]$. When $open[k]$ is empty or when $k < \underline{k}$, the $f_{\min}$ value associated to k is set to infinity. Otherwise, $f_{\min}$ is the minimum f-value of suproblems in $open[k]$, *i.e.,* $f_{\min} = \min_{(i,\mathcal{S}) \in open[k]} f(i, \mathcal{S})$.

The smallest element of q is obtained in constant time. When this smallest element has $f_{\min} = \infty$, then $open[k]$ is empty for all layers $k \geq \underline{k}$. Otherwise, value k of this smallest element corresponds to the layer containing the next *open* subproblem to expand. We efficiently maintain the min-heap q by:

- initializing it in $O(|\mathcal{V}|)$ time at the beginning of the loop L8,
- increasing in $O(\log |\mathcal{V}|)$ time the value $f_{\min}$ associated to layer k when an element is removed from $open[k]$ at L12,
- decreasing in $O(\log |\mathcal{V}|)$ time the value $f_{\min}$ associated to layer $k+1$ when an element is inserted into $open[k+1]$ at L28,
- setting $f_{\min}$ to infinity for each layer k such that $k \leq \underline{k}$ in $O(|\mathcal{V}| \log |\mathcal{V}|)$ time whenever $\underline{k}$ is increased at L16.

Note that performing random accesses in q (*i.e.,* increasing or decreasing the $f_{\min}$-value associated to an arbitrary layer k) requires using an array of size $|\mathcal{V}|$ which maps layers k to their position in q, and maintaining these references each time the elements of q are rearranged.

Discussion. Our description of AWinA* differs from the original formulation of [ACK07], but it has the same semantics in the context of a layered DAG. Our formulation is arguably simpler and it is also very similar to our instantiation of ACS. More importantly, the original article does not provide implementation details regarding data structures: authors use a single *open* set and a *suspended* set. This *open* set may contain states from layers k such that $k < \underline{k}$: these states are lazily removed from *open*, *i.e.,* when a state from layer $k < \underline{k}$ is extracted from *open*, it is added to the *suspended* set instead of being expanded. At the end of a WinA* search, *suspended* states are moved to the *open* set to be considered in the next search iteration.

Each lazy removal from *open* takes $O(\log |open|)$ time and building a min-heap from the *suspended* set takes $O(|suspended|)$ time: the method we propose avoids these overheads by maintaining k *open* sets and using a min-heap to efficiently determine which *open* set contains the next state to expand, according to $\underline{k}$.

4.4 Computation of lower bounds h

Recall from Section 3.3 that $h^*(s)$ denotes the cost of the shortest path from a given state s to a final state in a state transition graph. In the context of the TD-TSPTW$_m$, given a state $s = (i, \mathcal{S}, t)$, $h^*(s)$ represents the time required to reach the destination vertex n when departing from vertex i at time t and visiting every vertex in $\mathcal{V} \setminus \mathcal{S}$ within its TW. Of course, it is possible that no final state is reachable from state s, *i.e.,* that it is not possible to complete the partial tour represented by s while satisfying TW constraints: in this case, $h^*(s) = \infty$.

An admissible heuristic $h(s)$ is a lower bound on $h^*(s)$. Lower bounds are used in the algorithms instantiated in Section 4.2 in order to (i) determine the most promising state to expand, and (ii) admissibly prune states which are proven to lead to an infeasible solution, or to a solution worse than or equivalent to the current best-known.

In this section, we describe three lower bounds for the TD-TSPTW$_m$ which provide different trade-offs between computational cost and tightness. These bounds are computed in polynomial time by solving relaxations of the shortest Hamiltonian path problem in a graph which loosely represents TD travel times and TWs. We first define constant edge costs and the graph associated to a given state $(i, \mathcal{S}, t)$. We then introduce the lower bounds h_{FEA}, h_{OIA}, and h_{MSA}: h_{FEA} is a new bound whereas h_{OIA} and h_{MSA} combine h_{FEA} with classic TSP bounds.

4.4.1 Definition of constant costs

The travel time on a given edge (j,k) depends on the departure time from j, which is not known exactly when computing $h(s)$. To ensure that $h(s)$ is a lower bound, we compute a lower bound $\underline{c}_{j,k}$ on the cost of every usable edge $(j,k) \in \mathcal{E}$. Lower bound $\underline{c}_{j,k}$ may be defined straightforwardly by considering the minimum travel time from j to k when departing from j during its TW, *i.e.*, $\underline{c}_{j,k} = \min_{t \in [e_j, l_j]} c_{j,k}(t)$.

Notice however that traveling from j to k while arriving on k no later than l_k might not be possible during the entire time interval $[e_j, l_j]$: indeed, there may exist a time $t' \in [e_j, l_j]$, such that departing from j later than t' leads to arriving on k later than its deadline l_k. We call time t' the *Latest Departure Time* (LDT) of edge (j,k) and note it $LDT(j,k)$. Given an edge $(j,k) \in \mathcal{E}$, $LDT(j,k)$ denotes the latest time at which one can depart from j and arrive on k while satisfying TW constraints at both vertices, *i.e.*,

$$LDT(j,k) = \min\left(l_j, a_{j,k}^{-1}(l_k)\right) \tag{4.6}$$

Therefore, a tighter lower bound on the cost of edge (j,k) may be defined as the minimum travel time from j to k for each departure time $t \in [e_j, LDT(j,k)] \subseteq [e_j, l_j]$. Additionally, because we consider the makespan objective, we include in $\underline{c}_{j,k}$ the waiting time on k when arriving earlier than e_k, *i.e.*,

$$\underline{c}_{j,k} = \min_{t \in [e_j, LDT(j,k)]} \left(c_{j,k}(t) + \max\left(0, e_k - a_{j,k}(t)\right)\right) \tag{4.7}$$

These constant costs are precomputed and updated every time TWs are tightened using the rules described in Section 1.3 (we shall describe when we use these rules in Subsection 5.2.1).

4.4.2 Definition of the graph G_s used to compute $h(s)$

We now define the digraph $G_s = (\mathcal{V}_s, \mathcal{E}_s)$ associated to a state $s = (i, \mathcal{S}, t)$ which we use to compute $h(s)$. The vertex set $\mathcal{V}_s$ of G_s contains the current vertex i and the vertices left to be visited, *i.e.*, $\mathcal{V}_s = (\mathcal{V} \backslash \mathcal{S}) \cup \{i\}$. Edge set $\mathcal{E}_s$ may be straightforwardly defined as $\mathcal{E}_s = \mathcal{E} \cap ((\mathcal{V}_s \backslash \{n\}) \times (\mathcal{V}_s \backslash \{i\}))$, given $\mathcal{E}$ contains edges which may be used in a feasible solution and the path must start from i and end on n.

When considering constant travel costs $\underline{c}$, the cost of the shortest Hamiltonian path from vertex i to vertex n in G_s is a lower bound on $h^*(s)$, given (i) travel costs are never overestimated, and (ii) TWs are loosely represented through removal of arcs proven to lead to TW violations.

In order to compute tighter bounds, we refine set $\mathcal{E}_s$ to represent TWs more accurately by exploiting state information, *e.g.*, the fact that vertices in $\mathcal{S}$ have already been visited and that travel must happen at time t or later, starting from vertex i. We now introduce three new rules which remove from $\mathcal{E}_s$ edges that cannot be used when the current state is s.

The first two rules are applied on edges which start from vertex i:

- Because the path starts from i, we remove any edge (i, k) such that vertex k has yet unvisited predecessors in $\mathcal{R}$ (*i.e.*, $pred(k) \not\subseteq \mathcal{S}$, where $pred(k) = \{j \in \mathcal{V} : (j, k) \in \mathcal{R}\}$):

$$\text{Rule 1} = \mathcal{E}_s \leftarrow \mathcal{E}_s \setminus \{(i, k) \in \mathcal{E}_s : pred(k) \not\subseteq \mathcal{S}\} \tag{4.8}$$

  Note that this rule removes edge (i, n) from $\mathcal{E}_s$ whenever $|\mathcal{V}_s| > 2$, given precedence constraints ensure that the destination vertex n is the last one to be visited.

- Because vertices must be visited before their deadline, we remove any edge (i, k) such that vertex k cannot be reached on time when departing from vertex i at time t, *i.e.*,

$$\text{Rule 2} = \mathcal{E}_s \leftarrow \mathcal{E}_s \setminus \{(i, k) \in \mathcal{E}_s : t > LDT(i, k)\} \tag{4.9}$$

A third rule is applied on edges which start from a vertex j distinct from i, *i.e.*, $j \in \mathcal{V}_s \setminus \{i\}$. Before traveling on these edges, it is necessary to travel from i to j: consequently, the departure time from vertex j is lower bounded by $a_{i,j}(t)$, provided triangle inequality holds. However, as this filtering is expensive to perform, we do it once for all possible vertices $j \in \mathcal{V}_s \setminus \{i\}$. We therefore consider a lower bound t' which is valid for all of these vertices, *i.e.*, $t' = \min_{(i,j)\in\mathcal{E}_s} a_{i,j}(t)$. We then use time t' to remove all edges (j, k) such that k cannot be reached on time when departing from j at time t':

$$\text{Rule 3} = \mathcal{E}_s \leftarrow \mathcal{E}_s \setminus \{(j, k) \in \mathcal{E}_s : j \neq i \wedge t' > LDT(j, k)\} \tag{4.10}$$

Similarly to the TW constraint propagation rules presented in Section 1.3, these rules aim to prove that some arcs cannot be used because of TW constraints. However, as we shall see in Section 5.2, TW constraint propagation rules are used at the scale of the entire instance (i) in preprocessing and (ii) each time the upper bound ub is decreased. On the other hand, the rules introduced in this section exploit the information available in a given state $s = (i, \mathcal{S}, t)$ and only consider a subset of vertices $\mathcal{V}_s$ which are constrained to be visited at time t or later. Efficiently implementing these rules is critical as they are used every time $h(s)$ is computed.

Implementation. Building graph G_s requires $O\left(|\mathcal{V}_s|^2\right)$ comparisons when removing arcs from $\mathcal{E}_s$ based on their LDT. 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. We update these sets when propagating TW constraints, given TWs become tighter and edges may get removed from $\mathcal{E}$. In Rules 2 and 3, when seeking to remove from $\mathcal{E}_s$ arcs for which the LDT is smaller than some time t, we search for the smallest time $t' \in RT$ such that $t' \geq t$ and only retain arcs present both in $\mathcal{E}_s$ and in $\mathcal{E}_{t'}$ (note that if $t > \max(RT)$, then no arcs can be used).

Proceeding this way allows us to compute graph G_s in $O(|\mathcal{V}_s|)$ time, provided $\mathcal{V}$ contains no

more than 64 elements. This strategy is analogous to the one used in Section 4.1 when efficiently implementing the feasible action function F (note that sets $\mathcal{E}_t$ can also be exploited to efficiently rule out infeasible actions when computing F).

4.4.3 Feasibility bound h_{FEA}

This bound performs a simple feasibility check in 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 there exists no Hamiltonian path from i to n in G_s and it is thus proven that s cannot lead to a feasible solution. In this case, $h_{\text{FEA}}(i, \mathcal{S}, t) = \infty$. Otherwise, $h_{\text{FEA}}(i, \mathcal{S}, t) = 0$.

Provided $\mathcal{V}$ contains no more than 64 elements, these feasibility checks are achieved in $O(|\mathcal{V}_s|)$ time by leveraging the fact that the adjacency matrix representing $\mathcal{E}_s$ is encoded as bitsets (verifying that a vertex does not have outgoing or incoming edges simply requires testing bitset emptiness).

We experimented with other feasibility checks, such as (i) checking that each vertex in $\mathcal{V}_s \setminus \{i\}$ is reachable from i (with a linear-time graph traversal), or (ii) checking that the set of strongly connected components of G_s has a single topological ordering (given the existence of multiple topological orderings implicates the inexistence of a Hamiltonian path). Both these feasibility checks were implemented but discarded as they did not bring enough benefits relatively to their computational cost.

4.4.4 Outgoing/Incoming Arcs bound h_{OIA}

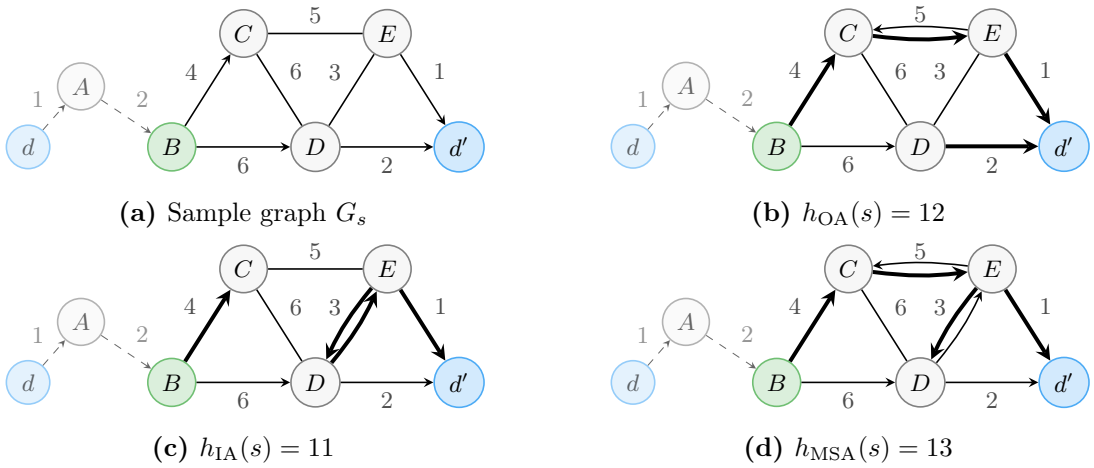
This bound is an adaptation to the TD-TSPTW of the I/O bound used by [Lib+20] for the Sequential Ordering Problem. It is weaker than the assignment relaxation (*i.e.,* the minimum assignment in the bipartite graph between $\mathcal{V}_s \setminus \{n\}$ and $\mathcal{V}_s \setminus \{i\}$). Indeed, it 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, *i.e.,* it is computed in linear time with respect to the number of edges in $\mathcal{E}_s$. Because G_s is not necessarily symmetric, we combine two different bounds: h_{OA}, which considers Outgoing Arcs, and h_{IA}, which considers Incoming Arcs. More precisely, h_{OA} computes the sum of the minimum-weight outgoing arc for each vertex $j \in \mathcal{V}_s \setminus \{n\}$, *i.e.,*

$$h_{\text{OA}}(i, \mathcal{S}, t) = \sum_{j \in \mathcal{V}_s \setminus \{n\}} \min_{(j,k) \in \mathcal{E}_s} \underline{c}_{j,k} \tag{4.11}$$

whereas h_{IA} considers incoming arcs, *i.e.,*

$$h_{\text{IA}}(i, \mathcal{S}, t) = \sum_{k \in \mathcal{V}_s \setminus \{i\}} \min_{(j,k) \in \mathcal{E}_s} \underline{c}_{j,k} \tag{4.12}$$

Finally, given a state $s = (i, \mathcal{S}, t)$, we define $h_{\text{OIA}}(s) = \max\{h_{\text{FEA}}(s), h_{\text{OA}}(s), h_{\text{IA}}(s)\}$ and note that this value can be computed using a single traversal of edge set $\mathcal{E}_s$.



(a) Sample graph G_s **(b)** $h_{\mathrm{OA}}(s) = 12$

(c) $h_{\mathrm{IA}}(s) = 11$ **(d)** $h_{\mathrm{MSA}}(s) = 13$

Figure 4.3: Comparison of the shortest Hamiltonian path problem relaxations computed by h_{OA}, h_{IA} and h_{MSA}. We consider the subgraph G_s associated to state $s = (B, \{d, A, B\}, 3)$, with vertices $\mathcal{V}_s = \{B, C, D, E, d'\}$. We assume all edges are usable except (B, E) and (C, d'). Thick edges belong to the solution of a given relaxation. The optimal Hamiltonian path from B to d' in G_s is $\langle B, C, D, E, d' \rangle$ and has cost 14.

We illustrate a sample graph G_s associated to state $s = (B, \{d, A, B\}, 3)$ in Figure 4.3. We highlight in Figure 4.3b (resp. Figure 4.3c) the edges selected by h_{OA} (resp. h_{IA}). The optimal Hamiltonian path from B to d' in G_s has cost 14 but it is not necessarily feasible, either due to (i) underestimated TD travel times or (ii) loosely represented TWs ; $h_{\mathrm{OIA}}(s) = 12$ is a lower bound on the cost of this Hamiltonian path. We can therefore conclude that a feasible path visiting each vertex of $\{d, A, B\}$ and ending on B at time 3 will require at least 12 time units to visit the remaining vertices while satisfying constraints: it will consequently end on vertex d' at time 15 or later.

4.4.5 Minimum Spanning Arborescence bound h_{MSA}

The *Minimum Spanning Arborescence* (MSA) is a classic relaxation of the Asymmetric TSP [RT12]. It relaxes the constraint that each vertex must have exactly one successor. More precisely, given a digraph $G = (\mathcal{V}, \mathcal{E})$ and a root vertex $u \in \mathcal{V}$, a spanning arborescence of G rooted at u is a subgraph $G' = (\mathcal{V}, \mathcal{E}')$ such that, for any vertex $v \in \mathcal{V} \setminus \{u\}$, there is exactly one directed path from u to v in G'. An MSA is a spanning arborescence for which the sum of all edges costs is minimum. Obviously, any Hamiltonian path starting from u is also a spanning arborescence rooted at u: therefore, the cost of an MSA rooted at u is a lower bound on the cost of the shortest Hamiltonian path starting from u.

We extend the MSA relaxation to the TD-TSPTW using the cost lower bound $\underline{c}$ and the graph G_s. More precisely, given a state $s = (i, \mathcal{S}, t)$, $h_{\mathrm{MSA}}(s)$ computes the cost of the MSA rooted at i in G_s (see Figure 4.3d). We set $h_{\mathrm{MSA}}(s) = \infty$ when (i) no such arborescence exists (*i.e.,* there exists at least one vertex in $\mathcal{V}_s \setminus \{i\}$ that cannot be reached from i) or (ii) a vertex in $\mathcal{V}_s \setminus \{n\}$ has no outgoing arc. The MSA is computed in $O(|\mathcal{E}_s| \log |\mathcal{V}_s|)$ time with the algorithm of [Gab+86].

4.4.6 Discussion

In this section, we have introduced constant lower bounds on travel costs $\underline{c}$ and a digraph G_s associated to a given state $s = (i, \mathcal{S}, t)$. The three lower bounds we propose either consist (i) in computing relaxations of the shortest Hamiltonian path problem in G_s using weights $\underline{c}$, or (ii) in trying to prove that no such path exists. These bounds are computed in polynomial time and offer different trade-offs between computational cost and tightness.

We shall see in Subsection 5.2.1 that we decrease the deadline l_n of the destination vertex n and propagate TW constraints each time a new best-known solution is found. This allows us to tighten TWs (and therefore LDTs) and to refine sets $\mathcal{E}$ and $\mathcal{R}$, which in turns allows us to compute tighter lower bounds, given (i) graph G_s represents TW constraints more accurately and (ii) lower bounds $\underline{c}$ on TD travel times become tighter.

Relationships with existing works. Note that the rules we use to tighten the graph G_s associated to a state s by removing edges from $\mathcal{E}_s$ resemble *Post Feasibility Tests* 1 and 3 proposed by [Dum+95]. However, they were originally used to discard the transition between two states s and s' whenever s' is guaranteed not to lead to a feasible solution. Here, we exploit these rules in a broader context, *i.e.,* we generalize them to build graph G_s: while h_{FEA} only uses this graph to try to prove that s cannot lead to a feasible solution, h_{OIA} and h_{MSA} also exploit it to compute a lower bound on $h^*(s)$.

Also, h_{OIA} resembles the I/O lower bound used by [Lib+20] to solve the SOP: the state transition graph of the SOP (see Section 1.1) is similar to the one of the TD-TSPTW$_m$, except that states are couples $(i, \mathcal{S})$. It consists in defining $h_{\text{I/O}}(i, S) = \max\left(\sum_{j \in (\mathcal{V} \setminus \mathcal{S}) \cup \{i\}} c_j^{out}, \sum_{j \in (\mathcal{V} \setminus \mathcal{S}) \cup \{n\}} c_j^{in}\right)$, where c_j^{out} and c_j^{in} are constants respectively representing the minimum cost of arcs leaving (resp. entering) a vertex $j \in \mathcal{V} \setminus \{n\}$ (resp. $j \in \mathcal{V} \setminus \{0\}$). This bound recognizes that when a vertex j precedes another vertex k, arc (k, j) cannot be used in a feasible solution and is therefore not included in the computation of c_k^{out} and c_j^{in}. This bound is an order cheaper to compute than h_{OIA}, but it is not as tight given constants c_j^{out} and c_j^{in} do not depend on the vertices already visited in a given state $(i, \mathcal{S})$. More recently, authors of [KB23c] adapted the I/O bound to tackle the TSPTW$_{\Sigma t}$ with their problem-agnostic solver based on DP: note that in this adaptation, c_j^{out} and c_j^{in} are constants which (i) do not depend on the current state $(i, \mathcal{S}, t)$ and (ii) fail to recognize that some edges can never be used in a feasible solution. In other words, authors defined these constants as $c_j^{out} = \min_{k \in \mathcal{V} \setminus \{j\}} c_{j,k}$ and $c_j^{in} = \min_{k \in \mathcal{V} \setminus \{j\}} c_{k,j}$.

Future works. Note that h_{MSA} is more informed than h_{IA}, but it is not more informed than h_{OA} (and, consequently, than h_{OIA}): in order for h_{MSA} to be more informed than h_{OIA}, it could be defined as the maximum of the cost of (i) the MSA rooted at i in G_s, and (ii) the MSA rooted at n in the transpose of G_s (*i.e.,* the graph in which all arcs of $\mathcal{E}_s$ are reversed). A cheaper alternative in terms of computation time could be to define it as the maximum value of h_{OA} and h_{MSA}.

Another way to compute tighter lower bounds could be to use tighter relaxations $\underline{c}$ of TD travel costs: in Subsection 4.4.2, when removing arcs from the edge set $\mathcal{E}_s$ associated to a given state $s = (i, \mathcal{S}, t)$, we have seen that arcs leaving vertex i can only be used at time t, while the remaining arcs can only be used at a time $t' > t$ or later. Consequently, for each arc $(j, k) \in \mathcal{E}$, travel costs estimations $\underline{c}_{j,k}$ could be defined as a function of the earliest possible departure time from j. More precisely, $\underline{c}_{j,k}$ would be a non-decreasing function defined on interval $[e_j, LDT(j,k)]$, *i.e.,* $\underline{c}_{j,k}(t) = \min_{t' \in [t, LDT(j,k)]} c_{j,k}(t')$.

