A Constraint Programming Approach for the Time Dependent Traveling Salesman Problem

Penelope Aguiar-Melgarejo

▶ To cite this version:

Penelope Aguiar-Melgarejo. A Constraint Programming Approach for the Time Dependent Traveling Salesman Problem. Computer Science [cs]. INSA Lyon, 2016. English. ⟨NNT : 2016LYSEI142⟩. ⟨hal-01514369⟩

HAL Id: hal-01514369

https://hal.science/hal-01514369v1

Submitted on 20 Jun 2018

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

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

HAL Authorization

N°d'ordre NNT : 2016LYSEI142

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée au sein de
INSA Lyon

Ecole Doctorale N° EDA512
Informatique et Mathématiques de Lyon

Spécialité/discipline de doctorat : Informatique

Soutenue publiquement le 16/12/2016, par :
Penélope AGUIAR MELGAREJO

A Constraint Programming Approach for the Time Dependent Traveling Salesman Problem

---

Devant le jury composé de :

EL-FAOUZI, Nour-Eddin Directeur de Recherche IFSTTAR Président

DEVILLE, Yves Professeur des Universités Université Catholique de Louvain Rapporteur
HUGUET, Marie-José Professeur des Universités INSA de Toulouse Rapporteur
CAMBAZARD, Hadrien Maître de Conférences INP Grenoble Examinateur

SOLNON, Christine Professeur INSA Lyon Directrice de thèse
LABORIE, Philippe Docteur IBM Co-directeur de thèse

Département FEDORA – INSA Lyon - Ecoles Doctorales – Quinquennal 2016-2020

SIGLE	ECOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
CHIMIE	**CHIMIE DE LYON** **http://www.edchimie-lyon.fr** Sec : Renée EL MELHEM Bat Blaise Pascal 3e etage secretariat@edchimie-lyon.fr Insa : R. GOURDON	M. Stéphane DANIELE Institut de Recherches sur la Catalyse et l'Environnement de Lyon IRCELYON-UMR 5256 Équipe CDFA 2 avenue Albert Einstein 69626 Villeurbanne cedex directeur@edchimie-lyon.fr
E.E.A.	**ELECTRONIQUE, ELECTROTECHNIQUE, AUTOMATIQUE** **http://edeea.ec-lyon.fr** Sec : M.C. HAVGOUDOUKIAN Ecole-Doctorale.eea@ec-lyon.fr	M. Gérard SCORLETTI Ecole Centrale de Lyon 36 avenue Guy de Collongue 69134 ECULLY Tél : 04.72.18 60.97 Fax : 04 78 43 37 17 Gerard.scorletti@ec-lyon.fr
E2M2	**EVOLUTION, ECOSYSTEME, MICROBIOLOGIE, MODELISATION** **http://e2m2.universite-lyon.fr** Sec : Safia AIT CHALAL Bat Darwin - UCB Lyon 1 04.72.43.28.91 Insa : H. CHARLES Safia.ait-chalal@univ-lyon1.fr	Mme Gudrun BORNETTE CNRS UMR 5023 LEHNA Université Claude Bernard Lyon 1 Bât Forel 43 bd du 11 novembre 1918 69622 VILLEURBANNE Cédex Tél : 06.07.53.89.13 e2m2@ univ-lyon1.fr
EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTE** **http://www.ediss-lyon.fr** Sec : Safia AIT CHALAL Hôpital Louis Pradel - Bron 04 72 68 49 09 Insa : M. LAGARDE Safia.ait-chalal@univ-lyon1.fr	Mme Emmanuelle CANET-SOULAS INSERM U1060, CarMeN lab, Univ. Lyon 1 Bâtiment IMBL 11 avenue Jean Capelle INSA de Lyon 696621 Villeurbanne Tél : 04.72.68.49.09 Fax :04 72 68 49 16 Emmanuelle.canet@univ-lyon1.fr
INFOMATHS	**INFORMATIQUE ET MATHEMATIQUES** **http://infomaths.univ-lyon1.fr** Sec :Renée EL MELHEM Bat Blaise Pascal 3e etage infomaths@univ-lyon1.fr	Mme Sylvie CALABRETTO LIRIS – INSA de Lyon Bat Blaise Pascal 7 avenue Jean Capelle 69622 VILLEURBANNE Cedex Tél : 04.72. 43. 80. 46 Fax 04 72 43 16 87 Sylvie.calabretto@insa-lyon.fr
Matériaux	**MATERIAUX DE LYON** **http://ed34.universite-lyon.fr** Sec : M. LABOUNE PM : 71.70 –Fax : 87.12 Bat. Saint Exupéry Ed.materiaux@insa-lyon.fr	M. Jean-Yves BUFFIERE INSA de Lyon MATEIS Bâtiment Saint Exupéry 7 avenue Jean Capelle 69621 VILLEURBANNE Cedex Tél : 04.72.43 71.70 Fax 04 72 43 85 28 Ed.materiaux@insa-lyon.fr
MEGA	**MECANIQUE, ENERGETIQUE, GENIE CIVIL, ACOUSTIQUE** **http://mega.universite-lyon.fr** Sec : M. LABOUNE PM : 71.70 –Fax : 87.12 Bat. Saint Exupéry mega@insa-lyon.fr	M. Philippe BOISSE INSA de Lyon Laboratoire LAMCOS Bâtiment Jacquard 25 bis avenue Jean Capelle 69621 VILLEURBANNE Cedex Tél : 04.72 .43.71.70 Fax : 04 72 43 72 37 Philippe.boisse@insa-lyon.fr
ScSo	**ScSo*** **http://recherche.univ-lyon2.fr/scso/** Sec : Viviane POLSINELLI Brigitte DUBOIS Insa : J.Y. TOUSSAINT viviane.polsinelli@univ-lyon2.fr	Mme Isabelle VON BUELTZINGLOEWEN Université Lyon 2 86 rue Pasteur 69365 LYON Cedex 07 Tél : 04.78.77.23.86 Fax : 04.37.28.04.48