4.5 Discussion

In this chapter, we have instantiated four EAS algorithms related to A* to solve the TD-TSPTW$_m$ and studied their similarities and differences: this led us to propose IBS$^+$, an improved version of IBS inspired by ACS. Also, we discussed of major implementation decisions and proposed efficient ways to implement AWinA*, IBS$^+$ and IBS. Finally, we proposed three heuristic functions for the TD-TSPTW$_m$ which (i) are used to guide search and to prune the search space and (ii) offer different trade-offs between computational cost and tightness.

In Chapter 6, we experimentally (i) tune the parameters of ACS and AWA*, (ii) evaluate different implementation choices for IBS and IBS$^+$, and (iii) compare these four EAS algorithms on the TD-TSPTW$_m$. Before that, we present in the next chapter how we combine EAS algorithms with (i) TW constraint propagation, in order to reduce the size of the search space and to compute tighter bounds, and (ii) a local search procedure which tries to improve each solution found.

Chapter

5 Combining EAS with TW constraint propagation and local search

Contents

In Chapter 4, we have instantiated four EAS algorithms to solve the TD-TSPTW$_m$ and described three lower bounding functions. In this chapter, we propose to combine these EAS algorithms with both problem-specific and problem-independent techniques for the purpose of improving convergence and resolution speeds.

Throughout this chapter, we describe each of these components and how they interact with each other. In Section 5.1 we briefly present these components and how they are organized around an EAS algorithm. Then, in Section 5.2, we discuss how we use TW constraint propagation rules to (i) reduce the size of the state space and (ii) compute tighter bounds. Finally, we describe (i) a Local Search procedure used to improve the solutions found in Section 5.3 and (ii) a greedy search procedure used to try to quickly find a first feasible solution in Section 5.4.

5.1 Overview of the proposed approach

We present a high-level view of our solving approach in Figure 5.1: resolution starts by propagating TW constraints. We then use a greedy procedure to try to quickly find a first feasible solution. After that, an EAS algorithm is used to find a new best-known solution, or to prove the optimality of the current best-known solution. Whenever a new best-known solution is found (either by the EAS algorithm or by the greedy search procedure), a local search procedure is used to try to improve it, and TW constraints are propagated again before resuming search.

This process stops once the best-known solution is proven to be optimal. This proof may either be obtained by the EAS algorithm or by TW constraint propagation rules. If no feasible solution has been found when optimality is proven, then the problem instance has no feasible solution. Because we output each new best-known solution found, resolution may be stopped before termination, *e.g.,* once a solution of sufficient quality has been found, or when space and time resources are exhausted.

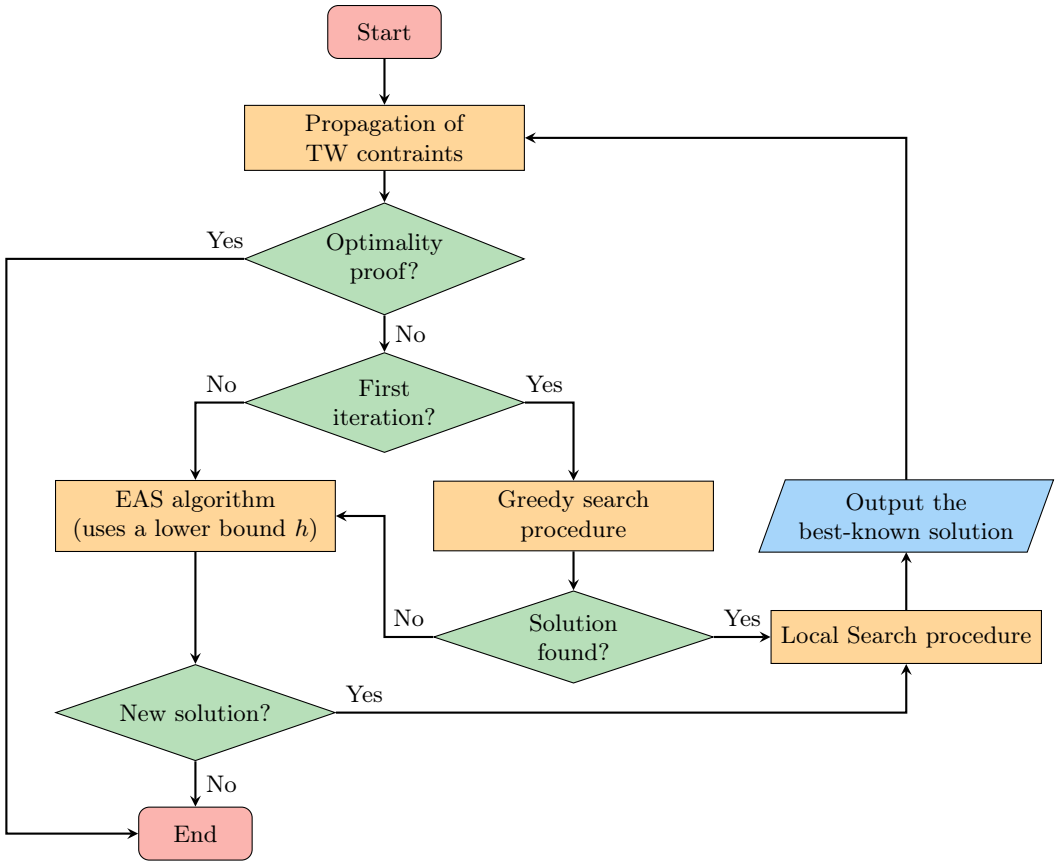


Figure 5.1: Overview of the proposed TD-TSPTW$_m$ solving approach.

5.2 Time Window constraint propagation

Similarly to most (TD-)TSPTW solving approaches, we use the classic TW constraint propagation rules presented in Section 1.3 before starting resolution (*e.g.,* before Line 3 of Algorithm 4.2). Recall that these rules allow one to obtain a tighter but equivalent formulation of the problem by (i) increasing release times e_i and decreasing deadlines l_i of vertices $i \in \mathcal{V}$, (ii) removing arcs from the set of usable arcs $\mathcal{E}$ and (iii) adding arcs to the set of precedence constraints $\mathcal{R}$.

In this section, we first describe how and why we also use these rules during resolution. We then discuss how one may adapt these rules to handle problem instances for which triangle inequality does not hold.

5.2.1 Propagation of constraints during resolution

To the best of our knowledge, existing solving approaches (such as those of [Vu+20], [LMS22] and [Pra23]) propagate TW constraints only at the beginning of resolution. We propose to also propagate them each time a new solution is found. More precisely, each time a new solution s is found, we update the TW of the destination vertex n by decreasing its deadline l_n to the makespan of s. We then propagate this TW tightening: this operation may refine TWs of other vertices, remove arcs from $\mathcal{E}$, and add new arcs to $\mathcal{R}$.

This strategy is specific to the makespan objective because it produces a tightened problem formulation which admits only a subset of feasible solutions with respect to the original instance (*i.e.,* solutions with a makespan strictly greater than the best-known are considered as infeasible).

Propagating TW constraints during resolution has multiple advantages:

- It may prove the optimality of the current best-known solution by tightening the TW of the destination vertex n to a single value $e_n = l_n$.
- It reduces the size of the state space: recall from Section 4.1 that the feasibility of action j in a given state s depends, amongst others, on l_j, $\mathcal{E}$ and $\mathcal{R}$.
- It allows one to compute tighter lower bounds h: as we have seen in Section 4.4, the tightness of the lower bounds we use depends (i) on sets $\mathcal{E}$ and $\mathcal{R}$, and (ii) on lower bounds on the travel time of arcs $(i, j) \in \mathcal{E}$, which are more accurate when TWs are tightened.

Note that the h-value – and consequently the f-value – of *open* states may be increased after this process. Because it would be computationally expensive to (i) compute the new h-value of each *open* state and (ii) to reorder the *open* set accordingly, we do not update the h-value of existing states, but use the tighter formulation for each newly generated state.

5.2.2 Adaptations of rules in absence of triangle inequality

When triangle inequality does not hold, two of the rules presented in Section 1.3 can be adapted in order to avoid wrongly ruling out any solution that is feasible in the original problem formulation:

- Rule $\mathbf{PFR}_1$: given distinct vertices $i, j \in \mathcal{V}$, condition $a_{i,j}(e_i) > l_j$ is necessary but not sufficient to infer that j precedes i. Indeed, it may be possible to reach j on time by visiting an intermediate vertex k within its TW, *i.e.,* there may exist a vertex $k \in \mathcal{V} \setminus \{i, j\}$ such that $\langle i, k, j \rangle$ is feasible. Therefore, we conclude that $(j, i) \in \mathcal{R}$ if j can only be reached later than l_j when departing from i at time e_i while visiting an arbitrary number of intermediate vertices within their TW (this is a constrained fastest path problem that can be solved in polynomial time using an adaptation of Dijkstra's algorithm). If such a path exists, we say that $\langle i, *, j \rangle$ is feasible.

- A similar argument applies to rule $\mathbf{PFR}_2$: given distinct vertices $i, j, k \in \mathcal{V}$, infeasibility of subpaths $\langle i, j, k \rangle$ and $\langle k, i, j \rangle$ is necessary but not sufficient to remove arc (i, j) from $\mathcal{E}$. Subpath $\langle i, j, k \rangle$ considers traveling to k immediately after using arc (i, j) whereas a fastest path from vertex j to k should be considered, *i.e.,* both $\langle i, j, *, k \rangle$ and $\langle k, *, i, j \rangle$ should be infeasible to remove arc (i, j) from $\mathcal{E}$. Additionally, if $\langle i, *, k, *, j \rangle$ is also infeasible, then vertex j must precede vertex i in any feasible solution.

Note that early works on the TSPTW (*e.g.,* [Dum+95]) broadly defined TW constraint propagation rules in terms of constrained fastest paths, only mentioning a simpler definition for the special case in which triangle inequality is satisfied.

When triangle inequality is not satisfied, these adapted rules may be used to guarantee that the transformed instance admits the exact same set of feasible solutions as the original one. They however come at the expense of an increased time complexity due to the computation of constrained TD fastest paths, although these rules are typically used only in instance preprocessing. One may also argue that TD travel time functions are inherently imperfect, and disregard this technical issue (a choice made by, *e.g.,* [Pra23]). We chose to use these adapted rules (i) because we also use them each time a new solution is found, and (ii) in order to ease the analysis of experimental results: indeed, for some instances considered for experimental evaluation in the next chapters, using the rules which assume that triangle inequality holds prevents one from finding a solution with optimal makespan (*i.e.,* optimal solutions are wrongly considered as infeasible). Using these adapted rules therefore allows us to obtain the same optimal makespan across the compared approaches, including the ones which do not propagate TW constraints.

5.3 Local Search

In order to converge faster towards good solutions, we use a greedy Local Search (LS) procedure to try to improve each solution provided by the EAS algorithm. If this procedure manages to improve the input solution, then l_n is set to the new best-known makespan and TW constraints are propagated (see Subsection 5.2.1). The LS procedure we use is similar to the one proposed by [DU10] for the TSPTW$_{\Sigma t}$: consequently, we only outline it for self-containedness.

This LS procedure is based on the *1-shift* and *2-opt* neighborhoods, which both have a size in $O(|\mathcal{V}|^2)$. The *1-shift* neighborhood consists in moving a single vertex backward or forward in the path and the *2-opt* neighborhood consists in reversing a subsequence of vertices of the path.

Sets $\mathcal{E}$ and $\mathcal{R}$ are used to reduce the size of the neighborhoods by discarding moves proven to lead to infeasible solutions. When evaluating a move, we update in $O(|\mathcal{V}|)$ time the visit time at each vertex impacted by it, and discard the move if it leads to constraint violations. Note that discarding moves using sets $\mathcal{E}$ and $\mathcal{R}$ also allows us to discard moves which are proven to lead to a solution worse than the best-known, given these sets are tightened when we propagate TW constraints during search according to the best-known makespan (see Subsection 5.2.1).

The actual LS procedure uses the *First Improvement* strategy, *i.e.,* each move leading to an improved solution is accepted. This procedure is based on a variant of *Variable Neighborhood Search* (VNS, [MH97]) called *Variable Neighborhood Descent* (VND, [HMM08]): neighborhoods *1-shift* and *2-opt* are considered alternatively until the current best-known solution is locally optimal with respect to both neighborhoods (*i.e.,* when this solution cannot be improved using a single move).

Note that in the original approach of [DU10], moves are evaluated more efficiently because this approach has been designed to handle a special case of the TSPTW$_{\Sigma t}$. More precisely, authors efficiently evaluate any given move by first computing the cost of the resulting solution in constant time. Then, the linear-time feasibility check is only performed if this cost is lower than the one of the current solution. Two conditions are necessary to be able to compute this cost in constant time: travel times must be symmetric and constant, and waiting times must not be included in the objective function. None of these conditions hold when considering the TD-TSPTW$_m$, *i.e.,* linear time is required both to (i) compute the makespan of a neighboring solution and (ii) check its feasibility.

We chose this local search procedure for its simplicity and for its relatively low time requirements: indeed, the execution time dedicated to this procedure may help to improve the quality of the solutions provided but it does not participate to the exploration of the state transition graph and thus does not directly contribute to prove the optimality of the current best-known solution. However, because this procedure helps to tighten the upper bound, it may allow the EAS algorithm to prune more states and thus make resolution faster. Additionally, this procedure helps to reduce the overall number of improvements of the upper bound ub, and thus the number of times TW constraint propagation rules are used during resolution.

5.4 Greedy computation of an initial solution

Before running the EAS algorithm, we use a greedy search procedure which tries to find an initial solution in $O\left(|\mathcal{V}|^2\right)$ time. This procedure consists in looking for a path from the initial state $(0, \{0\}, e_0)$ to a final state in the state transition graph G^P_{ST}. We iteratively build this path starting from the initial state. In a given state $s = (i, \mathcal{S}, t)$, we greedily move to one of its successors $s' = (i', \mathcal{S}', t')$ such that $(s, s') \in \mathcal{A}$: we choose the state s' which minimizes $(l_{i'}, e_{i'}, t')$, in lexicographical order. In other words, we select the non-visited vertex i' with the earliest deadline $l_{i'}$, break ties in favor of lower earliest visit time $e_{i'}$ and break further ties in favor of earlier arrival times on i'. Recall from Section 4.1 that the set of arcs $\mathcal{A}$ of the state transition graph recognizes that (i) some arcs cannot be used in a feasible solution and (ii) precedence relations may exist between vertices.

Note that computing such an upper bound before starting search is not a new idea: for example, this strategy is used in Enhanced A* [II99] (see Subsection 3.3.2) to add pruning capabilities to A*. In theory – provided this procedure finds a feasible solution – states can be pruned earlier because (i) the upper bound l_n is lower, and (ii) propagating TW constraints with this tighter upper bound leads to computing tighter lower bounds h. While this strategy is beneficial to A* (given A* is not anytime), results did not show significant performance improvements when considering EAS algorithms designed to find solutions quickly, *e.g.,* ACS with a small breadth-limit B. Indeed, when it is trivial to find a feasible solution, such algorithms are likely to find a solution of better quality as they are guided using additional heuristic information. When it is not trivial to find a feasible solution, this greedy procedure is unlikely to find one.

Finally, we shall see in Section 6.3 that using this procedure has the advantage of easing the analysis of experimental results when considering EAS algorithms which converge slowly.

5.5 Discussion

In this chapter, we have described how we combine EAS algorithms with various extra components in order to improve performance. More precisely, we proposed to combine EAS algorithms with (i) TW constraint propagation, both before and during resolution – in order to reduce the size of the search space and to compute tighter lower bounds – and (ii) an LS procedure which tries to improve the solutions found. Also, we have described a way to orchestrate these components discussed how they benefit each other.

In the next chapter, we experimentally (i) evaluate the performance impact of the main components of our approach, (ii) compare the four EAS algorithms we have considered in Chapter 4 and (iii) study the influence of several implementation choices. We then compare our solving approach to state-of-the-art TD-TSPTW$_m$ solvers in Chapter 7 and provide experimental results on constant benchmarks in Chapter 8.

Part

III

Experimental results

Chapter

6 Experimental comparison of different EAS algorithms

Contents

In this chapter, we evaluate experimentally the four EAS algorithms we instantiated for the TD-TSPTW$_m$ in Section 4.2. We start by describing the experimental setting in Section 6.1, which includes our performance criteria and the associated performance measures, the problem instances we consider and the hardware we used. In Section 6.2, we report results of preliminary experiments in order to validate the relevance of key components of our solving approach. We then evaluate the impact of (i) parameters of EAS algorithms in Section 6.3 and (ii) major implementation decisions in Section 6.4. Finally, in Section 6.5, we compare the best variants of each of the four EAS algorithms we consider.

6.1 Experimental setting

In this chapter, we consider the lower bound h_{OIA} because it provides a middle-ground between h_{FEA} and h_{MSA} in terms of computational cost. Also, as we shall see in Chapters 7 and 8, h_{OIA} tends to provide the best results amongst these three lower bounds. Moreover, preliminary experiments have shown us that results follow similar trends when considering either h_{FEA} or h_{MSA} instead of h_{OIA}.

Performance criteria. In this thesis, we experimentally compare solving approaches according to (i) their ability to quickly provide close-to-optimal solutions (*i.e.,* convergence speed), (ii) their ability to quickly prove that the current best-known solution is optimal (*i.e.,* solving speed), and (iii) the amount of memory they use.

Note that these criteria may be incompatible: we made the choice of preferring convergence speed over solving speed because TD travel times are inherently imperfect (see Section 1.2), *i.e.,* optimality proofs are mainly useful to determine when to stop searching. Also, we seek to obtain a scalable solving approach: when considering hard enough instances, optimality proofs are out of reach and it is preferable to provide the best possible approximate solution instead.

Performance measures. We say that an instance is solved by an approach whenever it finds an optimal solution and proves its optimality within one hour. We note $\#s$ the number of solved instances and t_s the average time required to solve them. Similarly, we note $\#r$ the number of instances for which an approach has found a reference solution, and t_r the average time needed to find them. A reference solution is either an optimal solution or a solution of best-known makespan. The reference solutions of all instances considered in this chapter are known to be optimal.

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) amongst all approaches which maximize $\#s$ (resp. $\#r$).

Also, to evaluate the ability of an approach to quickly converge towards close-to-optimal solutions, we plot the evolution of the gap to the reference solution (in percentage) with respect to time: given the makespan m of a solution and a reference makespan m_r, we define the gap of m with respect to m_r as $gap(m, m_r) = \frac{m-m_r}{m_r}$. When considering results over multiple instances, we display the evolution of the average gap with respect to time.

Finally, we compare the space requirements of each approach by reporting the average peak memory use (in GiB) during resolution.

Problem instances. Throughout this chapter, we consider a subset of instances from benchmark B_{ARI19} (see Section 1.5): although imperfect in terms of TD travel times, this benchmark contains challenging instances and is widely used to evaluate the performance of TD-TSPTW$_m$ solvers.

Recall that in this benchmark, TW tightness is determined by parameter $\beta \in \{0, 0.25, 0.50, 1\}$: TWs are widest when β is close to 0 and tightest when β is close to 1. Because we are interested in evaluating both the convergence speed and the ability to prove the optimality of the solutions found, we consider instances of intermediate difficulty, *i.e.,* instances with $n = 31$ and $\beta \leq 0.50$ (we exclude instances with $\beta = 1$ because it is trivial for our approach to solve them). Despite their relatively small size n, instances from this subset are relatively challenging to solve due to the wideness of their TWs (see the sample TW distributions in Figure 1.5, page 26).

Given an instance size n, this benchmark contains 300 instances for each value of β. Two additional

parameters, $\Delta \in \{.70, .80, .90, .95, .98\}$ and $P \in \{B_1, B_2\}$ determine the variations in TD travel time functions. We shall see in Section 7.2 that our approach is relatively insensitive to these parameters, *i.e.,* its performance mainly depends on the instance size n and on the TW tightness β. Consequently, we only consider a smaller yet representative set of instances for each value of β. We obtain this representative subset by randomly sampling six out of the thirty instances provided for each of the ten combinations of values for parameters Δ and P. This leads us to considering 60 instances per value of β (*i.e.,* 20% of them): because we consider instances with $n = 31$ and $\beta \leq 0.50$, our subset of B_{ARI19} is composed of 180 instances overall.

Hardware. Unless otherwise specified, experiments presented in this thesis were executed on 2.1 gigahertz Intel Xeon E5-2620 v4 processors with 64 GiB of RAM. To favor reproducibility, experiments were carried out using the Grid'5000 testbed, which is supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`). Additionally, as suggested by [Fic+21], *Turbo Boost* was disabled and each machine solved one instance at a time, using a single processor.

6.2 Preliminary experiments

Before comparing the performance of various EAS algorithms, we first study the performance impact of the main components on our approach. Our approach is composed of three key components, *i.e.,* the propagation of TW constraints (see Section 5.2), Local Search (LS, see Section 5.3), and rules that exploit LDTs to filter the set of usable edges $\mathcal{E}_s$ when computing lower bounds h (see Subsection 4.4.2).

We evaluate the relevance of these components by reporting results of Anytime Column Search (ACS, see Subsection 4.2.3) with breadth-limit $B = 1$ (we study the impact of this parameter in Section 6.3). We first report results by adopting an additive method: starting from an approach in which all components are disabled, we enable each component one by one until obtaining the complete solving approach in which all components are enabled. We then adopt a complementary subtractive method by separately disabling each component to study their individual impact.

Additive analysis. We first evaluate the relevance of these components by reporting results obtained with the following variants of ACS:

- ACS_0 is the variant in which the three components are disabled;
- ACS_1 is obtained from ACS_0 by enabling TW constraint propagation before starting search (*i.e.,* in preprocessing);
- ACS_2 is obtained from ACS_1 by also enabling TW constraint propagation during search (*i.e.,* each time the upper bound ub is decreased);
- ACS_3 is obtained from ACS_2 by enabling LS;

- ACS is obtained from ACS_3 by enabling the filtering of the usable edge set $\mathcal{E}_s$ based on LDTs.

In Table 6.1, we display performance measures of these variants on the representative subset of B_{ARI19} described in Section 6.1 (similar results have been obtained on other benchmarks). When looking at the number of solved instances on the left-hand side of Table 6.1, we see that all components except LS improve performance: ACS_1, which propagates TW constraints before starting search, solves 64 more instances than ACS_0; ACS_2, which also propagates TW constraints during search, solves four more instances. Finally, ACS also filters set $\mathcal{E}_s$ and solves 67 additional instances.

When looking at the number of reference solutions found on the right-hand side of Table 6.1, we see that propagating TW constraints during search slightly degrades performance (ACS_2 finds two less reference solution than ACS_1). We also see that LS allows ACS_3 to find reference solutions faster. To compare the ability of our different variants to quickly converge towards quality solutions, we display in Figure 6.1 the evolution of the gap to the reference solution with respect to time. It demonstrates that LS allows ACS_3 to find better solutions than ACS_2 at the beginning of search, especially when TWs are wide.

	Solved instances										Reference solutions									
	ACS_0		ACS_1		ACS_2		ACS_3		ACS		ACS_0		ACS_1		ACS_2		ACS_3		ACS	
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	-	0	-	0	-	60	825	35	900	57	522	55	486	55	456	60	23
0.25	0	-	49	1614	53	1567	53	1567	60	249	60	83	60	11	60	13	60	12	60	3
0.50	45	1309	60	30	60	22	60	22	60	7	60	2	60	1	60	1	60	1	60	1
Total	45		109		113		113		180		155		177		175		175		180	

Table 6.1: Performance of ACS_0, ACS_1, ACS_2, ACS_3, and ACS on a representative subset of B_{ARI19} 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 find the reference solution (t_r).

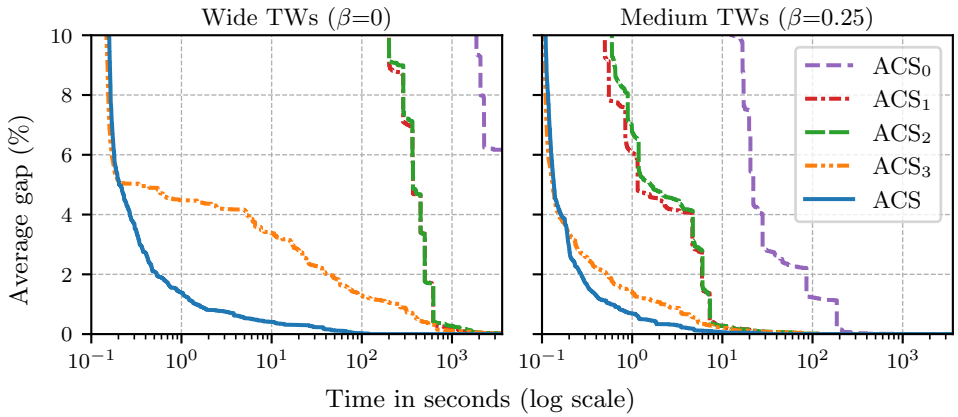


Figure 6.1: Evolution of the average gap to the optimal solution for ACS_0, ACS_1, ACS_2, ACS_3, and ACS on a representative subset of B_{ARI19} with $n = 31$ and $\beta \in \{0, 0.25\}$ (60 instances per value of β).

Finally, let us mention that all components except LS significantly reduce memory use. For example, when $\beta = .25$, ACS_0 (resp. ACS_1, ACS_2, ACS_3, and ACS) used on average 63.8 (resp. 23.1, 19.9, 19.9, and 2.1) GiB of memory.

Subtractive analysis. We now consider the relevance of each component by studying three variants of ACS in which we disable a single component at a time. Contrarily to the previous analysis, results are independent of the order in which components are enabled. We note $\text{ACS}_{-\text{FILT}}$ the variant of ACS in which the filtering of edge set $\mathcal{E}_s$ is disabled when computing lower bounds h. Similarly, we note $\text{ACS}_{-\text{CPD}}$ (resp. $\text{ACS}_{-\text{LS}}$) the variant of ACS in which TW constraint propagation during search (resp. local search) is disabled.

Note that we do not present results for the variant in which TW constraint propagation is completely disabled (*i.e.,* both before and during search), given it is a prerequisite to the other components (note that this variant corresponds to ACS_0). Also notice that $\text{ACS}_{-\text{FILT}}$ is equivalent to ACS_3: consequently, we do not discuss the results of this variant but display them anyway to allow comparison.

The left-hand side of Table 6.2 shows us that LS has no impact on solving performance. Propagating TW constraints during search is however beneficial as ACS solves two more instances than $\text{ACS}_{-\text{CPD}}$.

On the right-hand side of Table 6.2 we see that reference solutions are found faster when enabling LS for instances with $\beta \leq 0.25$ and, to a lesser extent, when propagating TW constraints during search for instances with $\beta = 0$. In Figure 6.2 (notice the shorter y-axis scale), we see that LS improves convergence speed early during search and that TW constraint propagation during search slightly degrades it, *e.g.,* for time limits shorter than one second when $\beta = 0$.

Each component except LS leads to a decrease in the average peak memory use, *e.g.,* when $\beta = 0.25$, $\text{ACS}_{-\text{FILT}}$ (resp. $\text{ACS}_{-\text{CPD}}$, $\text{ACS}_{-\text{FILT}}$ and ACS) used 19.9 (resp. 2.6, 2.1 and 2.1) GiB of memory.

	Solved instances								Reference solutions							
	$\text{ACS}_{-\text{FILT}}$		$\text{ACS}_{-\text{CPD}}$		$\text{ACS}_{-\text{LS}}$		ACS		$\text{ACS}_{-\text{FILT}}$		$\text{ACS}_{-\text{CPD}}$		$\text{ACS}_{-\text{LS}}$		ACS	
β	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r
0	0	-	58	826	60	825	60	825	55	456	60	25	60	27	60	23
0.25	53	1567	60	289	60	249	60	249	60	12	60	3	60	4	60	3
0.50	60	22	60	10	60	7	60	7	60	1	60	0	60	1	60	1
Total	113		178		180		180		175		180		180		180	

Table 6.2: Performance of $\text{ACS}_{-\text{FILT}}$, $\text{ACS}_{-\text{CPD}}$, $\text{ACS}_{-\text{FILT}}$ and ACS on a representative subset of B_{Ari19} 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 find the reference solution (t_r).

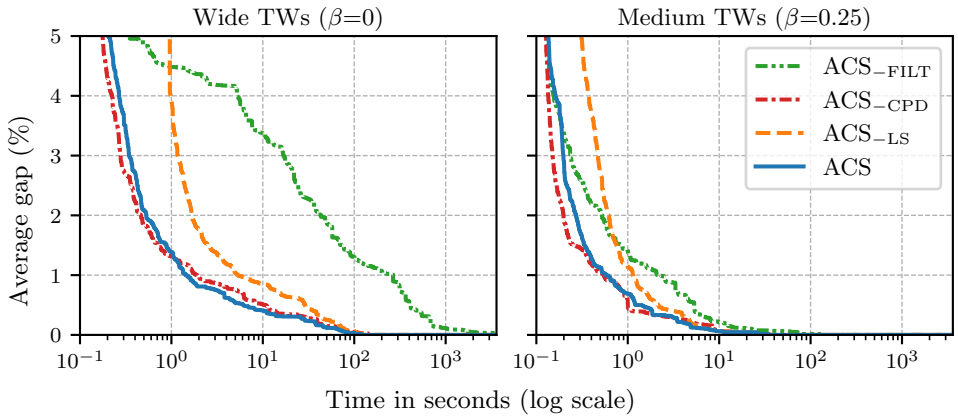


Figure 6.2: Evolution of the average gap to the optimal solution for ACS$_{-\text{FILT}}$, ACS$_{-\text{CPD}}$, ACS$_{-\text{FILT}}$ and ACS on a representative subset of B_{ARI19} with $n = 31$ and $\beta \in \{0, 0.25\}$ (60 instances per value of β).

Discussion. In this section, we have studied the impact of each key component of our approach. Results show that filtering set $\mathcal{E}_s$ based on LDTs when computing lower bounds is useful both for solving and convergence speed and also leads to a lower memory use. Also, LS has been shown to be beneficial for convergence speed at the beginning of resolution when TWs are wide, without having a significant impact on solving speed or memory use.

TW constraint propagation during search also turns out to be slightly beneficial on our three performance measures. While results of Table 6.1 show that it decreases convergence speed, we see in Table 6.2 that it leads to faster convergence on hard instances. These different conclusions may be explained by the fact that (i) when combined with LS, this procedure is called less often, and its time overhead is reduced, and (ii) when combined with the filtering of set $\mathcal{E}_s$, our lower bounds are tighter and benefit more from tightening TWs during search.

6.3 Parameter tuning

In this section, we seek to determine which parametrization of ACS and AWA* perform best. Note that the parameter of both algorithms has an influence on convergence speed. Also, recall from Section 5.4 that we try to greedily compute an initial solution and try to improve it through local search before running the EAS algorithm. These procedures are deterministic, so – on a given problem instance – all configurations of ACS and AWA* start with the same initial solution. Because these procedures run quickly, we are able to plot the evolution of the average gap with respect to time as early as possible (this average gap is undefined until at least one feasible solution has been found for each instance): this is useful when considering parametrizations of these algorithms which lead to slow convergence speed.

ACS's breadth-limit B. We now study the influence of ACS's parameter B, which determines the maximum number of states expanded in each layer of the state transition graph per iteration. We consider five values for parameter B based on a geometric progression, *i.e.*, $B \in \{1, 10, 100, 1\text{k}, 10\text{k}\}$ (for readability, we use the suffix "k" to denote multiplication by 1000). We note ACS^B the configuration of ACS with a given breadth-limit B.

The left-hand side of Table 6.3 shows us that greater values of B tend to lead to better solving performance, *e.g.*, when $\beta = 0$ or when $\beta = 0.25$, ACS^1 is – on average – 6% slower than $\text{ACS}^{10\text{k}}$. Also, the impact of increasing B by a factor of 10 is larger when B is small (*e.g.*, when $\beta = 0$, ACS^0 is 2.5% slower than ACS^{10} but ACS^{100} is only 1% slower than $\text{ACS}^{1\text{k}}$). The right-hand side of Table 6.3 shows no significant difference in terms of convergence speed.

However, when looking at the evolution of the average gap to the optimal solution in Figure 6.3, we remark that large values of B tend to harm convergence speed when considering short time limits. Also, when TWs are wide, ACS^1 converges slightly faster than the other variants for time limits lower than 400s.

Finally, note that parameter B has no significant influence on memory use.

We may explain these trends by the fact that larger values of B steers ACS to a more *breadth-first*

	Solved instances										Reference solutions									
	ACS^1		ACS^{10}		ACS^{100}		$\text{ACS}^{1\text{k}}$		$\text{ACS}^{10\text{k}}$		ACS^1		ACS^{10}		ACS^{100}		$\text{ACS}^{1\text{k}}$		$\text{ACS}^{10\text{k}}$	
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	60	825	60	804	60	785	60	777	60	776	60	23	60	23	60	23	60	23	60	23
0.25	60	249	60	244	60	239	60	236	60	234	60	3	60	3	60	3	60	3	60	4
0.50	60	7	60	6	60	6	60	6	60	6	60	1	60	0	60	0	60	1	60	1
Total	180		180		180		180		180		180		180		180		180		180	

Table 6.3: Performance of ACS with different breadth-limits B on a representative subset of B_{ARI19} 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 find the reference solution (t_r).

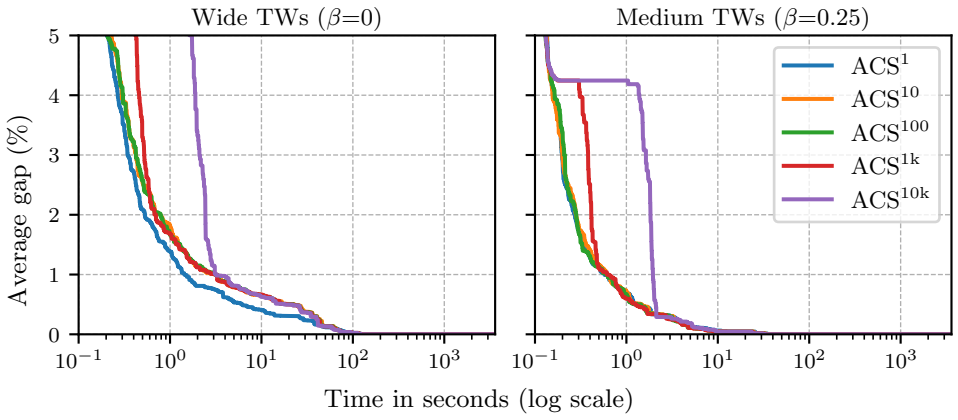


Figure 6.3: Evolution of the average gap to the optimal solution for ACS with different breadth-limits B on a representative subset of B_{ARI19} instances with $n = 31$ and $\beta \in \{0, 0.25\}$ (60 instances per value of β).

behavior, while values close to 1 lead to a *depth-first* behavior. When B tends to infinity, ACS degenerates into the Breadth-First algorithm presented in Section 4.1: because this algorithm only expands nodes with optimal g-values, it is not anytime but it minimizes node re-expansions. On the other hand, lower values of B allow to report solutions more frequently, given (i) solutions may only be found at the end of an iteration, and (ii) each iteration expands $O(B \cdot |\mathcal{V}|)$ nodes.

While larger values of B tend to lead to fewer node re-expansions, the solving speed does not steadily decrease because it delays decreases of the upper bound ub and thus opportunities of pruning using the heuristic function. Additionally, ACS relies on min-heaps to maintain the *open* sets, which are optimized for efficiently obtaining a single most promising state. In Section 6.4, we consider an alternative implementation more suited to larger (and increasing) values of B.

Because we are interested in scalability and convergence speed, we consider the variant of ACS with $B = 1$ (recall from Subsection 3.3.3 that ACS is a generalization of CBFS in which $B = 1$). Indeed, large values of B that perform well on this set of instances may lead to prohibitively long iteration times and thus poor convergence speed when (i) considering larger graphs or (ii) using lower bounds which are more expensive to compute than h_{OIA}.

AWA*'s heuristic weight w. We now consider Anytime Weighted A* (AWA*, see Subsection 4.2.1) and analyze the influence of the heuristic weight w on its performance. Recall that AWA* degenerates into A* when $w = 1$, and that it is guided by the heuristic function only when w tends to infinity. We study the performance of AWA* with five heuristic weights $w \in \{1.5, 2, 2.5, 3.5, 5\}$ and note AWA*^{w} the configuration of AWA* with weight w.

The left-hand side of Table 6.4 shows us that using a low heuristic weight is preferable for solving performance. Regarding convergence speed, the right-hand side of Table 6.4 shows us that extreme values of w (*e.g.,* 1.5 and 5) lead to poor performance, whereas the intermediate value of 2.5 allows AWA* to find all reference solutions in minimal average time.

Figure 6.4 presents the evolution of the average gap to the optimal solution for each TW width β: notice that AWA*$^{3.5}$ outperforms both AWA*^{2} and AWA*$^{2.5}$ when TWs are wide (*i.e.,* $\beta = 0$), but the opposite is true when considering tighter TWs (*i.e.,* $\beta = 0.50$)

Also, memory use increases when the heuristic weight w increases, *e.g.,* when $\beta = 0.25$, AWA* with weight $w = 1.5$ (resp. 2, 2.5, 3.5, 5) used 7.4 (resp. 7.8, 8.5, 14.4 and 18.7) GiB of memory.

These results show us that optimizing the convergence speed of AWA* requires adapting the weight w according to the TW width or, more generally, to the size of the graph. Indeed, excessively large heuristic weights lead to mainly expanding states located close to the goal (*i.e.,* states with low h-value) without trying to find better paths leading to these states (*i.e.,* failing to find short paths leading to states located close to the initial state). On the other hand, low heuristic weights fail to provide close-to-optimal solutions on hard instances because most of the time is spent expanding states located close to the initial state (*i.e.,* states with low f-value, like A*).

When comparing various EAS algorithms in Section 6.5, we report results for AWA*$^{2.5}$ because it

	Solved instances										Reference solutions									
	AWA*$^{1.5}$		AWA*^{2}		AWA*$^{2.5}$		AWA*$^{3.5}$		AWA*^{5}		AWA*$^{1.5}$		AWA*^{2}		AWA*$^{2.5}$		AWA*$^{3.5}$		AWA*^{5}	
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	57	769	57	767	56	810	51	816	46	1154	58	573	60	271	60	94	60	150	58	597
0.25	60	263	60	282	60	340	60	615	58	761	60	116	60	17	60	14	60	251	59	473
0.50	60	7	60	8	60	9	60	15	60	22	60	3	60	1	60	1	60	6	60	14
Total	177		177		176		171		164		178		180		180		180		177	

Table 6.4: Performance of AWA* with different heuristic weights w on a representative subset of B_{ARI19} 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 find the reference solution (t_r).

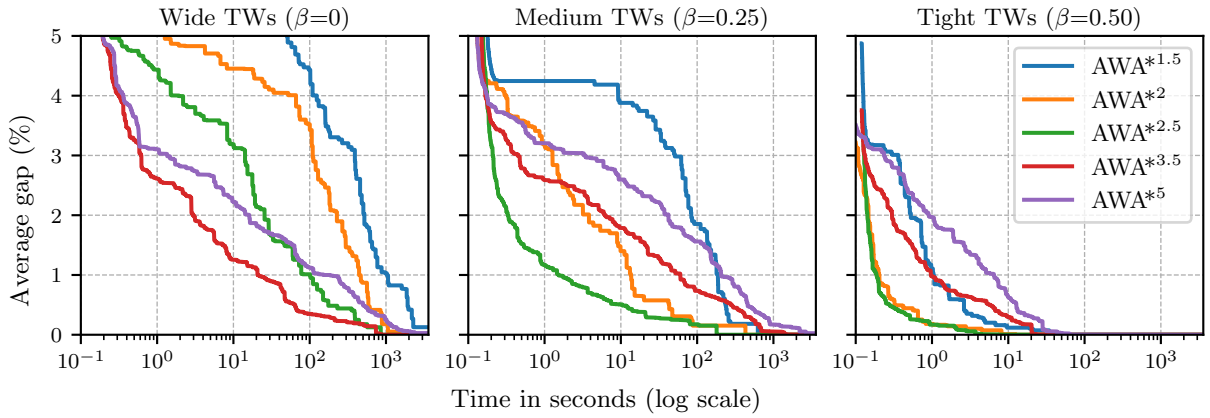


Figure 6.4: Evolution of the average gap to the optimal solution for AWA* with different heuristic weights w on a representative subset of B_{ARI19} with $n = 31$ and $\beta \in \{0, 0.25, 0.50\}$ (60 instances per value of β).

performs best for quickly finding reference solutions.

6.4 Validation of implementation choices

In this section, we compare variants of Iterative Beam Search (IBS, see Subsection 4.2.2) which differ in (i) the amount of nodes they re-expand and (ii) the data structures they use. This comparison shall also allow us to highlight the importance of avoiding node re-expansions, *i.e.*, of avoiding to perform redundant work. One of these variants was originally applied to the SOP in [Lib+20]: we show two ways to improve on this approach. These improvements are likely to improve on this original's approach performance in the context of the SOP, given the state transition graph of this problem has the same topology as the one of the TD-TSPTW$_m$.

Recall that IBS consists in performing a series of breadth-limited breadth-first searches: only the B most promising nodes are considered at each layer of the graph, and the remaining nodes are inadmissibly pruned. The breadth-limit B is increased geometrically after each search iteration and IBS proves optimality when a search iteration terminates without having inadmissibly pruned any node. We consider two implementations of IBS (we describe them more in depth in

Subsection 4.3.2):

- IBS_{heap} is the algorithm used to tackle the SOP in [Lib+20]. With this implementation, set *next* has a bounded size (*i.e.,* it contains at most B states) but it may contain duplicate elements: therefore, this implementation may needlessly re-expand states.
- $\text{IBS}_{\text{select}}$ improves on the implementation of IBS_{heap}, *i.e.,* elements of set *next* are unique and a selection algorithm is used to determine which states are most promising.

Additionally, recall that in Subsection 4.2.3, we have proposed an improved variant of IBS called IBS^+: IBS^+ improves on IBS by avoiding to restart search after each iteration. This is achieved by memorizing states into *open* sets instead of inadmissibly pruning them, which tends to reduce the number of re-expansions. In Subsection 4.3.2, we proposed to implement IBS^+ using a selection algorithm, as it is theoretically more efficient than implementing it using min-heaps: we experimentally validate this choice by comparing these two variants, which we respectively note $\text{IBS}^+_{\text{select}}$ and $\text{IBS}^+_{\text{heap}}$ (note that $\text{IBS}^+_{\text{heap}}$ is very similar to ACS, the only difference between these algorithms being the geometrical growth of B).

The left-hand side of Table 6.5 shows us that $\text{IBS}_{\text{select}}$ solves 11 more instances than IBS_{heap}, and that $\text{IBS}^+_{\text{select}}$ in turns solves 10 more instances than $\text{IBS}_{\text{select}}$. When $\beta = 0.50$, all instances are solved by these three variants of IBS: on average, $\text{IBS}_{\text{select}}$ solves these instances six times as fast as IBS_{heap}, and $\text{IBS}^+_{\text{select}}$ solves them 18 times as fast as IBS_{heap}. Also notice that $\text{IBS}^+_{\text{select}}$ and $\text{IBS}^+_{\text{heap}}$ solve the same number of instances overall: when $\beta = 0$, $\text{IBS}^+_{\text{select}}$ is – on average – 14% as fast as $\text{IBS}^+_{\text{heap}}$.

On the left-hand side of Table 6.5, we see that the four algorithms we consider find all reference solutions. When $\beta = 0$, $\text{IBS}_{\text{select}}$ and $\text{IBS}^+_{\text{select}}$ are respectively 1.5 and 2.8 times as fast, on average, as IBS_{heap}. We also see that $\text{IBS}^+_{\text{select}}$ and $\text{IBS}^+_{\text{heap}}$ have comparable performance and we draw similar conclusions when looking at the evolution of the average gap to the optimal solution in Figure 6.5.

When $\beta = 0.25$, IBS_{heap} (resp. $\text{IBS}_{\text{select}}$, $\text{IBS}^+_{\text{select}}$ and $\text{IBS}^+_{\text{heap}}$) used 2.3 (resp. 2.5, 2.2 and 2.8) GiB of memory. We may explain these trends by the fact that all of these implementations memorize the value associated to each *known* state in a map: $\text{IBS}^+_{\text{select}}$ tends to use less memory because, contrarily to IBS_{heap} and $\text{IBS}_{\text{select}}$, it does not maintain an additional set of states *next*. Similarly, $\text{IBS}^+_{\text{heap}}$ memorizes *open* states both in a hash map and in min-heaps (also recall from Subsection 4.3.1 that these heaps may contain obsolete states due to lazy heap operations).

Discussion. Results show that our improved version of IBS, IBS$^+$, outperforms IBS$_{\text{heap}}$, the algorithm applied to the SOP in [Lib+20]. The difference in performance is particularly important when considering solving performance. Indeed, IBS$^+$ is more efficient at avoiding node re-expansions than IBS$_{\text{heap}}$ and IBS$_{\text{select}}$: contrarily to these approaches, IBS$^+_{\text{select}}$ does not restart search to ensure completeness but maintains *open* sets instead. Such a strategy avoids overheads associated to re-expansions, which include redundant computations of the heuristic function h.

We have also compared two implementations of IBS$^+$ (*i.e.,* IBS$^+_{\text{select}}$ and IBS$^+_{\text{heap}}$), which differ only in the data structures used to implement the *open* set: as expected, results show that when the breadth-limit grows exponentially, using a selection algorithm is more efficient to solve instances than using min-heaps.