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

Résumé

L'optimisation des tournées de livraison est souvent modélisée par un problème de voyageur de commerce (Traveling Salesman Problem / TSP). Pour ce problème, il est fréquent d'avoir des contraintes additionnelles telles que, par exemple, des fenêtres horaires limitant les heures de livraison chez le client ou des pauses obligatoires pour les conducteurs des camions. Le temps est une dimension importante à prendre en compte pour respecter ces contraintes. Cependant, les durées des trajets ne sont généralement pas constantes mais varient en fonction des congestions, et cette variabilité doit être intégrée au moment de l'optimisation des tournées. Ainsi, le problème du voyageur de commerce dépendant du temps (Time Dependent TSP / TD-TSP) est la version étendue du TSP où le coût d'un arc dépend de l'heure à laquelle cet arc est emprunté.

Dans cette thèse nous proposons un nouveau benchmark pour le TDTSP basé sur des données réelles de trafic (fournies par la Métropole de Lyon) et nous montrons l'intérêt de prendre en compte la variabilité des durées dans ce problème. Nous étudions comment mieux modéliser les fonctions de durée de trajet dépendantes du temps. Nous introduisons et comparons différents modèles pour résoudre le TDTSP avec la programmation par contraintes (Constraint Programming / CP). Un premier modèle est directement dérivé du modèle CP classique pour le TSP. Nous montrons que ce modèle ne permet pas de raisonner avec des relations de précédence indirectes, ce qui pénalise sa performance sur notre benchmark. Nous introduisons une nouvelle contrainte globale qui est capable d'exploiter des relations de précédence indirectes sur des données dépendantes du temps et nous introduisons un nouveau modèle CP basé sur notre nouvelle contrainte. Nous comparons expérimentalement les deux modèles sur notre benchmark, et nous montrons que notre nouvelle contrainte permet de résoudre le TDTSP plus efficacement.

Abstract

In the context of urban deliveries, the optimization of delivery tours is usually modeled as a Traveling Salesman Problem (TSP). Side constraints like time-windows constraining the delivery times at the client or breaks for the drivers are also common in this kind of problem and time is an important dimension to take into account to respect these constraints. With travel times' variability in big cities time also tends to have a greater influence in costs and therefore it should be included in the optimization of delivery routes. The Time-Dependent Traveling Salesman Problem (TDTSP) is the extended version of the Traveling Salesman Problem (TSP) where arc costs depend on the time when the arc is traveled.

In this thesis we propose a set of benchmarks for the TDTSP based on real traffic data (obtained from the city of Lyon) and show the interest of handling time dependency in the problem. A study of how to better model time-dependent travel functions in general and specifically for our approach is performed. We introduce and compare different models to solve the TDTSP with Constraint Programming (CP). A first model is derived in a straightforward way from the classical CP model for the TSP. We show that this model is not able to reason on indirect precedence relations, so that it has poor performance on our benchmark. We introduce a new global constraint which is able to exploit indirect precedence relations on time-dependent data, and we introduce a second model which is based on our new constraint. We experimentally compare the two models on our benchmark and show that the new model is more efficient.