	Solved instances								Reference solutions							
	IBS$_{\text{heap}}$		IBS$_{\text{select}}$		IBS$^+_{\text{select}}$		IBS$^+_{\text{heap}}$		IBS$_{\text{heap}}$		IBS$_{\text{select}}$		IBS$^+_{\text{select}}$		IBS$^+_{\text{heap}}$	
β	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r
0	43	1495	50	1039	60	660	60	751	60	74	60	49	60	26	60	27
0.25	56	852	60	512	60	201	60	229	60	11	60	7	60	4	60	4
0.50	60	89	60	14	60	5	60	6	60	2	60	1	60	1	60	1
Total	159		170		180		180		180		180		180		180	

Table 6.5: Performance of different implementations of IBS on a representative subset of B_{ARI19} 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 find the reference solution (t_r).

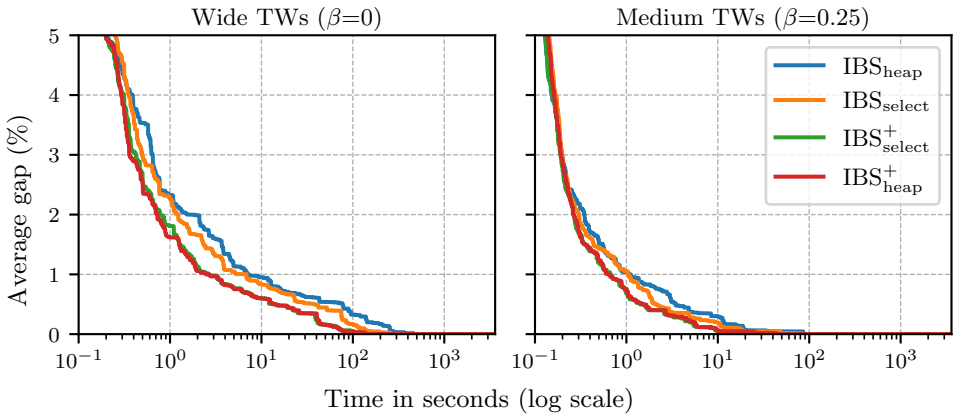


Figure 6.5: Evolution of the average gap to the optimal solution for different implementations of IBS on a representative subset of B_{ARI19} with $n = 31$ and $\beta \in \{0, 0.25\}$ (60 instances per value of β).

6.5 Overall comparison

In this section, we compare the performance of ACS, IBS, AWA* and AWinA* in order to determine which of these EAS algorithms has the best abilities for fast convergence and scalability. We consider the best performing parametrizations of ACS and AWA* (*i.e.,* ACS^{1} and AWA*$^{2.5}$, see Section 6.3), the best performing implementation of IBS^{+} (*i.e.,* IBS$^{+}_{\text{select}}$, see Section 6.4), and Anytime Window A* (AWinA*, see Subsection 4.2.4).

The left-hand side of Table 6.6 shows us that ACS, IBS^{+} and AWinA* manage to solve all instances: IBS^{+} is the fastest, followed by AWinA* and ACS. AWA*, on the other hand, failed to solve four instances.

Regarding convergence speed, we see on the right-hand side of Table 6.6 that all algorithms have similar performance on instances with tight TWs (*i.e.,* when $\beta = 0.50$). When TWs are wider (*i.e.,* $\beta \in \{0, 0.25\}$), ACS outperforms the other algorithms. More specifically, when $\beta = 0$, ACS finds – on average – the reference solution 13% as fast as IBS^{+}, two times as fast as AWinA*, and four times as fast as AWA*. We observe similar trends when looking at the evolution of the average gap to the optimal solution in Figure 6.6.

	Solved instances								Reference solutions							
	ACS^{1}		IBS^{+}		AWinA*		AWA*$^{2.5}$		ACS^{1}		IBS^{+}		AWinA*		AWA*$^{2.5}$	
β	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r
0	60	825	60	660	60	756	56	810	60	23	60	26	60	45	60	94
0.25	60	249	60	201	60	239	60	340	60	3	60	4	60	8	60	14
0.50	60	7	60	5	60	6	60	9	60	1	60	1	60	0	60	1
Total	180		180		180		176		180		180		180		180	

Table 6.6: Performance of ACS, IBS^{+}, AWA* and AWinA* with h_{OIA} on a representative subset of B_{ARI19} 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 find the reference solution (t_r).

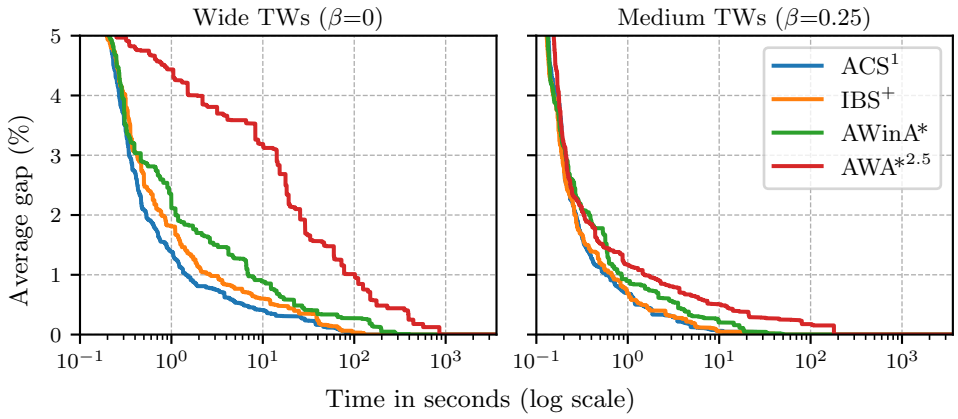


Figure 6.6: Evolution of the average gap to the optimal solution for ACS, IBS^{+}, AWA* and AWinA* with h_{OIA} on a representative subset of B_{ARI19} with $n = 31$ and $\beta \in \{0, 0.25\}$ (60 instances per value of β).

Finally, both ACS and IBS$^+$ tend to require less memory than AWinA* and AWA*, *e.g.,* when $\beta = 0.25$, ACS (resp. IBS, AWinA* and AWA*) used 2.1 (resp. 2.2, 2.4, 8.5) GiB of memory.

Discussion. Results show that breadth-constrained algorithms (*i.e.,* ACS and IBS$^+$) generally perform better on the TD-TSPTW$_m$ than AWinA* and AWA*, which are respectively based on depth-constraints and weighted heuristics (see the taxonomy established in Subsection 3.3.3). We believe this to stem from the fact that ACS and IBS$^+$ exploit the layered structure of the graph in order to balance depth-first and breadth-first behaviors. Additionally, authors of [Pea84] pointed out that systematic heuristic error is often homogeneous within a layer: these two algorithms benefit from this as they only compare states by f-value within a given layer, which leads to fairer comparisons.

The key difference between ACS and IBS$^+$ is that the breadth-limit remains constant in ACS, while in IBS$^+$, it grows exponentially at the end of each iteration. Consequently, the behavior of ACS remains mainly depth-first throughout search, whereas it progressively shifts from being purely depth-first to being mainly breadth-first in IBS$^+$: switching to this breadth-first behavior tends to favor optimality proofs at the expense of convergence speed.

AWinA* performs better than ACS and worse than IBS$^+$ for solving instances, but it performs worse than both of them in terms of convergence speed. This may be explained by the fact that, while the first iterations of this algorithm are greedy, it quickly degenerates into A*. When considering the TD-TSPTW$_m$, it performs $O(|\mathcal{V}|)$ iterations, and therefore only provides $O(|\mathcal{V}|)$ solutions. We believe the trade-off between convergence speed and solving speed in AWinA* could be tuned by increasing the depth tolerance every k iterations (instead of increasing it after each iteration), where $k \geq 1$ is a parameter.

AWA* is outperformed by the three other EAS algorithms we consider, both in terms of convergence speed, solving speed, and memory use. Contrarily to ACS, IBS$^+$, and AWinA*, this algorithm does not exploit the layered structure of the graph but instead relies on weighted heuristics. We have seen in Section 6.3 that tuning the heuristic weight w is not trivial as it heavily depends on the size of the state transition graph, which is a function of both the instance size and TW tightness.

6.6 Discussion

In this chapter, we have validated experimentally the relevance of the key components of our solving approach for the TD-TSPTW$_m$. Then, we have considered four EAS algorithms and studied the influence of (i) their parameters and (ii) different implementation decisions. Finally, we have compared the performance of the best configuration of each algorithm: because ACS with breadth-limit $B = 1$ appears to be the most promising for convergence speed, we compare it in Chapter 7 (resp. Chapter 8) to state-of-the-art or recent solving approaches for the TD-TSPTW$_m$ (resp. TSPTW$_m$).

In the future, it would be interesting to apply other EAS algorithms based on weighted heuristics (see Subsection 3.3.3) to the TD-TSPTW$_m$, although they are likely to be more suited to problems for which the state transition graph does not have a layered structure. These algorithms usually adapt the heuristic weight dynamically, which requires performing relatively expensive operations, *i.e.,* either (i) reordering the *open* set, or (ii) maintaining multiple orderings of the *open* set.

Another possibility would be to consider memory-bounded EAS algorithms which are able to perform – to some extent – duplicate detection, given experiments of Section 6.4 have shown us how important it is both for convergence and solving speeds when considering the TD-TSPTW$_m$.

A last way to improve on the experiments of this chapter would be to consider (i) instances of larger size (given the instance size determines the space requirements to store a state), and (ii) instances for which it is not trivial to find feasible solutions (we consider such instances in Chapter 8).

Chapter 7

Experimental comparison with state-of-the-art approaches on Time-Dependent benchmarks

Contents

In this chapter, we experimentally compare our solving approach to recent or state-of-the-art solving approaches on the TD-TSPTW$_m$. We first describe the experimental setting in Section 7.1. Then, in Section 7.2 and Section 7.3, we provide experimental results on benchmarks commonly used to evaluate TD-TSPTW$_m$ solving approaches. After that, in Section 7.4, we perform this comparison on a benchmark with more realistic TD travel times and study how it impacts performance.

7.1 Experimental setting

Considered approaches. We consider three variants of our approach[1], which are based on Anytime Column Search (ACS, see Subsection 4.2.3) with breadth-limit $B = 1$ (we have studied the influence of parameter B in Section 6.3, and found ACS to be the fastest converging of four EAS algorithms in Section 6.5). These three variants only differ in the computation of the lower bound h: we note FEA (resp. OIA and MSA) the variant of ACS in which $h = h_{\text{FEA}}$ (resp. $h = h_{\text{OIA}}$ and $h = h_{\text{MSA}}$).

We compare our solving approach to the ILP approaches of [Ari+19] and [Vu+20], respectively denoted Ari19 and Vu20, to the DP approach of [LMS22] denoted Ler22, and to the LNS approach of Pra23 [Pra23]. Ler22 is the state-of-the-art for solving instances and Pra23 is the

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

state-of-the-art for quickly finding optimal or close-to-optimal solutions. We provide more details about these solving approaches in Section 1.4.

Pra23 is mainly heuristic, *i.e.,* its main goal is to find good approximate solutions quickly. Note however that it can solve highly constrained instances (*i.e.,* prove the optimality of the best-known solution) because it starts by computing a lower bound on the optimal makespan: optimality is proven when it finds a solution whose makespan is equal to this lower bound. Because Pra23 is not deterministic, we executed this approach five times with different random seeds and report results for the median run: runs are compared in lexicographical order according to (i) the makespan of the best solution found and (ii) the time required to find it. Consequently, we consider that Pra23 has found the optimal solution of a given instance when at least three out of the five runs managed to find it. Note that Pra23 has a parameter $W_{\max}$ which determines the size of the DP neighborhoods considered, and thus allows one to balance exploration and exploitation: as recommended by the author, we report results for Pra23 with $W_{\max} = 4$ throughout this thesis.

Also note that in Ler22, resolution starts by looking for a feasible solution using depth-first search (DFS): therefore, although this approach is not anytime, it can provide up to two solutions (*i.e.,* a single approximate solution, and the optimal one). This first solution is usually far from being optimal, unless TWs are very tight: in this case, Ler22 almost instantly finds the optimal solution and proves its optimality.

Our approach is both exact and anytime, *i.e.,* it provides a middle-ground between Pra23 and Ler22 as (i) it provides a sequence of solutions of increasing quality and (ii) it is able to solve any instance, provided enough time and memory.

Considered hardware and performance measures. Whenever possible, we use the same experimental setting as described in Section 6.1, *i.e.,* Pra23, Ler22, FEA, OIA, and MSA are run on 2.1GHz Intel Xeon E5-2620 v4 processors with 64GiB RAM. Run times of Ari19 and Vu20 are those reported by [Ari+19] and [Vu+20], as source codes are not available: Ari19 is run on a 2.33GHz Intel Core 2 Duo processor with 4GiB RAM and Vu20 on a 3.4GHz Intel Core i7-2600 processor (unknown RAM).

All solving approaches were executed with a time limit of one hour per instance. We consider the same performance measures as in Section 6.1, *i.e.,* we note $\#s$ (resp. $\#r$) the number of solved instances (resp. reference solutions found) and t_s (resp. t_r) the average time required to solve (resp. find) them. 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) amongst all approaches which maximize $\#s$ (resp. $\#r$). We say that an instance is solved by an approach whenever it finds an optimal solution and proves its optimality within the time limit. A reference solution is either an optimal solution or a solution of best-known makespan.

Acknowledgements. We thank G. Lera-Romero and C. Pralet for helping us to reproduce their results. We also thank D. M. Vu for sharing their benchmark and results.

7.2 Benchmark B_{ARI19}

In this section, we compare our approach to ARI19, LER22 and PRA23 on benchmark B_{ARI19}. We do not report results of VU20 because (i) this benchmark was not considered in the original publication [Vu+20] and (ii) the solver is not publicly available.

Benchmark. Recall from Section 1.5 that in B_{ARI19} [Ari+19], TW width is determined by parameter $\beta \in \{0, 0.25, 0.50, 1\}$ (TWs are widest when β is close to 0 and tightest when β is close to 1, see Figure 1.5, page 26). B_{ARI19} contains 300 instances for each combination of β and $n \in \{16, 21, 31, 41\}$, *i.e.,* 4800 instances overall. Also recall that TD travel time functions are composed of 73 time-steps and modeled using the IGP model (see Section 1.2).

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 found by (i) PRA23 within one hour or (ii) OIA and MSA within an extended time limit of three hours. The reference solution is known to be optimal for all instances with $n \leq 31$ or $\beta \geq 0.50$. When $n = 41$ and $\beta = 0$ (resp. $\beta = 0.25$), the percentage of instances for which the reference solution is known to be optimal is equal to 42% (resp. 96%)[2].

Results. We provide results about solving speed in Table 7.1a by reporting the number of solved instances and average solving times. We study convergence speed by (i) providing in Table 7.1b the number of reference solutions found and the average time required to find them and (ii) displaying in Figure 7.1 the evolution of the average gap to reference solution with respect to time for the hardest instance classes (*i.e.,* $n = 41$ and $\beta \leq 0.50$).

Comparison of FEA, OIA and MSA. Regarding solving speed, we see in Table 7.1a that MSA is always outperformed by both OIA and FEA in terms of solving speed, showing that using a 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, which shows us that it is beneficial to use a tighter bound when considering larger instances.

Similar conclusions are drawn when considering convergence speed: Table 7.1b shows us that OIA outperforms both FEA and MSA for quickly finding reference solutions, and Figure 7.1 shows us that this conclusion also holds for time limits shorter than one hour.

[2]For this benchmark, optimal makespans slightly differ depending on whether or not travel times are rounded. We round travel times to the nearest integer in PRA23 and our approach, whereas LER22 requires modeling them as floating-points numbers. This problem does not occur when considering more realistic TD travel times in Section 7.4.

Comparison to LER22 and PRA23. We see in Table 7.1a that LER22 manages to solve more instances than our approach when $n = 41$ and $\beta \leq 0.50$. However, when $n = 31$ and $\beta \leq 0.50$, FEA is faster than LER22. Both LER22 and our approach outperform PRA23 for solving instances when $\beta \leq 0.50$ (*i.e.,* when TWs are wide) because fast convergence speed is the main goal of PRA23 (recall that PRA23 can only prove optimality when TWs are tight).

However, unlike LER22, our approach is anytime: when our approach has not solved an instance, it has found approximate solutions which are often optimal. When looking at the evolution of the gap to reference solutions with respect to time for the hardest instance classes (*i.e.,* $n = 41$ and $\beta \leq 0.50$) in Figure 7.1, we see that that OIA converges faster than FEA and MSA and that it reaches an average gap to the reference solution of 1% in 236s (resp. 5s and 0.6s) when $\beta = 0$ (resp. $\beta = 0.25$ and $\beta = 0.50$). We also see that PRA23 in turns converges much faster than OIA, *i.e.,* it respectively requires only 7s, 2s, and 0.4s to reach these goals. On the other hand, LER22 either obtains the optimal solution when it manages to solve an instance, or only provides a relatively low-quality initial solution obtained through DFS. As a comparison, LER22 requires 2511s to reach a 1% average gap when $n = 41$ and $\beta = 0.50$ and fails to reach this goal within one hour when $n = 41$ and $\beta \in \{0, 0.25\}$.

While PRA23 generally excels at quickly finding close-to-optimal solutions, notice that it fails to find two reference solutions when $n = 41$ and $\beta = 0.50$ (some runs of PRA23 managed to find these solutions, but not the majority) whereas OIA and MSA found the reference solutions for all of these instances.

Memory use. When $n = 41$, LER22, FEA, OIA and MSA respectively used – on average – 6, 35, 25 and 7 GiB of memory. This demonstrates that using more expensive lower bounds tends to reduce memory needs. On the other hand, PRA23 used less than 10^{-2} GiB of memory, on average: this is due to the fact that this approach is mainly heuristic, *i.e.,* it can only prove optimality of highly constrained instances.

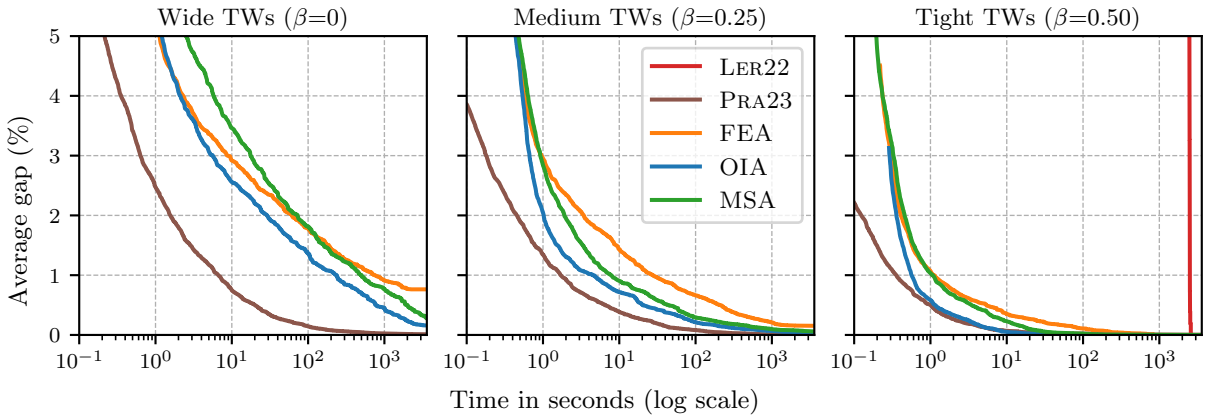


Figure 7.1: Evolution of the average gap to the reference solution for LER22, PRA23, FEA, OIA and MSA on B_{ARI19} when $n = 41$ and $\beta \in \{0, 0.25, 0.50\}$ (300 instances per value of β).

		Ari19		Ler22		Pra23		FEA		OIA		MSA	
n	β	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s
16	0	287	299	300	5	0	-	300	0	300	0	300	0
	0.25	299	143	300	3	0	-	300	0	300	0	300	0
	0.50	299	26	300	2	146	0	300	0	300	0	300	0
	1	300	2	300	0	300	0	300	0	300	0	300	0
21	0	248	660	300	198	0	-	300	1	300	1	300	3
	0.25	286	383	300	87	0	-	300	0	300	0	300	1
	0.50	296	289	300	19	0	-	300	0	300	0	300	0
	1	300	29	300	0	300	0	300	0	300	0	300	0
31	0	155	1631	300	1788	0	-	300	496	294	808	235	1334
	0.25	199	1274	300	1084	0	-	300	145	300	219	299	637
	0.50	157	1433	300	389	0	-	300	5	300	6	300	16
	1	233	608	300	0	300	0	300	0	300	0	300	0
41	0	110	2263	126	2778	0	-	0	-	0	-	0	-
	0.25	131	1950	244	2593	0	-	35	2444	16	2986	0	-
	0.50	55	2276	300	1837	0	-	252	566	280	645	235	1069
	1	106	528	300	0	300	0	300	0	300	0	300	0
Total		3461		4570		1346		4187		4190		4069	

(a) Number of solved instances ($\#s$) and solving time (t_s).

		Ler22		Pra23		FEA		OIA		MSA	
n	β	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r
16	0	300	5	300	0	300	0	300	0	300	0
	0.25	300	3	300	0	300	0	300	0	300	0
	0.50	300	2	300	0	300	0	300	0	300	0
	1	300	0	300	0	300	0	300	0	300	0
21	0	300	198	300	0	300	0	300	0	300	1
	0.25	300	87	300	0	300	0	300	0	300	0
	0.50	300	19	300	0	300	0	300	0	300	0
	1	300	0	300	0	300	0	300	0	300	0
31	0	300	1788	300	6	300	64	300	20	300	67
	0.25	300	1084	300	4	300	19	300	5	300	12
	0.50	300	389	300	1	300	1	300	0	300	1
	1	300	0	300	0	300	0	300	0	300	0
41	0	126	2778	285	241	171	456	235	471	213	648
	0.25	244	2593	300	137	210	318	283	208	268	282
	0.50	300	1837	298	21	299	75	300	6	300	19
	1	300	0	300	0	300	0	300	0	300	0
Total		4570		4783		4580		4718		4681	

(b) Number of reference solutions found ($\#r$) and time to find the reference solution (t_r).

Table 7.1: Instances solved and reference solutions found by Ari19, Ler22, Pra23, FEA, OIA and MSA on B_{Ari19} (300 instances per row).

Comparison to ARI19. Table 7.1a also presents results about the solving performance of ARI19 (recall from Section 7.1 that ARI19 was run on different hardware). We can see that LER22 outperforms it, and that our approach is more successful on many instance classes: OIA solves 729 more instances than ARI19 overall 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 ARI19 is able to solve 241 instances.

Recall from Section 1.5 that TD travel time functions of B_{ARI19} depend on two parameters Δ and P: the success of ARI19 is strongly related to Δ as it relies on bounds which are tighter when Δ is closer to 1, as explained in Section 1.4. To illustrate this, we detail in Table 7.2 the number of solved instances for each value of β, each value of Δ and each traffic pattern P when $n = 41$. It shows us that ARI19 is very sensitive to Δ and P, whereas our approach is mainly sensitive to the TW width β.

Note that the model used to generate TD travel time functionf of B_{ARI19} allows one to control Δ. In Section 7.4, we report results on B_{RIF20}, a benchmark generated from real-world data: in it, the value of Δ is not controlled and it is much lower than 0.70 (see Section 1.5). Before that, we report results on B_{VU20} in which TD travel time functions were generated using a model similar to the one of B_{ARI19}.

	ARI19											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
0.25	6	8	13	23	29	1	0	5	16	30	131	2	2	3	4	4	4	4	4	4	4	35
0.50	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 7.2: Number of instances solved by ARI19 and FEA with respect to P, Δ and β on B_{ARI19} when $n = 41$ (30 instances per class).

7.3 Benchmark B_{VU20}

We now compare our approach to VU20[3], LER22 and PRA23 on benchmark B_{VU20}. We do not present results of ARI19 as this solver is not publicly available.

Context. B_{VU20} [Vu+20] is an extension of B_{ARI19} with similar TD travel time functions but an increased number of vertices $n \in \{60, 80, 100\}$. Also, TWs are obtained in a different way and have a fixed width of w time units, where $w \in \{40, 60, 80, 100, 120, 150\}$: TWs of this benchmark are much tighter than those of B_{ARI19} (see Figure 1.5, page 26). There are 40 instances for each combination of instance size n and TW width w, *i.e.,* 720 instances overall.

Reference solutions are known to be optimal for all instances of this benchmark.

Solving speed. In Table 7.3a, 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 and inexpensive feasibility check of h_{FEA} is often enough to detect inconsistencies.

If FEA is able to solve all instances, VU20, LER22 and PRA23 respectively fail at solving 19, two and 150 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).

Convergence speed. Table 7.3b 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. PRA23 outperforms our approach for convergence speed, *i.e.,* it almost instantly finds all reference solutions. We do not display the evolution of the gap with respect to time on B_{VU20} because TW tightness makes it trivial to find close-to-optimal solutions (TW constraint propagation rules are very efficient in such a context).

Memory use. For instances with $n = 100$, LER22 used on average 0.8 GiB of memory, whereas FEA, OIA and MSA used 0.2 GiB each. This can be explained by the fact that our bounds prune the search space efficiently because of TW tightness. As we have seen in Section 7.2, PRA23 uses a negligible amount of memory compared to the other approaches.

[3] Results of VU20 have been sent to us by authors in a personal communication.

		Vu20		Ler22		Pra23		FEA		OIA		MSA	
n	w	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s
60	≤ 80	120	3.3	120	1.0	119	0.0	120	0.1	120	0.1	120	0.1
	100	40	15.5	40	8.0	36	0.0	40	0.1	40	0.1	40	0.1
	120	40	84.8	40	25.9	30	0.0	40	0.1	40	0.1	40	0.2
	150	39	219.6	40	154.7	20	0.0	40	0.2	40	0.4	40	1.3
80	≤ 80	120	65.4	120	8.4	105	0.0	120	0.2	120	0.2	120	0.2
	100	39	198.3	40	52.7	22	0.0	40	0.2	40	0.3	40	0.7
	120	37	433.3	40	96.6	24	0.0	40	0.4	40	0.8	40	2.3
	150	39	629.4	40	193.2	20	0.0	40	1.5	40	4.3	40	14.3
100	≤ 80	120	59.4	120	58.2	101	0.1	120	0.4	120	0.5	120	1.2
	100	39	292.5	40	219.0	31	0.1	40	1.3	40	3.4	40	11.7
	120	39	435.8	40	365.9	31	0.1	40	5.0	40	16.2	40	55.9
	150	29	1291.1	38	722.5	31	0.1	40	79.4	39	165.1	39	564.6
Total		701		718		570		720		719		719	

(a) Number of solved instances ($\#s$) and solving time (t_s).

		Ler22		Pra23		FEA		OIA		MSA	
n	w	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r
60	≤ 80	120	1.0	120	0.0	120	0.0	120	0.0	120	0.0
	100	40	8.0	40	0.0	40	0.0	40	0.1	40	0.0
	120	40	25.9	40	0.0	40	0.1	40	0.1	40	0.1
	150	40	154.7	40	0.0	40	0.1	40	0.1	40	0.1
80	≤ 80	120	8.4	120	0.0	120	0.1	120	0.1	120	0.1
	100	40	52.7	40	0.0	40	0.1	40	0.1	40	0.1
	120	40	96.6	40	0.0	40	0.2	40	0.2	40	0.2
	150	40	193.2	40	0.0	40	0.2	40	0.2	40	0.3
100	≤ 80	120	58.2	120	0.1	120	0.2	120	0.2	120	0.2
	100	40	219.0	40	0.1	40	0.3	40	0.3	40	0.4
	120	40	365.9	40	0.1	40	0.3	40	0.4	40	0.5
	150	38	722.5	40	0.1	40	3.6	40	9.5	40	33.9
Total		718		720		720		720		720	

(b) Number of reference solutions found ($\#r$) and time to find the reference solution (t_r).

Table 7.3: Instances solved and reference solutions found by Vu20, Ler22, Pra23, FEA, OIA and MSA on B_{Vu20} (40 instances per row when $w \in \{100, 120, 150\}$, 120 instances per row when $w \leq 80$).

7.4 Benchmark B_{RIF20}

We now consider more realistic TD travel time functions by comparing our approach to LER22 and PRA23 on B_{RIF20}.

Context. As we have seen in Section 1.5, TD travel time functions of B_{RIF20} [RCS20] are composed of 120 time-steps and were generated using a realistic traffic simulation built from real-world data. To ease the comparison of results with the ones obtained on B_{ARI19}, we consider instances with $n \in \{21, 31, 41\}$ and TWs similar to those of B_{ARI19}, *i.e.,* TW tightness is controlled by $\beta \in \{0, 0.25, 0.50, 1\}$.

We cannot report results of ARI19 or VU20 on B_{RIF20} as source codes of these approaches are not available. However, relatively poor performance can be expected from both these approaches. Indeed, we have seen that ARI19 is outperformed by LER22 and that ARI19's performance drops when $\Delta < 0.90$ (see Table 7.2): recall from Section 1.5 that in B_{RIF20}, Δ is always smaller than 0.23. Regarding VU20, we have seen in Section 7.3 that this approach performs best when TWs are very tight: nonetheless, it is outperformed by both LER22 and by our approach (*e.g.,* FEA solves more instances than VU20 and is always ten to 100 times as fast as VU20, on average).

In B_{RIF20}, reference solutions are known to be optimal for all instance classes except when $n = 41$ and $\beta = 0$: in this case, 9% of the reference solutions are known to be optimal.

Performance of LER22, PRA23, FEA, OIA and MSA. In Table 7.4, we report performance measures of LER22, PRA23 and our approach on this benchmark. We also report the evolution of the average gap to the reference solution when $n = 41$ and $\beta \in \{0, 0.25, 0.50\}$ in Figure 7.2. Trends in the results of our approach are relatively similar to those obtained on B_{ARI19} (see Section 7.2), *i.e.,* OIA is still the best performing of our three bounds, both for solving and convergence speeds. More generally, when considering solving speed, PRA23 is outperformed by FEA, OIA and MSA, which are in turn outperformed by LER22 on hard instances classes, *i.e.,* when $n = 41$ and $\beta \in \{0, 0.25\}$. The opposite is true when considering convergence speed, *i.e.,* PRA23 outperforms FEA, OIA and MSA, which in turn outperform LER22.

LER22, FEA, OIA and MSA respectively used 7, 31, 15 and 3 GiB of memory.

Performance impact of travel time functions. Note however that LER22 tends to perform worse on B_{RIF20} than on B_{ARI19}, *e.g.,* it failed to solve two instances with $n = 21$ and $\beta = 0$, and solved only 7% of instance class $n = 41$ and $\beta = 0$ whereas it solved 42% of them on B_{ARI19}. On the other hand, PRA23 and our approach tend to perform better on B_{RIF20} than on B_{ARI19}. To better understand these trends, we report in Table 7.5 the results of LER22, PRA23 and our approach on B_{ARI19} and B_{RIF20} when $n = 41$. We also consider a constant variant of B_{RIF20} which we note $B^{\text{c}}_{\text{RIF20}}$ (we provide detailed results on this benchmark in Section 8.3, including a comparison with recent TSPTW$_m$ solving approaches).

		Ler22		Pra23		FEA		OIA		MSA	
n	β	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s
21	0	148	363	0	-	150	0	150	1	150	2
	0.25	150	89	0	-	150	0	150	0	150	0
	0.50	150	15	0	-	150	0	150	0	150	0
	1	150	0	150	0	150	0	150	0	150	0
31	0	149	2176	0	-	149	397	148	488	136	1151
	0.25	150	1503	0	-	150	84	150	68	149	152
	0.50	150	431	0	-	150	1	150	1	150	3
	1	150	0	150	0	150	0	150	0	150	0
41	0	11	2902	0	-	0	-	0	-	0	-
	0.25	132	2744	0	-	12	2236	27	1950	15	1800
	0.50	149	1450	0	-	150	40	150	35	150	120
	1	150	0	150	0	150	0	150	0	150	0
Total		1639		450		1511		1525		1500	

(a) Number of solved instances ($\#s$) and solving time (t_s).

		Ler22		Pra23		FEA		OIA		MSA	
n	β	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r
21	0	148	363	150	0	150	0	150	0	150	0
	0.25	150	89	150	0	150	0	150	0	150	0
	0.50	150	15	150	0	150	0	150	0	150	0
	1	150	0	150	0	150	0	150	0	150	0
31	0	149	2176	150	3	150	67	150	19	150	67
	0.25	150	1503	150	0	150	11	150	1	150	3
	0.50	150	431	150	0	150	0	150	0	150	0
	1	150	0	150	0	150	0	150	0	150	0
41	0	11	2902	150	61	68	397	135	410	116	626
	0.25	132	2744	150	17	120	413	149	51	147	199
	0.50	149	1450	150	1	150	1	150	1	150	3
	1	150	0	150	0	150	0	150	0	150	0
Total		1639		1800		1688		1784		1763	

(b) Number of reference solutions found ($\#r$) and time to find the reference solution (t_r).

Table 7.4: Instances solved and reference solutions found by Ler22, Pra23, FEA, OIA and MSA on B_{Rif20} (150 instances per row).

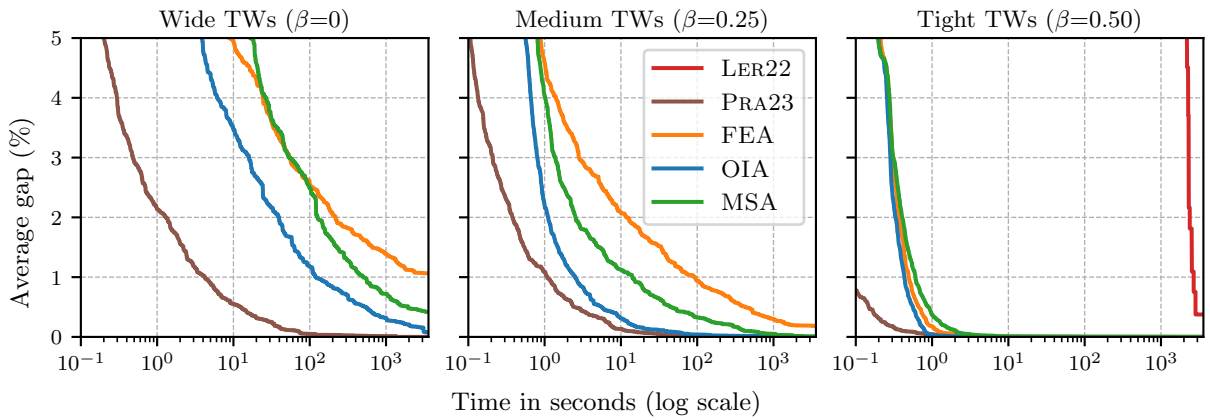


Figure 7.2: Evolution of the average gap to the reference solution for LER22, PRA23, FEA, OIA and MSA on B_{RIF20} when $n = 41$ and $\beta \in \{0, 0.25, 0.50\}$ (150 instances per value of β).

This shall allow us to evaluate the difference in performance when (i) considering the realistic TD travel times of B_{RIF20} instead of the artificial ones of B_{ARI19}, and (ii) optimizing with TD travel time functions instead of constant travel times. In particular, TD travel time functions of B_{RIF20} are composed of 120 time-steps whereas the ones of B_{ARI19} only contain 73 time-steps: they also have a lower value of Δ, *i.e.,* TD travel times vary less homogeneously in B_{RIF20} than in B_{ARI19}.

More specifically, we report in Table 7.5a (resp. Table 7.5b) the percentage of solved instances (resp. of reference solutions found) and time required to solve (resp. find) them. Additionally, we report (i) the time needed by each approach to reach an average gap of 1% for the hardest instance class (*i.e.,* $\beta = 0$) in Table 7.5c, and (ii) the average peak memory use in Table 7.5d. Note that B_{ARI19} (resp. B_{RIF20} and B^c_{RIF20}) contain 1200 (resp. 600, 240) instances with $n = 41$.

As expected, all approaches perform better when travel times are constant (*i.e.,* in B^c_{RIF20}) than when they are time-dependent (*i.e.,* in B_{RIF20} and B_{ARI19}). When looking at the performance difference between B_{ARI19} and B_{RIF20}, we see that:

- LER22 proportionally solves more instances from B_{ARI19} than from B_{RIF20}, and also uses more memory on B_{RIF20} instances.
- PRA23 converges faster on B_{RIF20} than on B_{ARI19}, *i.e.,* it finds more reference solutions and requires less time to reach a 1% average gap.
- The three variants of our approach have better solving performance and use less memory on B_{RIF20} than on B_{ARI19}. Also, OIA converges faster on B_{RIF20} than on B_{ARI19}. On the other hand, FEA's convergence speed is lower (*e.g.,* it does not manage to reach a 1% average gap on B_{RIF20}), and MSA finds more reference solutions, but requires a similar amount of time to reach a 1% average gap.

	Ler22		FEA		OIA		MSA	
	%s	t_s	%s	t_s	%s	t_s	%s	t_s
B_{Ari19}	80.8	1581	48.9	389	49.7	383	44.6	469
B_{Rif20}	73.7	1381	52.0	105	54.5	177	52.5	143
B^c_{Rif20}	88.3	1341	55.4	213	68.3	419	71.7	414

(a) Percentage of solved instances ($\%s$) and mean solving time (t_s).

	Pra23		FEA		OIA		MSA	
	%r	t_r	%r	t_r	%r	t_r	%r	t_r
B_{Ari19}	98.6	98	81.7	170	93.2	153	90.1	203
B_{Rif20}	100.0	20	81.3	157	97.3	108	93.8	182
B^c_{Rif20}	100.0	19	84.6	174	99.6	80	97.1	154

(b) Percentage of reference solutions found ($\%r$) and mean time to find reference solutions (t_r).

	Pra23	FEA	OIA	MSA
B_{Ari19}	7	741	236	496
B_{Rif20}	4	-	119	498
B^c_{Rif20}	1	656	79	492

(c) Time to reach an average gap of 1% when $\beta = 0$.

	Ler22	FEA	OIA	MSA
B_{Ari19}	6	35	25	7
B_{Rif20}	7	31	15	3
B^c_{Rif20}	4	29	5	1

(d) Average peak memory use in GiB.

Table 7.5: Performance measures of Ler22, Pra23, FEA, OIA and MSA on B_{Ari19}, B_{Rif20} and B^c_{Rif20} when $n = 41$.

Discussion. Pra23 and our solving approach appear to be more robust than Ler22 when considering more realistic travel time functions: we believe these functions lead Ler22 to computing looser lower bounds. Also, note that Ler22 uses bidirectional search: searching backwards in the DP state space requires associating pieces of linear functions to subproblems $(i, \mathcal{S})$: in this case, TD travel time functions with more time-steps tend to require modeling a greater set of alternatives, which leads to a greater memory use. On the other hand, both Pra23 and our approach only use forward search in the DP state space: a single value is associated to each subproblem $(i, \mathcal{S})$, independently of the number of time-steps which compose TD travel time functions.

Performance improvements of Pra23 and our approach may partly be explained by the simple fact that TD travel times are modeled differently in B_{Ari19} and in B_{Rif20}: B_{Ari19} uses the IGP model whereas B_{Rif20} uses piecewise-constant functions (see Section 1.2). Consequently, linear time with respect to the number of time-steps is required to compute travel times $c_{i,j}(t)$ in B_{Ari19}, whereas this is achieved in constant time in B_{Rif20}. Ler22 does not benefit from this because it operates on piecewise linear functions which, in both cases, are pre-computed at the beginning of resolution.

Additionally, for our approach, trends in memory and solving performance are improved when considering more realistic TD travel times. However, only OIA converges significantly faster on

B_{RIF20} than on B_{ARI19}. We believe FEA performs better on B_{ARI19} because search requires less guidance due to the relatively homogeneous variations in TD travel times. On the other hand, we believe the more expensive h_{MSA} performs worse on B_{RIF20} because our lower bounds $\underline{c}$ on travel costs are not as tight when TD travel time functions are more variable.

When travel times are constant (*i.e.,* in B^c_{RIF20}), LER22 and our approach perform better than on TD benchmarks because the lower bounds used to guide search are tightest. Note that PRA23 does not rely on lower bounding functions: we may explain the increase in performance of PRA23 and our approach by the fact that computing the travel time function $c_{i,j}(t)$ has better memory locality and requires even less operations than on B_{RIF20}.

7.5 Discussion

In this chapter, we have compared recent and state-of-the-art solving approaches for the TD-TSPTW$_m$, both in terms of solving and convergence speeds. Results have shown LER22 to be the best performing approach for solving instances with wide TWs. This approach, however, is not anytime: the heuristic approach PRA23 clearly outperforms the other approaches on most challenging instance classes, and – due to its heuristic nature – has a very low memory footprint. We have also seen that our approach, which is both exact and anytime, offers a decent compromise between solving and convergence speed, and performs particularly well when TWs are tight. Additionally, our three lower bounds provide different compromises between computational cost and tightness, and therefore different trade-offs between performance and memory use.

We have seen in Section 1.2 that estimations of TD travel times for planning tours in a urban context are inherently imperfect: in such a case, quickly finding close-to-optimal solutions is more desirable than finding the optimal solution and proving its optimality. Approaches which are both exact and anytime are useful in this context, as optimality proofs can be used to determine when to stop searching (or to avoid searching in vain for a feasible solution when none exists).

LP-based approaches ARI19 and VU20 are outperformed on the benchmarks they respectively introduced: ARI19 appears to be most suited to TD travel time functions which vary rather homogeneously, and VU20 to instances with very tight TWs (one could argue that travel times are barely time-dependent when considering such TWs).

As we have seen in Chapter 1, TD travel time functions of benchmarks commonly used to evaluate TD-TSPTW$_m$ solving approaches are not very realistic: consequently, we also reported results on B_{RIF20}, whose TD travel times were obtained using a realistic traffic simulator. Results have shown PRA23 and our approach to be more robust than LER22 when TD travel times vary more often and more heterogeneously. We also compared the performance of these approaches on a variant of B_{RIF20} with constant travel times: results have shown that performance is improved in this context, but also that the difficulty of these instances – for exact approaches – stems mostly from the wideness of TWs and only partly from considering TD travel times.

Note that the TW generation schemes used in the benchmarks of this chapter are not very realistic (*e.g.,* in B_{Ari19}, customers only have a deadline when TWs are widest). Therefore, in the next chapter, we consider constant benchmarks commonly used to evaluate TSPTW solving approaches: these benchmarks contain diverse instance sizes and TW layouts. We shall compare Ler22, Pra23, and our approach to recent TSPTW$_m$ solving approaches on (i) these classic benchmarks and (ii) the variant of B_{Rif20} in which travel times are constant.

Chapter

8 Experimental comparison with other approaches on constant benchmarks

Contents

In this chapter, we consider the TSPTW$_m$, *i.e.,* a special case of the TD-TSPTW$_m$ in which travel times are constant. Our aims are to provide an overview of the performance of recent TSPTW$_m$ solving approaches in a common experimental context and to study how TD solving approaches perform compared to them.

To these ends, we present in Section 8.1 the experimental setting and the approaches considered. Then, we provide results on classic benchmarks with heterogeneous sizes and TW layouts in Section 8.2 before considering a variant of a TD benchmark with constant travel times in Section 8.3.

8.1 Experimental setting

Considered approaches. As in the previous chapter, we consider three variants of our approach which are based on Anytime Column Search (ACS, see Subsection 4.2.3) with breadth-limit $B = 1$: these three variants only differ in the computation of the lower bound h: we note FEA (resp. OIA and MSA) the variant of ACS in which $h = h_{\text{FEA}}$ (resp. $h = h_{\text{OIA}}$ and $h = h_{\text{MSA}}$). Our approach is adapted to handle constant travel times in a straightforward way by setting $\underline{c}_{i,j} = \max(l_i + c_{i,j}, e_j) - l_i$, and $a^{-1}_{i,j}(t) = t - c_{i,j}$.

We summarize in Table 8.1 the approaches we consider for experimental evaluation on the TSPTW$_m$ in this chapter.

Reference	Name	Exact	Anytime	Problem-specific	Type
[DU10]	DAS10	–	✓	✓[a]	Variable Neighborhood Search
[Lóp+13]	LOP13	–	✓	✓	Ant Colony Optimization, Beam Search
[Gil+21]	GIL21	✓	✓	–	Multivalued Decision Diagrams, Branch-and-Bound
[LMS22]	LER22	✓	–[b]	✓[c]	DP, State-Space Relaxations
[Pra23]	PRA23	–[d]	✓	✓	Large Neighborhood Search, DP
[RCR23]	RUD23	✓	✓	–	Multivalued Decision Diagrams, Peel-and-Bound

Table 8.1: Solving approaches considered for experimental evaluation on the TSPTW$_m$. Caveats: [a]DAS10 was originally designed for the travel time objective; [b]LER22 can provide up to two solutions: a single approximate solution (if it manages to find a feasible solution) and an optimal solution (if it also manages to prove optimality); [c]LER22 handles both the makespan and travel time objectives; [d]PRA23 is mainly heuristic but manages to solve highly constrained instances.

We compare our approach to LER22 and PRA23, *i.e.,* solving approaches for the TD-TSPTW$_m$ we evaluated on TD benchmarks in Chapter 7. Recall that PRA23 is mainly heuristic although it can prove optimality on highly constrained instances, and that LER22 is exact but not anytime, although it reports an initial solution computed using depth-first search.

Additionally, we consider the heuristic solving approaches of [DU10] and [Lóp+13]: we respectively refer to these approaches as DAS10 and LOP13. DAS10 is based on Variable Neighborhood Search, and LOP13 on *Beam-ACO*, a hybridization of Beam Search and Ant Colony Optimization.

We also consider two problem-independent solving approaches based on Multivalued Decision Diagrams (MDDs, see Section 2.4): more precisely, these approaches build upon the framework of [Ber+16] which uses a branch-and-bound approach and computes bounds using Restricted Dynamic Programming and State Space Relaxations (see Section 2.3). GIL21 [Gil+21] improves on this approach by computing new bounds (some of them being problem-specific). RUD23 [RCR23] uses a strategy called *peel-and-bound*, which reuses results of previous search iteration for efficiency and therefore allows one to consider wider decisions diagrams. We consider these two approaches because (i) they are recent and anytime, (ii) they were originally evaluated – amongst others – on the TSPTW$_m$, and (iii) they share similarities with our approach, given MDDs are related to DP.

We sought to include to our study the results of the branch-cut-and-price approach of [Pes+20], which is generic enough to handle a wide class of vehicle routing problems. Unfortunately, R. Sadykov informed us in a personal communication that non-trivial adaptations are required to handle the makespan objective (originally, it optimizes the travel time). He also noted that such adaptations are expected to have negative effects on the solver's performance. Consequently, we do not consider approaches based on Integer Linear Programming in this chapter and leave this task as future works.

Specifics of DAS10. Recall from Section 5.3 that DAS10 was originally designed to handle the travel time objective, in the special case where travel times are symmetric and constant. Note that it also requires triangle inequality to be satisfied. We include results of this approach to our analysis anyway to show that it is outperformed by other approaches which do not have such requirements (*i.e.,* these approaches tackle a more general problem) and do not rely on these properties for efficiency.

Consequently, we only provide results of DAS10 on symmetric benchmarks and, when necessary, run it on transformed instances in which we enforced triangle inequality by computing shortest paths. However, it is not possible to adapt DAS10 to the makespan objective as its efficiency also relies on waiting times being excluded from the objective function. Therefore, we consider that a run of DAS10 has succeeded whenever it has found a solution whose travel time is lower than or equal to the best-known makespan for these transformed instances. Because the optimal travel time is always lower than or equal to the optimal makespan (see Section 1.1), proceeding this way does not disadvantage DAS10 (also note that the best travel time found by DAS10 is never lower than the best-known makespan).

Acknowledgements. We thank X. Gillard, G. Lera-Romero, M. López-Ibáñez, C. Pralet and I. Rudich for helping us to reproduce their results on our hardware. We also thank R. Sadykov for answering our questions and authors of [DU10] for publishing their solver's source code.

Considered hardware and performance measures. We use the same experimental setting as described in Section 6.1, *i.e.,* all of the solving approaches considered in this chapter are run on 2.1GHz Intel Xeon E5-2620 v4 processors with 64GiB RAM.

All solving approaches were executed with a time limit of one hour per instance. We consider the same performance measures as in Section 6.1, *i.e.,* we note $\#s$ (resp. $\#r$) the number of solved instances (resp. reference solutions found) and t_s (resp. t_r) the average time required to solve (resp. find) them. 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) amongst all approaches which maximize $\#s$ (resp. $\#r$). We say that an instance is solved by an approach whenever it finds an optimal solution and proves its optimality within the time limit. A reference solution is either an optimal solution or a solution of best-known makespan.

We handle non-deterministic approaches (*i.e.,* DAS10, LOP13 and PRA23) in the same way as in Chapter 7, *i.e.,* we execute these approaches five times per instance with different random seeds, and report results for the median run: runs are compared in lexicographical order according to (i) the makespan of the best solution found and (ii) the time required to find it. Consequently, we consider that non-deterministic approaches have found the reference solution of a given instance when at least three out of the five runs managed to find it.

8.2 Classic benchmarks

Considered benchmarks. We use the same set of benchmarks as in [Gil+21], [RCR23] and [Pra23], plus the benchmark introduced in [DU10], which leads us to a total of 592 instances. The main features of these benchmarks are described in Table 8.2.

Reference	Name	#inst	n		OTW (%)		S
			min	max	min	max	
[Asc96]	Asc	50	11	232	5.3	100.0	
[DU10]	DaS	125	201	401	0.2	4.5	✓
[Dum+95]	Dum	135	21	201	3.9	58.9	✓
[Gen+98]	Gen	130	21	101	21.3	88.9	✓
[Lan+93]	Lan	70	20	60	2.0	12.9	✓
[OT07]	Ohl	25	151	201	24.2	37.2	✓
[Pes+98]	Pes	27	20	45	24.1	100.0	
[PB96]	Pot	30	4	46	23.3	100.0	

Table 8.2: Description of classic 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 smallest and largest instance sizes n, and the minimum and maximum value of OTW (*i.e.,* the percentage of overlapping TWs, see Section 1.5). Column S contains ✓ whenever cost functions are symmetrical.

For these benchmarks, reference solutions are known to be optimal for 590 out of 592 instances, and their makespans are identical to the listing provided in [LB15]. Note that twelve instances remained open in October 2023: we managed to close ten of these instances (the two instances which remain open are `n200w140.003.txt` and `n200w120.002.txt` from benchmark Ohl).

Solving speed. In Table 8.3a, we provide results about solving speed for all approaches except DaS10 and Lop13, which are purely heuristic.

We see that the three variants of our approach are the ones which solve the most instances overall. More precisely, FEA solves 96% of the instances (*i.e.,* all but 23): OIA and MSA solve a few instances less, but they are complementary to FEA. We also see that Rud23 manages to solve more instances than Ler22, which in turns solves more instances than Gil21 and Pra23.

These trends are confirmed when looking at the left-hand side of Figure 8.1, which presents the evolution of the percentage of solved instances with respect to time (we omit results of OIA and MSA for clarity, as they are very similar to FEA's). Note however that (i) Pra23 outperforms FEA for very short time-limits, *i.e.,* time-limits shorter than 0.3s, and (ii) Rud23 is respectively outperformed by Gil21 and Ler22 for time-limits shorter than 130s and 300s.

		Gil21		Ler22		Rud23		Pra23		FEA		OIA		MSA	
	$\#inst$	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s
Asc	50	22	384	48	21	48	184	26	0	50	18	49	1	49	2
DaS	125	110	6	125	485	112	615	101	0	125	13	125	13	125	13
Dum	135	109	188	135	39	126	112	63	0	135	0	135	0	135	1
Gen	130	27	525	91	524	117	233	21	0	117	111	112	53	110	88
Lan	70	70	0	70	0	70	1	70	0	70	0	70	0	70	0
Ohl	25	0	-	0	-	15	2052	2	0	20	50	20	70	20	9
Pes	27	8	118	22	203	25	91	6	0	25	152	26	13	27	66
Pot	30	15	247	25	160	27	136	10	0	27	14	28	42	29	83
Total	592	361		516		540		299		569		565		565	

(a) Number of solved instances ($\#s$) and solving time (t_s).

		DaS10		Lop13		Gil21		Ler22		Rud23		Pra23		FEA		OIA		MSA	
	$\#inst$	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r
Asc	50	-	-	50	1	47	56	48	21	49	100	50	0	50	0	50	0	50	0
DaS	125	124	17	120	477	125	7	125	484	119	462	125	0	125	11	125	11	125	11
Dum	135	135	0	135	0	135	4	135	39	127	75	135	0	135	0	135	0	135	0
Gen	130	98	154	130	0	129	19	91	524	120	119	130	0	130	0	130	1	130	0
Lan	70	70	0	70	0	70	0	70	0	70	1	70	0	70	0	70	0	70	0
Ohl	25	18	520	25	1	25	304	0	-	19	1797	25	0	25	40	25	133	25	44
Pes	27	-	-	27	1	23	143	22	203	26	105	27	0	27	6	27	4	27	1
Pot	30	-	-	30	1	25	52	25	160	28	120	30	0	30	4	30	1	30	9
Total	592	(445/485)		587		579		516		558		592		592		592		592	

(b) Number of reference solutions found ($\#r$) and time to find the reference solution (t_r).

		DaS10		Lop13		Gil21		Ler22		Rud23		Pra23		FEA		OIA		MSA	
	$\#inst$	$\#f$	t_f	$\#f$	t_f	$\#f$	t_f	$\#f$	t_f	$\#f$	t_f	$\#f$	t_f	$\#f$	t_f	$\#f$	t_f	$\#f$	t_f
Asc	50	-	-	50	0	50	23	50	0	50	63	50	0	50	0	50	0	50	0
DaS	125	125	0	123	491	125	7	125	4	122	328	125	0	125	11	125	11	125	11
Dum	135	135	0	135	0	135	4	135	0	128	47	135	0	135	0	135	0	135	0
Gen	130	130	0	130	0	129	13	130	3	128	52	130	0	130	0	130	1	130	0
Lan	70	70	0	70	0	70	0	70	0	70	1	70	0	70	0	70	0	70	0
Ohl	25	25	0	25	0	25	304	25	2	25	553	25	0	25	39	25	132	25	42
Pes	27	-	-	27	0	27	23	24	1	27	78	27	0	27	0	27	3	27	0
Pot	30	-	-	30	0	30	31	25	3	30	14	30	0	30	1	30	1	30	0
Total	592	(485/485)		590		591		584		580		592		592		592		592	

(c) Number of feasible solutions found ($\#f$) and time to find the feasible solution (t_f).

Table 8.3: Instances solved, reference solutions found and feasible solutions found by DaS10, Lop13, Gil21, Ler22, Rud23, Pra23, FEA, OIA and MSA on eight classic TSPTW benchmarks.

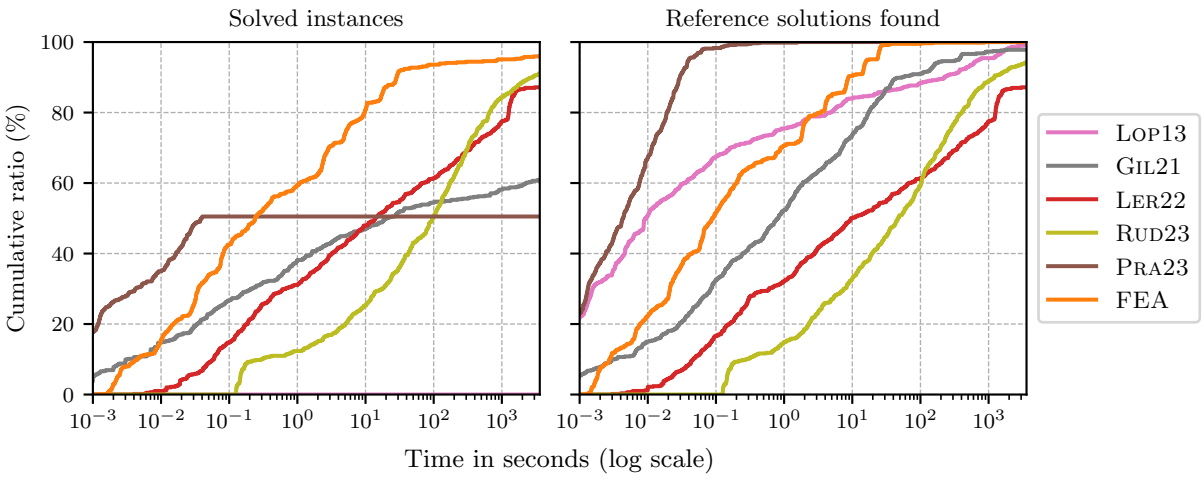


Figure 8.1: Evolution of the cumulative ratio of solved instances (left) and reference solutions found (right) with respect to time for LOP13, GIL21, LER22, RUD23, PRA23 and FEA on eight classic TSPTW benchmarks (592 instances).

Convergence speed. In Table 8.3b, we provide results about convergence speed for all approaches. Results show that PRA23, FEA, OIA and MSA found all reference solutions: PRA23 is the approach which finds them fastest (note that our approach takes more time on benchmark DAS because TW constraint propagation rule $\mathbf{PFR}_2$ defined in Section 1.3 requires $O(n^3)$ time and is therefore not suited to such large instance sizes; also, each of our bounds struggle to find a single reference solution from benchmark OHL.) LOP13 also shows good convergence abilities, although it requires more time and fails to find five reference solutions on the large instances from benchmark DAS. Despite the fact that DAS10 exploits special instance properties for efficiency (see Section 8.1), this approach is clearly outperformed by PRA23, by our approach, and also by LOP13 (except on benchmark DAS). Finally, GIL21 finds 21 more reference solutions than RUD23, which in turns finds 42 more reference solutions than LER22.

We observe similar trends when looking at the right-hand side of Figure 8.1, which presents the evolution of the percentage of reference solutions found with respect to time. It shows us that FEA is outperformed by LOP13 for time-limits shorter than 2s and that RUD23 is outperformed by LER22 for time-limits shorter than 110s.

Finding feasible solutions. Unlike in the TD benchmarks we considered in the previous chapter, it is not trivial for all approaches to find feasible solutions in these classic constant benchmarks. Consequently, in Table 8.3c, we report – for each approach and for each benchmark – the number $\#f$ of instances for which at least one feasible solution has been found and the average time t_f required to find a first feasible solution. We underline the maximal value of $\#f$ and we highlight in red the smallest value of t_f amongst all approaches which maximize $\#f$.

For LOP13, PRA23, FEA, OIA and MSA, trends are very similar to those about reference solutions in Table 8.3b: PRA23 and our approach find feasible solutions to all instances; PRA23 is the fastest, and LOP13 performs very well on all benchmarks except DAS. GIL21 (resp LER22 and RUD23) finds feasible solutions to all but one (resp. 8 and 12) instances. Recall that LER22

uses a depth-first search at the beginning of resolution in order to find a first feasible solution: while it succeeds relatively quickly on most instances, this search did not terminate within the time-limit for three instances of Pes and five instances of Pot.

Memory use. Due to their heuristic nature, DaS10, Lop13 and Pra23 use a negligible amount of memory compared to exact approaches. FEA, OIA and MSA respectively used 2.4, 0.5 and 0.2 GiB of memory, on average: this shows us that using tighter and more expensive lower bounds is beneficial on this dimension. Ler22, Gil21 and Rud23 used significantly more memory than OIA and MSA: they respectively required – on average – 2.6, 6.7 and 16.1 GiB of memory.

Discussion. Pra23 clearly outperforms all of the approaches we consider when trying to quickly find feasible or reference solutions. Lop13 comes close to it (except on large instances); our approach is competitive with but slower than Pra23. While DaS10 finds feasible solutions quickly, it is outperformed by Pra23 when considering convergence speed, even though the efficiency of DaS10 relies on special instances properties (*i.e.,* it tackles a special case of the TSPTW).

Even though Pra23 is not purely exact, it manages to solve more than half of the instances. The three variants of our approach are the ones which solve the most instances overall. The problem-agnostic approaches Gil21 and Rud23 – based on Multivalued Decision Diagrams – both tend to take more time to find feasible solutions than problem-specific approaches. When considering convergence and solving speeds, these approaches appear to be complementary: Gil21 tends to perform better for finding reference solutions, while Rud23 tends to perform better for solving instances. It is interesting to note that Rud23 solves overall more instances than the TD-TSPTW$_m$ solving approach Ler22 (recall from the previous chapter that Ler22 outperformed other TD solving approaches when considering relatively small instances with wide TWs). Also, because Ler22 it not anytime, it is outperformed on most benchmarks by both Gil21 and Rud23 in terms of convergence speed. These results also shed light on a weakness of Ler22, which relies on depth-first search to find a feasible solution: this strategy fails on eight instances from benchmarks Pes and Pot (in these benchmarks, instances never contain more than 46 vertices).

In the next section, we study how these conclusions differ when considering a benchmark composed of relatively small instances with fairly wide TWs.

8.3 Benchmark B^c_{Rif20}

Benchmark. We now consider benchmark B^c_{Rif20}, which we generated from the TD benchmark of [RCS20]: recall from Section 1.5 that authors provided TD travel times functions with different temporal granularities, including instances with constant travel times. We generated TWs for this benchmark using a model similar to the one used to generate TWs of B_{Ari19} and B_{Rif20} (see Section 1.5). TW tightness is therefore controlled by parameter $\beta \in \{0, 0.25, 0.50, 1\}$: TWs are widest when $\beta = 0$ and tightest when $\beta = 1$. As for B_{Rif20}, we consider instance sizes similar to those of B_{Ari19}, *i.e.,* $n \in \{21, 31, 41\}$. We generated 60 instances for each pair of instance size n

and TW tightness β, so this B^c_{RIF20} contains 720 instances overall.

In B_{RIF20}, reference solutions are known to be optimal for all instance classes except when $n = 41$ and $\beta = 0$: in this case, 92% of them are known to be optimal (*i.e.,* all but five).

Solving speed. We provide results about the number of instances solved and time to solve them in Table 8.4a. As we have seen in the previous chapter, LER22 outperforms our approach only on the largest instances with wide TWs (*i.e.,* $n = 41$ and $\beta \in \{0, 0.25\}$); our approach outperforms LER22 when $n < 41$ and $\beta < 1$. RUD23 solves more instances than GIL21, but both these approaches struggle to solve intermediate or large instances sizes when TWs are wide.

Notice that MSA outperforms OIA on the hardest instance classes (*i.e.,* $n = 41$ and $\beta \in \{0, 0.25\}$). Recall that opposite conclusions were drawn when considering TD benchmarks in Chapter 7: this shows us that using this more expensive lower bounding function only pays off when the lower bounds on travel times (see Section 4.4) are tight (these bounds are tightest when considering constant instances given that, in this case, they are equal to the real travel time).

Convergence speed. In Table 8.4b, we display results regarding the reference solutions found. OIA is the variant of our approach which performs best: it outperforms LOP13, but it is outperformed by PRA23 on the hardest instance classes (*i.e.,* $n = 41$ and $\beta \in \{0, 0.25\}$). Overall, LER22 finds less reference solutions than LOP13, PRA23, OIA and MSA and requires more time to find them as it is not anytime. Similarly to what we have seen for solving speed, the problem-independent approaches GIL21 and RUD23 struggle to find reference solutions when $n > 21$ and $\beta < 1$.

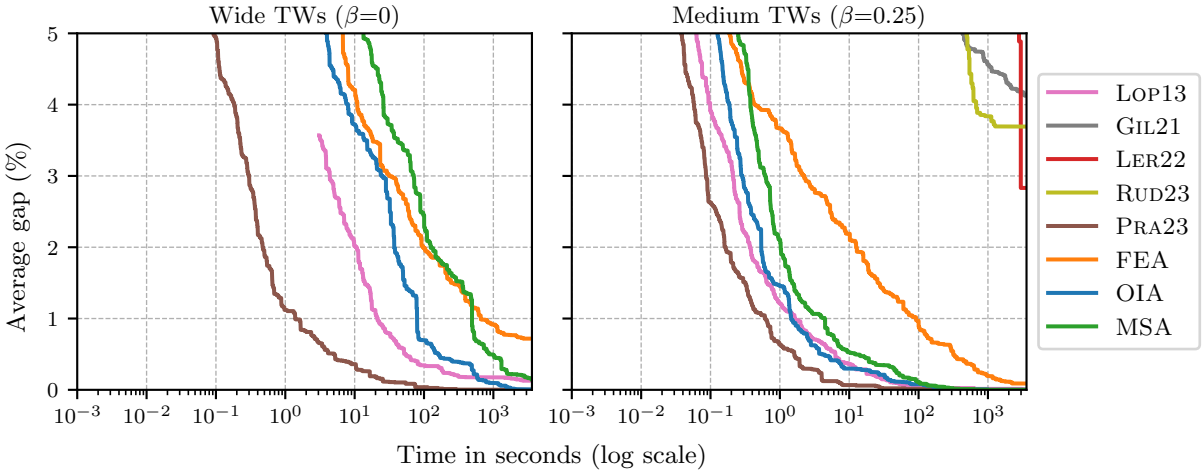


Figure 8.2: Evolution of the average gap to the reference solution for LOP13, GIL21, LER22, RUD23, PRA23, FEA, OIA and MSA on B^c_{RIF20} when $n = 41$ and $\beta \in \{0, .25\}$ (60 instances per class).

When looking at the evolution of the gap to the reference solution on the hardest instance classes in Figure 8.2, we see that GIL21, LER22 and RUD23 are clearly outperformed by the other approaches. PRA23 is the fastest: for instance, when $\beta = 0$, PRA23, LOP13, OIA, MSA and FEA respectively required 1.4s, 22s, 79s, 492s and 656s to reach an average gap of 1%. Notice that, when $\beta = 0$, OIA outperforms LOP13 for time-limits longer than 583s. Also, the difference in performance between PRA23, LOP13, OIA and MSA is not as extreme when $\beta = 0.25$: in this

		Gil21		Ler22		Rud23		Pra23		FEA		OIA		MSA	
n	β	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s	$\#s$	t_s
21	0	60	55	60	59	60	25	0	-	60	0	60	0	60	0
	0.25	60	33	60	16	60	17	0	-	60	0	60	0	60	0
	0.50	60	12	60	5	60	9	1	0	60	0	60	0	60	0
	1	60	0	60	0	60	0	60	0	60	0	60	0	60	0
31	0	0	-	60	1376	10	434	0	-	60	232	60	129	60	177
	0.25	2	2587	60	643	27	361	0	-	60	71	60	29	60	36
	0.50	5	2062	60	129	58	105	0	-	60	1	60	0	60	1
	1	60	0	60	0	60	1	60	0	60	0	60	0	60	0
41	0	0	-	34	2837	0	-	0	-	1	3372	4	2766	5	2011
	0.25	0	-	58	2318	0	-	0	-	12	1858	40	1403	47	1221
	0.50	0	-	60	888	14	350	0	-	60	43	60	26	60	61
	1	60	0	60	0	60	1	60	0	60	0	60	0	60	0
Total		367		692		469		181		613		644		652	

(a) Number of solved instances ($\#s$) and solving time (t_s).

		Lop13		Gil21		Ler22		Rud23		Pra23		FEA		OIA		MSA	
n	β	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r	$\#r$	t_r
21	0	60	3	60	20	60	59	60	18	60	0	60	0	60	0	60	0
	0.25	60	0	60	12	60	16	60	11	60	0	60	0	60	0	60	0
	0.50	60	0	60	4	60	5	60	5	60	0	60	0	60	0	60	0
	1	60	0	60	0	60	0	60	0	60	0	60	0	60	0	60	0
31	0	59	105	20	896	60	1376	16	306	60	0	60	18	60	4	60	11
	0.25	60	11	17	839	60	643	31	307	60	0	60	9	60	1	60	2
	0.50	60	2	35	503	60	129	58	92	60	0	60	0	60	0	60	0
	1	60	0	60	0	60	0	60	0	60	0	60	0	60	0	60	0
41	0	50	239	0	-	34	2837	1	366	60	68	35	528	59	269	53	588
	0.25	59	105	2	584	58	2318	1	280	60	9	48	347	60	54	60	78
	0.50	60	23	14	346	60	888	17	264	60	0	60	2	60	0	60	3
	1	60	0	60	0	60	0	60	1	60	0	60	0	60	0	60	0
Total		708		448		692		484		720		683		719		713	

(b) Number of reference solutions found ($\#r$) and time to find the reference solution (t_r).

Table 8.4: Instances solved and reference solutions found by Lop13, Gil21, Ler22, Rud23, Pra23, FEA, OIA and MSA on B^c_{Rif20} (60 instances per row).

case, LOP13 and OIA show similar performance.

Memory use. When $n = 41$, GIL21, LER22, RUD23, FEA, OIA and MSA respectively required 1.5, 4, 46.4, 29.1, 5.4 and 1 GiB of memory, on average (heuristic approaches LOP13 and PRA23 require only a negligible amount of memory).

Discussion. For TD solving approaches (*i.e.,* PRA23, LER22 and our approach), trends on this constant benchmark are coherent with those observed on TD benchmarks in the previous chapter: PRA23 is best for convergence speed, LER22 is best for solving speed on large and loosely constrained instances, and our approach provides a middle-ground between both approaches.

Note also that OIA is competitive with the heuristic approach of LOP13 in terms of convergence speed. On this benchmark, RUD23 outperforms GIL21 for both solving and convergence speed, but also requires much more memory.

8.4 Discussion

In this chapter, we have compared constant and TD solving approaches on classic TSPTW$_m$ benchmarks and on a benchmark which resembles B_{ARI19} by its TW layouts. We have seen that the best-performing approaches were originally designed to handle TD travel times, both when considering convergence or solving speeds.

In terms of convergence speed, PRA23 is undoubtedly the best of the approaches considered in this study: the heuristic approach LOP13 and our approach are slower than but competitive with PRA23.

Regarding the ability to quickly solve instances, the answer unsurprisingly depends on the instance sizes and TW layouts considered: we have seen that LER22 outperforms other approaches on relatively small instances with wide TWs (*i.e.,* on benchmarks similar to B_{ARI19}). When considering classic TSPTW$_m$ benchmarks with heterogeneous instance sizes and TW layouts, our approach is the one which performs best, and LER22 is outperformed both by our approach and by the problem-agnostic approach RUD23.

Our solving approach is amongst the best-performing when considering both convergence speed and solving speed, and thus appears to provide a good trade-off between both these criteria.

A possibility for future works would be to experiment with benchmarks based on different TW generation schemes, including schemes which do not guarantee instance feasibility. Such benchmarks are likely to contain instances for which it is neither trivial to find a solution nor to prove the inexistence of a feasible solution: we believe exact and anytime solving approaches (like the one we propose) would shine in such cases, whereas purely heuristic algorithms might struggle and search for feasible solutions in vain.

Conclusion

Summary of contributions. In this thesis, we have proposed an exact and anytime solving approach for the TD-TSPTW$_m$ in Chapters 4 and 5, and provided extensive experimental results on both TD and constant benchmarks in Chapters 6 to 8.

More precisely, we proposed in Chapter 4 to exploit EAS algorithms to solve the TD-TSPTW in a scalable way: we first defined a state transition graph associated to its DP formulation and then instantiated four EAS algorithms to solve it. We compared these algorithms and discussed their similarities and differences, which led us to proposing an improved version of an existing EAS algorithm. We also discussed major decisions regarding choices of data structures: we have shown that obvious implementations of A* are not necessarily the best-suited for all EAS algorithms, and also proposed an efficient implementation of an existing EAS algorithm. Finally, we introduced three heuristic functions and discussed their relationship with those of related works. In Chapter 5, we proposed to combine EAS algorithms with local search and TW constraint propagation in order to improve convergence and solving speeds.

In Chapter 6, we experimentally validated the relevance of the main components of our approach and compared four EAS algorithms after having (i) tuned their parameters and (ii) compared different choices of data structures. Finally, in Chapters 7 and 8, we experimentally compared our approach to a variety of state-of-the-art or recent solving approaches on both TD and constant benchmarks. Results have shown (i) our three heuristic functions to be complementary and (ii) our approach to provide a good compromise between convergence and solving speeds. Our approach is generally competitive with or outperforms most of the existing approaches we considered for experimental evaluation. Additionally, because we identified flaws in the TD travel time functions of existing benchmarks, we considered a more realistic benchmark: results have shown our approach to be relatively robust to increases in the granularity of TD travel times functions. Last but not least, we have made our solving approach open-source[1] to favor reproducibility.

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

Future works. Throughout this thesis, we have evoked potential tracks for future works. We now summarize them and delve deeper by categorizing them into four main avenues, namely to (i) improve our TD-TSPTW$_m$ solving approach, (ii) broaden our experimental study, (iii) generalize our approach to other problems, and (iv) build upon our contributions regarding EAS algorithms.

Our solving approach could be improved by computing tighter heuristic functions (see Section 4.4): this could be achieved by computing (i) tighter lower bounds on TD travel times, (ii) a tighter graph G_s, and (iii) tighter relaxations of the shortest Hamiltonian path problem. Another possibility could be to compute them using State Space Relaxations (see Section 2.3), similarly to [LMS22]. An obvious way to improve the convergence speed of our approach would be to improve the solutions reported by EAS algorithms using the DP-based Large Neighborhood Search approach of [Pra23], which has been shown to excel on both TD and constant benchmarks. Alternatively, we could use Algorithm Selection (see, *e.g.,* [Kot16]) to determine which of our three complementary heuristic functions (see, *e.g.,* Section 8.2) – or, more generally, which TD-TSPTW$_m$ solving approach – is expected to perform best based on instance characteristics (*e.g.,* the variability of travel times and instance constrainedness).

We could also broaden our experimental results by evaluating LP-based approaches (*e.g.,* [Pes+20]) and the declarative and EAS-based approach of [KB23c] on constant benchmarks. It would be also useful to propose new metrics for measuring both (i) the variability and heterogeneity of TD travel times (while taking TWs into account, ideally), and (ii) instance constrainedness, *e.g.,* based on the percentage of usable arcs in set $\mathcal{E}$ (the OTW metric used in this thesis does not take travel times into account). Considering other benchmarks with realistic TD travel times (*e.g.,* those of [Agu16] and [Pra23]) also appears to be an essential subject, given some widely used benchmarks are based on unrealistic assumptions (see Section 1.5). Varied TW layouts and instance sizes should also be considered (we have seen in Chapter 8 that it may radically change the performance of some solvers): this includes instances which are not guaranteed to contain feasible solutions, and thus potentially instances for which it is neither trivial to find a solution nor to prove its inexistence. We believe that such critical instances may show the limits of heuristic approaches and expect exact and anytime approaches like ours to be more robust in this case.

Another possibility would be to generalize our approach to other problems, *e.g.,* by handling the TSPTW under the travel time objective: as we have seen in Section 2.2, doing so would require associating a set of Pareto-optimal values to subproblems instead of a single value, and thus increase the size of the search space (the makespan objective is particularly well-suited to DP modelling [Dum+95]). Considering this objective when travel times are TD brings additional difficulties, as it becomes necessary to optimize the departure time from the origin location (see Section 1.1): [LMS22] handles this by associating pieces of linear functions to DP subproblems. In absence of feasible solutions, it might be desirable to consider Flexible TWs [FA20] (*i.e.,* to allow constraint violations, to some extend), Orienteering Problems [Kho+22; TPF23] (*i.e.,* to allow visiting only a subset of customers), or Vehicle Routing Problems ([Hoo16] discusses DP models for many variants of this problem). Adapting our approach to solve these problems would require adapting the heuristic functions, handling the fact that a single subproblem may

have multiple Pareto-optimal solutions, and adapting the state transition graph (*e.g.,* in the Orienteering Problem, final states are not necessarily located in the last layer of this graph). Finally, our approach could be adapted straightforwardly to solve the Sequential Ordering Problem, due to its proximity with the TSPTW$_m$. Preliminary experiments have shown our approach to outperform the related approach of [Lib20]: this is likely to be due to (i) our improvements in the EAS algorithm it uses (*i.e.,* IBS, see Sections 4.2 and 6.4) and (ii) the fact that this approach is based on looser heuristic functions (see Section 4.4).

The last avenue we propose regarding future works concerns EAS algorithms. A key contribution of thesis in this respect is the proposal of an improved variant of IBS, associated to efficient (and atypical) choices of data structures: our contribution could be integrated as a drop-in replacement for the widely used original version of IBS, *e.g.,* in [Lib+20] to tackle the SOP, in [Lib+22] to tackle the Permutation Flowshop Problem, and within the declarative optimization framework of [KB23c], amongst others. More generally, we believe that it would be useful to build an open-source framework for EAS algorithms in order to ease their comparison on other problems. Another possibility would be to integrate these contributions to an existing optimization framework, *e.g.,* to the declarative DP-based framework of [KB23c], or potentially to the framework based on Multivalued Decision Diagrams of [GSC20] (this would allow one to use EAS algorithms as alternatives to branch-and-bound).

List of Figures

List of Tables

List of Algorithms

Bibliography

[ACK07] Sandip Aine, P. P. Chakrabarti, and Rajeev Kumar. "AWA* - A window constrained anytime heuristic search algorithm". In: *IJCAI International Joint Conference on Artificial Intelligence* (2007), pp. 2250–2255. ISSN: 10450823 (cit. on pp. 58, 60, 76, 77, 84).

[AFG01] Norbert Ascheuer, Matteo Fischetti, and Martin Grötschel. "Solving the Asymmetric Travelling Salesman Problem with time windows by branch-and-cut". In: *Mathematical Programming* 90.3 (May 2001), pp. 475–506. ISSN: 0025-5610. DOI: 10.1007/PL00011432 (cit. on p. 20).

[Agu16] Penelope Aguiar-Melgarejo. "A Constraint Programming Approach for the Time Dependent Traveling Salesman Problem". PhD thesis. INSA Lyon, Dec. 2016 (cit. on pp. 17, 19, 28, 138).

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1974. ISBN: 0201000296 (cit. on p. 81).

[ALS15] Penelope Aguiar-Melgarejo, Philippe Laborie, and Christine Solnon. "A time-dependent no-overlap constraint: Application to urban delivery problems". In: *Lecture Notes in Computer Science* 9075 (2015), pp. 1–17. ISSN: 16113349 (cit. on pp. 16–18, 23).

[Ari+18] Anna Arigliano, Tobia Calogiuri, Gianpaolo Ghiani, and Emanuela Guerriero. "A branch-and-bound algorithm for the time-dependent travelling salesman problem". In: *Networks* 72.3 (Oct. 2018), pp. 382–392. ISSN: 0028-3045. DOI: 10.1002/net.21830 (cit. on p. 24).

[Ari+19] Anna Arigliano, Gianpaolo Ghiani, Antonio Grieco, Emanuela Guerriero, and Isaac Plana. "Time-dependent asymmetric traveling salesman problem with time windows: Properties and an exact algorithm". In: *Discrete Applied Mathematics* 261 (May 2019), pp. 28–39. ISSN: 0166218X. DOI: 10.1016/j.dam.2018.09.017 (cit. on pp. 2, 24–26, 29, 113–115).

[Asc+93] Norbert Ascheuer, Laureano F. Escudero, Martin Grötschel, and Mechthild Stoer. "A Cutting Plane Approach to the Sequential Ordering Problem (with Applications to Job Scheduling in Manufacturing)". In: *SIAM Journal on Optimization* 3.1 (Feb. 1993), pp. 25–42. ISSN: 1052-6234. DOI: 10.1137/0803002 (cit. on pp. 14, 38).

[Asc96] Norbert Ascheuer. “Hamiltonian path problems in the on-line optimization of flexible manufacturing systems”. PhD thesis. Zuse Institute Berlin, 1996 (cit. on p. 130).

[Bel57] Richard Bellman. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957. ISBN: 9780691079516 (cit. on p. 38).

[Bel62] Richard Bellman. “Dynamic Programming Treatment of the Travelling Salesman Problem”. In: *Journal of the ACM (JACM)* 9.1 (1962), pp. 61–63. ISSN: 1557735X (cit. on pp. 24, 32).

[Ben+21] Hamza Ben Ticha, Nabil Absi, Dominique Feillet, Alain Quilliot, and Tom Van Woensel. “The Time-Dependent Vehicle Routing Problem with Time Windows and Road-Network Information”. In: *Operations Research Forum* 2.1 (2021) (cit. on p. 20).

[Ber+16] David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and John Hooker. *Decision Diagrams for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer International Publishing, 2016. ISBN: 978-3-319-42847-5. DOI: 10.1007/978-3-319-42849-9 (cit. on pp. 40, 128).

[Bjö+05] Yngvi Björnsson, Markus Enzenberger, Robert C. Holte, and Jonathan Schaeffer. “Fringe Search: Beating A* at Pathfinding on Game Maps”. In: *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05), Essex University, Colchester, Essex, UK, 4-6 April, 2005*. IEEE, 2005 (cit. on p. 75).

[BMR12] Roberto Baldacci, Aristide Mingozzi, and Roberto Roberti. “New State-Space Relaxations for Solving the Traveling Salesman Problem with Time Windows”. en. In: *INFORMS Journal on Computing* 24.3 (Aug. 2012), pp. 356–371. ISSN: 1091-9856. DOI: 10.1287/ijoc.1110.0456 (cit. on pp. 24, 37).

[BS19] Natashia L. Boland and Martin W. P. Savelsbergh. “Perspectives on integer programming for time-dependent models”. In: *TOP* 27.2 (July 2019), pp. 147–173. ISSN: 1134-5764. DOI: 10.1007/s11750-019-00514-4 (cit. on p. 23).

[BSZ21] Abhinav Bhatia, Justin Svegliato, and Shlomo Zilberstein. “On the Benefits of Randomly Adjusting Anytime Weighted A*”. In: *14th International Symposium on Combinatorial Search, SoCS 2021* (2021), pp. 116–120. ISSN: 2832-9171. DOI: 10.1609/socs.v12i1.18558 (cit. on pp. 58, 59).

[CGG14] Jean-François Cordeau, Gianpaolo Ghiani, and Emanuela Guerriero. “Analysis and Branch-and-Cut Algorithm for the Time-Dependent Travelling Salesman Problem”. In: *Transportation Science* 48.1 (Feb. 2014), pp. 46–58. ISSN: 0041-1655. DOI: 10.1287/trsc.1120.0449 (cit. on p. 23).

[CH66] Kenneth L. Cooke and Eric Halsey. “The shortest route through a network with time-dependent internodal transit times”. In: *Journal of Mathematical Analysis and Applications* 14.3 (June 1966), pp. 493–498. ISSN: 0022247X. DOI: 10.1016/0022-247X(66)90009-6 (cit. on p. 18).

[CMT81] Nicos Christofides, Aristide Mingozzi, and Paolo Toth. "State-space relaxation procedures for the computation of bounds to routing problems". In: *Networks* 11.2 (June 1981), pp. 145–164. ISSN: 0028-3045. DOI: 10.1002/net.3230110207 (cit. on pp. 11, 24, 37, 39, 40).

[Coo+11] William J. Cook, David L. Applegate, Robert E. Bixby, and Vasek Chvátal. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2011. ISBN: 9781400841103. DOI: 10.1515/9781400841103 (cit. on p. 23).

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8 (cit. on p. 52).

[DDS92] Martin Desrochers, Jacques Desrosiers, and Marius Solomon. "A New Optimization Algorithm for the Vehicle Routing Problem with Time Windows". In: *Operations Research* 40.2 (Apr. 1992), pp. 342–354. ISSN: 0030-364X. DOI: 10.1287/opre.40.2.342 (cit. on p. 20).

[Des+95] Jacques Desrosiers, Yvan Dumas, Marius M. Solomon, and François Soumis. "Chapter 2 Time constrained routing and scheduling". In: *Network Routing*. Vol. 8. Handbooks in Operations Research and Management Science. Elsevier, 1995, pp. 35–139. DOI: 10.1016/S0927-0507(05)80106-9 (cit. on p. 20).

[Dij59] Edsger W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390 (cit. on p. 51).

[Don+08] Alberto V. Donati, Roberto Montemanni, Norman Casagrande, Andrea E. Rizzoli, and Luca M. Gambardella. "Time dependent vehicle routing problem with a multi ant colony system". In: *European Journal of Operational Research* 185.3 (Mar. 2008), pp. 1174–1191. ISSN: 03772217. DOI: 10.1016/j.ejor.2006.06.047 (cit. on p. 23).

[DP85] Rina Dechter and Judea Pearl. "Generalized best-first search strategies and the optimality of A*". In: *Journal of the ACM* 32.3 (July 1985), pp. 505–536. ISSN: 0004-5411. DOI: 10.1145/3828.3830 (cit. on p. 54).

[DU10] Rodrigo Ferreira Da Silva and Sebastián Urrutia. "A General VNS heuristic for the traveling salesman problem with time windows". In: *Discrete Optimization* 7.4 (Nov. 2010), pp. 203–211. ISSN: 15725286. DOI: 10.1016/j.disopt.2010.04.002 (cit. on pp. 95, 128–130).

[Dum+95] Yvan Dumas, Jacques Desrosiers, Eric Gelinas, and Marius M. Solomon. "An Optimal Algorithm for the Traveling Salesman Problem with Time Windows". In: *Operations Research* 43.2 (Apr. 1995), pp. 367–371. ISSN: 0030-364X. DOI: 10.1287/opre.43.2.367 (cit. on pp. 69, 89, 94, 130, 138).

[FA20] Ramon Faganello Fachini and Vinícius Amaral Armentano. "Exact and heuristic dynamic programming algorithms for the traveling salesman problem with flexible time windows". In: *Optimization Letters* 14.3 (Apr. 2020), pp. 579–609. ISSN: 1862-4472. DOI: 10.1007/s11590-018-1342-y (cit. on pp. 15, 138).

[FDS23a] Romain Fontaine, Jilles Dibangoye, and Christine Solnon. "Exact and Anytime Approach for Solving the Time Dependent Traveling Salesman Problem with Time Windows". In: *24ème congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision*. 2023 (cit. on p. 3).

[FDS23b] Romain Fontaine, Jilles Dibangoye, and Christine Solnon. "Exact and anytime approach for solving the time dependent traveling salesman problem with time windows". In: *European Journal of Operational Research* 311.3 (2023), pp. 833–844. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2023.06.001 (cit. on p. 3).

[FGG04] Bernhard Fleischmann, Martin Gietz, and Stefan Gnutzmann. "Time-Varying Travel Times in Vehicle Routing". In: *Transportation Science* 38.2 (May 2004), pp. 160–173. ISSN: 0041-1655. DOI: 10.1287/trsc.1030.0062 (cit. on pp. 18, 19).

[Fic+21] Johannes Klaus Fichte, Markus Hecher, Ciaran McCreesh, and Anas Shahab. "Complications for Computational Experiments from Modern Processors". In: *27th International Conference on Principles and Practice of Constraint Programming, CP 2021*. Ed. by Laurent D Michel. Vol. 210. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 25:1–25:21. DOI: 10.4230/LIPICS.CP.2021.25 (cit. on p. 101).

[FT87] Michael L. Fredman and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *Journal of the ACM* 34.3 (July 1987), pp. 596–615. ISSN: 0004-5411. DOI: 10.1145/28869.28874 (cit. on p. 52).

[Gab+86] Harold N. Gabow, Zvi Galil, Thomas Spencer, and Robert E. Tarjan. "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs". In: *Combinatorica* 6.2 (June 1986), pp. 109–122. ISSN: 0209-9683. DOI: 10.1007/BF02579168 (cit. on p. 88).

[Gen+98] Michel Gendreau, Alain Hertz, Gilbert Laporte, and Mihnea Stan. "A Generalized Insertion Heuristic for the Traveling Salesman Problem with Time Windows". In: *Operations Research* 46.3 (June 1998), pp. 330–335. ISSN: 0030-364X. DOI: 10.1287/opre.46.3.330 (cit. on p. 130).

[GGG15] Michel Gendreau, Gianpaolo Ghiani, and Emanuela Guerriero. "Time-dependent routing problems: A review". In: *Computers and Operations Research* 64 (Dec. 2015), pp. 189–197. ISSN: 03050548. DOI: 10.1016/j.cor.2015.06.001 (cit. on p. 23).

[Gil+21] Xavier Gillard, Vianney Coppé, Pierre Schaus, and André Augusto Cire. "Improving the Filtering of Branch-and-Bound MDD Solver". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Vol. 12735 LNCS. 2021, pp. 231–247. ISBN: 9783030782290. DOI: 10.1007/978-3-030-78230-6_15 (cit. on pp. 41, 128, 130).

[Gmi+21] Maha Gmira, Michel Gendreau, Andrea Lodi, and Jean-Yves Potvin. "Tabu search for the time-dependent vehicle routing problem with time windows on a road network". In: *European Journal of Operational Research* 288.1 (Jan. 2021), pp. 129–140. ISSN: 03772217. DOI: 10.1016/j.ejor.2020.05.041 (cit. on p. 23).

[Gra+79] Ronald Lewis Graham, Eugene Leighton Lawler, Jan Karel Lenstra, and A.H.G. Rinnooy Kan. "Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey". In: *Annals of Discrete Mathematics*. Vol. 5. C. 1979, pp. 287–326. DOI: 10.1016/S0167-5060(08)70356-X (cit. on p. 14).

[GS22] Xavier Gillard and Pierre Schaus. "Large Neighborhood Search with Decision Diagrams". In: *IJCAI International Joint Conference on Artificial Intelligence* (2022), pp. 4754–4760. ISSN: 10450823. DOI: 10.24963/ijcai.2022/659 (cit. on p. 41).

[GSC20] Xavier Gillard, Pierre Schaus, and Vianney Coppé. "Ddo, a Generic and Efficient Framework for MDD-Based Optimization". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. Ed. by Christian Bessiere. ijcai.org, 2020, pp. 5243–5245. DOI: 10.24963/IJCAI.2020/757 (cit. on p. 139).

[HHE18] Steven Halim, Felix Halim, and Suhendry Effendy. *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*. 4th ed. Vol. 1. 2018. ISBN: 9781716745522 (cit. on p. 80).

[HK62] Michael Held and Richard M. Karp. "A Dynamic Programming Approach to Sequencing Problems". In: *Journal of the Society for Industrial and Applied Mathematics* 10.1 (1962), pp. 196–210. ISSN: 03684245 (cit. on p. 32).

[HM23] Claire Hanen and Alix Munier Kordon. "Fixed-parameter tractability of scheduling dependent typed tasks subject to release times and deadlines". In: *Journal of Scheduling* (July 2023). ISSN: 1094-6136. DOI: 10.1007/s10951-023-00788-4 (cit. on p. 15).

[HMM08] Pierre Hansen, Nenad Mladenović, and José A. Moreno Pérez. "Variable neighbourhood search: methods and applications". In: *4OR* 6.4 (Dec. 2008), pp. 319–360. ISSN: 1619-4500. DOI: 10.1007/s10288-008-0089-1 (cit. on p. 95).

[HNR68] Peter Hart, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136 (cit. on p. 53).

[Hoa61] Charles A. R. Hoare. "Algorithm 65: Find". In: *Communications of the ACM* 4.7 (1961), pp. 321–322. ISSN: 15577317. DOI: 10.1145/366622.366647 (cit. on p. 81).

[Hoo13] John N. Hooker. "Decision Diagrams and Dynamic Programming". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7874 LNCS. 2013, pp. 94–110. ISBN: 9783642381706. DOI: 10.1007/978-3-642-38171-3_7 (cit. on p. 40).

[Hoo16] Jelke J. van Hoorn. "Dynamic Programming for Routing and Scheduling: Optimizing Sequences of Decisions". PhD thesis. Vrije Universiteit, 2016. ISBN: 978-94-6332-008-5 (cit. on pp. 24, 38, 138).

[HZ07] Eric A. Hansen and Rong Zhou. "Anytime heuristic search". In: *Journal of Artificial Intelligence Research* 28 (2007), pp. 267–297. ISSN: 10769757. DOI: 10.1613/jair.2096 (cit. on pp. 56, 58, 59, 71, 72, 80).

[IGP03] Soumia Ichoua, Michel Gendreau, and Jean-Yves Potvin. "Vehicle dispatching with time-dependent travel times". In: *European Journal of Operational Research* 144.2 (Jan. 2003), pp. 379–396. ISSN: 03772217. DOI: 10.1016/S0377-2217(02)00147-9 (cit. on pp. 12, 19, 23).

[II99] Takahiro Ikeda and Hiroshi Imai. "Enhanced A* algorithms for multiple alignments: optimal alignments for several sequences and k-opt approximate alignments for large cases". In: *Theoretical Computer Science* 210.2 (Jan. 1999), pp. 341–374. ISSN: 03043975. DOI: 10.1016/S0304-3975(98)00093-0 (cit. on pp. 55, 96).

[Kar72] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations*. Boston, MA: Springer US, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9 (cit. on p. 15).

[KB23a] Ryo Kuroiwa and J. Christopher Beck. "Domain-Independent Dynamic Programming: Generic State Space Search for Combinatorial Optimization". In: *Proceedings of the International Conference on Automated Planning and Scheduling* 33.1 (July 2023), pp. 236–244. ISSN: 2334-0843. DOI: 10.1609/icaps.v33i1.27200 (cit. on p. 41).

[KB23b] Ryo Kuroiwa and J. Christopher Beck. "Large Neighborhood Beam Search for Domain-Independent Dynamic Programming". In: *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Ed. by Roland H C Yap. Vol. 280. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Sept. 2023, 23:1–23:22. ISBN: 978-3-95977-300-3. DOI: 10.4230/LIPIcs.CP.2023.23 (cit. on p. 41).

[KB23c] Ryo Kuroiwa and J. Christopher Beck. "Solving Domain-Independent Dynamic Programming Problems with Anytime Heuristic Search". In: *Proceedings of the International Conference on Automated Planning and Scheduling* 33.1 (July 2023), pp. 245–253. ISSN: 2334-0843. DOI: 10.1609/icaps.v33i1.27201 (cit. on pp. 41, 56, 72, 89, 138, 139).

[Kho+22] Masoud Khodadadian, Ali Divsalar, Cedric Verbeeck, Aldy Gunawan, and Pieter Vansteenwegen. "Time dependent orienteering problem with time windows and service time dependent profits". In: *Computers and Operations Research* 143.March (2022), p. 105794. ISSN: 03050548. DOI: 10.1016/j.cor.2022.105794 (cit. on pp. 15, 138).

[KK97] Hermann Kaindl and Gerhard Kainz. "Bidirectional Heuristic Search Reconsidered". In: *Journal of Artificial Intelligence Research* 7 (1997), pp. 283–317. DOI: 10.1613/jair.460 (cit. on p. 55).

[Kor85] Richard E. Korf. “Depth-first iterative-deepening: An optimal admissible tree search”. In: *Artificial Intelligence* 27.1 (1985), pp. 97–109. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(85)90084-0 (cit. on p. 55).

[Kor90] Richard E. Korf. “Real-time heuristic search”. In: *Artificial Intelligence* 42.2-3 (Mar. 1990), pp. 189–211. ISSN: 00043702. DOI: 10.1016/0004-3702(90)90054-4 (cit. on p. 56).

[Kor93] Richard E. Korf. “Linear-space best-first search”. In: *Artificial Intelligence* 62.1 (July 1993), pp. 41–78. ISSN: 00043702. DOI: 10.1016/0004-3702(93)90045-D (cit. on p. 55).

[Kot16] Lars Kotthoff. “Algorithm selection for combinatorial search problems: A survey”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10101 LNCS (2016), pp. 149–190. ISSN: 16113349. DOI: 10.1007/978-3-319-50137-6_7. arXiv: 1210.7959 (cit. on p. 138).

[KS93] David E. Kaufman and Robert L. Smith. “Fastest paths in time-dependent networks for intelligent vehicle-highway systems application”. In: *Journal of Intelligent Transportation Systems* 1.1 (1993), pp. 1–11 (cit. on p. 17).

[KSJ09] Gio K. Kao, Edward C. Sewell, and Sheldon H. Jacobson. “A branch, bound, and remember algorithm for the $1\,|r_i|\,\Sigma t_i$ scheduling problem”. In: *Journal of Scheduling* 12.2 (Apr. 2009), pp. 163–175. ISSN: 1094-6136. DOI: 10.1007/s10951-008-0087-3 (cit. on pp. 58, 59, 74).

[Lan+93] André Langevin, Martin Desrochers, Jacques Desrosiers, Sylvie Gélinas, and François Soumis. “A two-commodity flow formulation for the traveling salesman and the makespan problems with time windows”. In: *Networks* 23.7 (Oct. 1993), pp. 631–640. ISSN: 0028-3045. DOI: 10.1002/net.3230230706 (cit. on pp. 20, 130).

[LB15] Manuel López-Ibáñez and Christian Blum. *Benchmark Instances for the Travelling Salesman Problem with Time Windows*. 2015. URL: https://lopez-ibanez.eu/tsptw-instances (visited on 04/10/2024) (cit. on p. 130).

[LF21] Luc Libralesso and Florian Fontan. “An anytime tree search algorithm for the 2018 ROADEF/EURO challenge glass cutting problem”. In: *European Journal of Operational Research* 291.3 (2021), pp. 883–893. ISSN: 03772217. DOI: 10.1016/j.ejor.2020.10.050. arXiv: 2004.00963 (cit. on p. 56).

[LGT03] Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. “ARA*: Anytime A* with Provable Bounds on Sub-Optimality”. In: *Advances in Neural Information Processing Systems 16 [NIPS 2003]*. Ed. by Sebastian Thrun, Lawrence K Saul, and Bernhard Schölkopf. MIT Press, 2003, pp. 767–774 (cit. on pp. 56, 58, 59).

[Lib+20] Luc Libralesso, Abdel-Malik Bouhassoun, Hadrien Cambazard, and Vincent Jost. "Tree Search for the Sequential Ordering Problem". In: *ECAI 2020 - 24th European Conference on Artificial Intelligence*. Vol. 325. Frontiers in Artificial Intelligence and Applications. IOS Press, 2020, pp. 459–465. DOI: 10.3233/FAIA200126 (cit. on pp. 58, 60, 72, 77, 82, 83, 87, 89, 107–109, 139).

[Lib+22] Luc Libralesso, Pablo Andres Focke, Aurélien Secardin, and Vincent Jost. "Iterative beam search algorithms for the permutation flowshop". In: *European Journal of Operational Research* 301.1 (Aug. 2022), pp. 217–234. ISSN: 03772217. DOI: 10.1016/j.ejor.2021.10.015. arXiv: 2009.05800 (cit. on pp. 72, 139).

[Lib20] Luc Libralesso. "Anytime tree search algorithms for combinatorial optimization". PhD thesis. Université Grenoble Alpes, 2020 (cit. on pp. 56, 139).

[Lik+08] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. "Anytime search in dynamic graphs". In: *Artificial Intelligence* 172.14 (Sept. 2008), pp. 1613–1643. ISSN: 00043702. DOI: 10.1016/j.artint.2007.11.009 (cit. on p. 59).

[LMS22] Gonzalo Lera-Romero, Juan José Miranda Bront, and Francisco J. Soulignac. "Dynamic Programming for the Time-Dependent Traveling Salesman Problem with Time Windows". In: *INFORMS Journal on Computing* 34.6 (Nov. 2022), pp. 3292–3308. ISSN: 1091-9856. DOI: 10.1287/ijoc.2022.1236 (cit. on pp. 2, 24, 25, 39, 93, 113, 128, 138).

[Lóp+13] Manuel López-Ibáñez, Christian Blum, Jeffrey W. Ohlmann, and Barrett W. Thomas. "The travelling salesman problem with time windows: Adapting algorithms from travel-time to makespan optimization". In: *Applied Soft Computing Journal* 13.9 (2013), pp. 3806–3815. ISSN: 15684946. DOI: 10.1016/j.asoc.2013.05.009 (cit. on p. 128).

[Low76] Bruce T. Lowerre. "The HARPY speech recognition system". PhD thesis. Carnegie-Mellon University, Apr. 1976 (cit. on p. 59).

[Mal89] Chryssi Malandraki. *Time-dependent vehicle routing problems: Formulations, solution algorithms and computational experiments*. 1989 (cit. on p. 18).

[MD92] Chryssi Malandraki and Mark S. Daskin. "Time dependent vehicle routing problems: Formulations, properties and heuristic algorithms". In: *Transportation Science* 26.3 (Aug. 1992), pp. 185–200. ISSN: 00411655. DOI: 10.1287/trsc.26.3.185 (cit. on pp. 1, 11, 15, 18, 23).

[MD96] Chryssi Malandraki and Robert B. Dial. "A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem". In: *European Journal of Operational Research* 90.1 (Apr. 1996), pp. 45–55. ISSN: 03772217. DOI: 10.1016/0377-2217(94)00299-1 (cit. on pp. 24, 36, 37, 39).

[MH97] Nenad Mladenović and Pierre Hansen. "Variable neighborhood search". In: *Computers and Operations Research* 24.11 (1997), pp. 1097–1100. DOI: 10.1016/S0305-0548(97)00031-2 (cit. on p. 95).

[MMM17] Agustín Montero, Isabel Méndez-Díaz, and Juan José Miranda-Bront. "An integer programming approach for the time-dependent traveling salesman problem with time windows". In: *Computers and Operations Research* 88 (Dec. 2017), pp. 280–289. ISSN: 03050548. DOI: 10.1016/j.cor.2017.06.026 (cit. on pp. 20, 23).

[Mus97] David R. Musser. "Introspective sorting and selection algorithms". In: *Software - Practice and Experience* 27.8 (1997), pp. 983–993. ISSN: 00380644 (cit. on p. 81).

[OT07] Jeffrey W. Ohlmann and Barrett W. Thomas. "A Compressed-Annealing Heuristic for the Traveling Salesman Problem with Time Windows". In: *INFORMS Journal on Computing* 19.1 (Feb. 2007), pp. 80–90. ISSN: 1091-9856. DOI: 10.1287/ijoc.1050.0145 (cit. on p. 130).

[PB96] Jean-Yves Potvin and Samy Bengio. "The Vehicle Routing Problem with Time Windows Part II: Genetic Search". In: *INFORMS Journal on Computing* 8.2 (May 1996), pp. 165–172. ISSN: 1091-9856. DOI: 10.1287/ijoc.8.2.165 (cit. on p. 130).

[Pea84] Judea Pearl. *Heuristics - intelligent search strategies for computer problem solving*. Addison-Wesley series in artificial intelligence. Addison-Wesley, 1984. ISBN: 978-0-201-05594-8 (cit. on pp. 78, 111).

[Pes+20] Artur Pessoa, Ruslan Sadykov, Eduardo Uchoa, and François Vanderbeck. "A generic exact solver for vehicle routing and related problems". In: *Mathematical Programming* 183.1-2 (2020), pp. 483–523. ISSN: 14364646. DOI: 10.1007/s10107-020-01523-z (cit. on pp. 128, 138).

[Pes+98] Gilles Pesant, Michel Gendreau, Jean-Yves Potvin, and Jean-Marc Rousseau. "An Exact Constraint Logic Programming Algorithm for the Traveling Salesman Problem with Time Windows". In: *Transportation Science* 32.1 (Feb. 1998), pp. 12–29. ISSN: 0041-1655. DOI: 10.1287/trsc.32.1.12 (cit. on p. 130).

[Poh69] Ira Pohl. "Bi-directional and heuristic search in path problems". PhD thesis. Stanford University, USA, 1969 (cit. on p. 55).

[Poh70] Ira Pohl. "Heuristic Search Viewed as Path Finding in a Graph". In: *Artificial Intelligence* 1.3 (1970), pp. 193–204 (cit. on p. 59).

[Poh71] Ira Pohl. "Bi-directional search". In: *Machine Intelligence*. Vol. 6. Edinburgh University Press, 1971, pp. 127–140 (cit. on p. 55).

[Pra23] Cédric Pralet. "Iterated Maximum Large Neighborhood Search for the Traveling Salesman Problem with Time Windows and its Time-dependent Version". In: *Computers and Operations Research* 150 (Feb. 2023), p. 106078. ISSN: 03050548. DOI: 10.1016/j.cor.2022.106078 (cit. on pp. 2, 15, 24, 25, 28, 93, 94, 113, 128, 130, 138).

[RCR23] Isaac Rudich, Quentin Cappart, and Louis-Martin Rousseau. "Improved Peel-and-Bound: Methods for Generating Dual Bounds with Multivalued Decision Diagrams". In: *Journal of Artificial Intelligence Research* 77 (Aug. 2023), pp. 1489–1538. ISSN: 1076-9757. DOI: 10.1613/jair.1.14607 (cit. on pp. 41, 128, 130).

[RCS20] Omar Rifki, Nicolas Chiabaut, and Christine Solnon. "On the impact of spatio-temporal granularity of traffic conditions on the quality of pickup and delivery optimal tours". In: *Transportation Research Part E: Logistics and Transportation Review* 142 (Oct. 2020), p. 102085. ISSN: 13665545. DOI: 10.1016/j.tre.2020.102085 (cit. on pp. 1, 16–18, 24, 25, 27–29, 38, 121, 133).

[RN10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education, 2010. ISBN: 978-0-13-207148-2 (cit. on p. 56).

[RT12] Roberto Roberti and Paolo Toth. "Models and algorithms for the Asymmetric Traveling Salesman Problem: an experimental comparison". In: *EURO Journal on Transportation and Logistics* 1.1-2 (June 2012), pp. 113–133. ISSN: 21924376. DOI: 10.1007/s13676-012-0010-0 (cit. on p. 88).

[RTR10] Silvia Richter, Jordan T. Thayer, and Wheeler Ruml. "The joy of forgetting: Faster anytime search via restarting". In: *ICAPS 2010 - Proceedings of the 20th International Conference on Automated Planning and Scheduling* Icaps (2010), pp. 137–144. ISSN: 2334-0835. DOI: 10.1609/icaps.v20i1.13412 (cit. on pp. 58, 59).

[Sal19] Julien Salotti. "Méthodes de sélection de voisinage et de prévision à court-terme pour l'analyse du trafic urbain". Theses. INSA Lyon ; Université de Lyon, Sept. 2019 (cit. on p. 1).

[Sav85] Martin W. P. Savelsbergh. "Local search in routing problems with time windows". In: *Annals of Operations Research* 4.1 (Dec. 1985), pp. 285–305. ISSN: 0254-5330. DOI: 10.1007/BF02022044 (cit. on p. 15).

[Sun+20] Peng Sun, Lucas P. Veelenturf, Mike Hewitt, and Tom Van Woensel. "Adaptive large neighborhood search for the time-dependent profitable pickup and delivery problem with time windows". In: *Transportation Research Part E* 138 (2020), p. 101942. ISSN: 13665545 (cit. on p. 23).

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011. ISBN: 978-0-321-57351-3 (cit. on p. 80).

[TI17] Christian Tilk and Stefan Irnich. "Dynamic Programming for the Minimum Tour Duration Problem". In: *Transportation Science* 51.2 (May 2017), pp. 549–565. ISSN: 0041-1655. DOI: 10.1287/trsc.2015.0626 (cit. on p. 24).

[TPF23] Trong-Hieu Tran, Cédric Pralet, and Hélène Fargier. "Combining Incomplete Search and Clause Generation: An Application to the Orienteering Problems with Time Windows". In: *Integration of Constraint Programming, Artificial Intelligence, and*

Operations Research. Ed. by Andre A Cire. Cham: Springer Nature Switzerland, 2023, pp. 493–509. ISBN: 978-3-031-33271-5 (cit. on pp. 15, 138).

[TR10] Jordan Thayer and Wheeler Ruml. "Anytime heuristic search: Frameworks and algorithms". In: *Proceedings of the 3rd Annual Symposium on Combinatorial Search, SoCS 2010* (2010), pp. 121–128. ISSN: 2832-9171. DOI: 10.1609/socs.v1i1.18181 (cit. on p. 56).

[VAC11] Satya Gautam Vadlamudi, Sandip Aine, and Partha Pratim Chakrabarti. "MAWA* - A Memory-Bounded Anytime Heuristic-Search Algorithm". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 41.3 (June 2011), pp. 725–735. ISSN: 1083-4419. DOI: 10.1109/TSMCB.2010.2089619 (cit. on pp. 58, 60).

[VAC16] Satya Gautam Vadlamudi, Sandip Aine, and Partha Pratim Chakrabarti. "Anytime pack search". In: *Natural Computing* 15.3 (2016), pp. 395–414. ISSN: 15729796. DOI: 10.1007/s11047-015-9490-9 (cit. on pp. 58, 60).

[Vad+12] Satya Gautam Vadlamudi, Piyush Gaurav, Sandip Aine, and Partha Pratim Chakrabarti. "Anytime Column Search". In: *AI 2012: Advances in Artificial Intelligence - 25th Australasian Joint Conference*. Vol. 7691. LNCS. Springer, 2012, pp. 254–265. DOI: 10.1007/978-3-642-35101-3_22 (cit. on pp. 58, 59, 74, 77).

[Van+11] Jur Van den Berg, Rajat Shah, Arthur Huang, and Ken Goldberg. "Anytime Non-parametric A*". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 25.1 (Aug. 2011), pp. 105–111. ISSN: 2374-3468. DOI: 10.1609/aaai.v25i1.7819 (cit. on pp. 58, 59).

[VHV22] Duc Minh Vu, Mike Hewitt, and Duc D. Vu. "Solving the time dependent minimum tour duration and delivery man problems with dynamic discretization discovery". In: *European Journal of Operational Research* 302.3 (Nov. 2022), pp. 831–846. ISSN: 03772217. DOI: 10.1016/j.ejor.2022.01.029 (cit. on p. 23).

[Vu+20] Duc Minh Vu, Mike Hewitt, Natashia Boland, and Martin Savelsbergh. "Dynamic Discretization Discovery for Solving the Time-Dependent Traveling Salesman Problem with Time Windows". In: *Transportation Science* 54.3 (May 2020), pp. 703–720. ISSN: 0041-1655. DOI: 10.1287/trsc.2019.0911 (cit. on pp. 2, 23–25, 27, 29, 93, 113–115, 119).

[Wil64] John W. J. Williams. "Algorithm 232: Heapsort". In: *Communications of the ACM* 7.6 (1964), pp. 347–348 (cit. on p. 52).

[YMI00] Takayuki Yoshizumi, Teruhisa Miura, and Toru Ishida. "A* with Partial Expansion for Large Branching Factor Problems". In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*. Ed. by Henry A Kautz and Bruce W Porter. AAAI Press / The MIT Press, 2000, pp. 923–929 (cit. on p. 55).

[ZH02] Rong Zhou and Eric A. Hansen. "Multiple sequence alignment using anytime A*". In: *Proceedings of the National Conference on Artificial Intelligence* (2002), pp. 975–976 (cit. on pp. 58, 59, 71, 77).

[ZH05] Rong Zhou and Eric A. Hansen. "Beam-stack search: integrating backtracking with beam search". In: *Proceedings of the Fifteenth International Conference on International Conference on Automated Planning and Scheduling*. 2005, pp. 90–98. DOI: 10.1.1.71.4147 (cit. on pp. 58, 60).

[Zha98] Weixiong Zhang. "Complete Anytime Beam Search". In: *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*. AAAI '98/IAAI '98. USA: American Association for Artificial Intelligence, 1998, pp. 425–430. ISBN: 0262510987 (cit. on p. 72).

FOLIO ADMINISTRATIF

THÈSE DE L'INSA LYON, MEMBRE DE L'UNIVERSITÉ DE LYON

Nom : **FONTAINE** Date de soutenance : **09/07/2024**

Prénoms : **Romain**

Titre : **Exact and anytime heuristic search for the Time Dependent Traveling Salesman Problem with Time Windows**

Nature : **Doctorat** Numéro d'ordre : **2024ISAL0067**

École Doctorale : **Informatique et Mathématiques**

Spécialité : **Informatique**

Résumé :

Le problème du voyageur de commerce (TSP, pour *Traveling Salesman Problem*) dépendant du temps (TD, pour *Time Dependent*) est une généralisation du TSP qui permet de prendre en compte les conditions de trafic lors de la planification de tournées en milieu urbain : les temps de trajet varient en fonction des horaires de départ au lieu d'être constants. Le TD-TSPTW généralise ce problème en associant à chaque point de passage une fenêtre temporelle (TW, pour *Time Window*) qui restreint les horaires de visite. Les approches de résolution exactes telles que la programmation linéaire en nombres entiers ou la programmation dynamique passent mal à l'échelle, tandis que les approches heuristiques ne garantissent pas la qualité des solutions obtenues.

Dans cette thèse, nous proposons une nouvelle approche exacte et anytime pour le TD-TSPTW visant à obtenir rapidement des solutions approchées puis à les améliorer progressivement jusqu'à prouver leur optimalité. Nous montrons d'abord comment rapporter le TD-TSPTW à une recherche de meilleur chemin dans un graphe états-transitions. Nous décrivons ensuite des algorithmes permettant de résoudre ce problème en nous concentrant sur les extensions exactes et anytime d'A*, et en proposons une nouvelle par hybridation. Nous montrons comment combiner ces algorithmes avec de la recherche locale – afin de trouver plus rapidement de meilleures solutions – ainsi qu'avec des bornes et de la propagation de contraintes de TW – afin de réduire la taille de l'espace de recherche. Enfin, nous fournissons des résultats expérimentaux visant à (i) valider nos principaux choix de conception, (ii) comparer notre approche à l'état de l'art en considérant des benchmarks ayant différents degrés de réalisme et différentes granularités temporelles et (iii) comparer ces approches TD à de récents solveurs pour le TSPTW dans le cas constant. Ces résultats montrent que notre approche apporte un bon compromis entre le temps nécessaire pour (i) trouver de bonnes solutions et (ii) trouver des solutions optimales et prouver leur optimalité, aussi bien dans le cas TD que dans le cas constant.

Mots-clés : Optimisation Combinatoire, Problème du Voyageur de Commerce, Contraintes de fenêtres temporelles, Temps de trajets Time-Dependent, Programmation Dynamique, Algorithmes de recherche Exacts et Anytime, Algorithme A*, Passage à l'échelle

Laboratoire(s) de recherche : **CITI (INRIA – INSA Lyon)**

Directeur de thèse : **Christine SOLNON, Jilles S. DIBANGOYE**

Président du Jury : **Romain BILLOT**

Composition du Jury :

Cédric PRALET, Pierre SCHAUS, Romain BILLOT, Christine SOLNON, Jilles S. DIBANGOYE