Acknowledgements

First, I would like to thank Patrick Albert and specially Thomas Baudel, from IBM, who were responsible for making this PhD possible. Thanks to Thomas I had data for my tests and all the framework necessary to use it, he has also been a mentor and helped me a lot in the first half of the PhD. I spent most of my PhD working from IBM, sharing an office with Philippe Laborie, Jérôme Rogerie and Nicolas Bonifas, it was a big pleasure to share an office with them and I will always have nice memories from those days. I would also like to thank everybody at IBM that helped me or made my days a little sweeter.

My most special thanks go to my advisors who followed me and guided me along this path, Christine Solnon and Philippe Laborie. Seeing Philippe working with so much focus everyday was always a great inspiration. Christine also always impressed me, I always wondered how she can take care of so many things at the same time without going crazy! I'll always look up to them as some of the best examples to follow, in their kindness, dutifulness, humbleness and care for quality. I consider myself very lucky for having them as advisors.

I'm grateful for all the friends that supported me along this journey, to all the friends that I made along the way, to all the people that followed my work and gave me feedback. If I had a better memory (and more time) I would put all the names here but nevertheless I expect them to know that I am grateful. Thanks to my parents, without whom I would not be here today writing these words. They were my inspiration to pursue a PhD and the stronger supporters of this pursuit, helping with everything material and immaterial I needed to move to France.

In the last year of this amazing adventure I met the love of my life, Frode Johansen, it was both distracting and helpful concerning my PhD. It gave me direction and a good reason to move on, it also gave me roots when all I wanted was to fly away and explore the world. I am happy to have found someone to explore life with. I thank him so much for all his support, love and patience during these final (very) stressful months.

Those have been three years of a lot of learning, not just academical learning but above all learning about life. I wouldn't change a thing, those three years have been wonderful in so many ways. One of the greatest responsible for this is Nicolas Bonifas, I will never be able to thank him enough for all the mutual support, conversations, lessons, laughs, vegetarian restaurants, walks, runs, trainings, sharing, travels, challenges, crazy experiments. He is probably the biggest gift I got from this PhD, a friend like this is priceless. I hope life will bring us together again one day.

Contents

1 Introduction **8**
1.1 Thesis plan 10
1.2 List of publications 12

2 Background **14**
2.1 TSP 14
2.2 Introduction to Constraint Programming 17
2.2.1 Constraint Satisfaction Problem 18
2.2.2 Example of modeling language: OPL 19
2.2.3 Filtering and search 20
2.2.4 Global constraints 23
2.2.5 Constrained Optimization Problem 24
2.3 CP for the TSP 25
2.4 Scheduling and CP Optimizer 26
2.4.1 Precedence graph 27
2.4.2 Search 29
2.5 Discussion 30

3 TDTSP(TW) **32**
3.1 Definition of the problem 32
3.2 Classification of routing problems according to time-dependency 36
3.3 Literature review 38
3.3.1 Motivation: constant versus time-dependent approaches 38
3.3.2 Methods : existing approaches for solving the TDTSP and TDVRP 40
3.3.3 Experiments: instances generation and characteristics . 42
3.4 Discussion 44

4 A new benchmark for the TDTSP derived from real-world data **46**
4.1 Estimation of travel speeds 47

4.2 From road-network to instance graph 48
4.2.1 Estimating time-dependent travel times from time-dependent speeds 49
4.2.2 Time-dependent shortest paths calculation 53
4.2.3 Generation of dilated travel time functions 55
4.3 Benchmark instances generation 56
4.4 Discussion 59

5 Modeling time-dependent travel time functions 60
5.1 The FIFO property 60
5.1.1 Travel times or travel speeds? 61
5.1.2 Properties of FIFO functions 63
5.1.3 FIFO transformation 64
5.2 The triangular inequality property 67
5.3 Discussion 69

6 Modeling the TDTSP with existing CP concepts 71
6.1 Extension of the classical CP model for the TSP 72
6.2 Limitations 74
6.3 A scheduling model for the (TD)TSP 77
6.4 Discussion 79

7 The *TDNoOverlap* constraint 80
7.1 CP model with *TDNoOverlap* 80
7.2 Propagation of *TDNoOverlap* 81
7.2.1 Computation of $f^{next}_{earliest}$ 82
7.2.2 Computation of f^{next}_{latest} 83
7.2.3 Computation of $f^{succ}_{earliest}$ 84
7.2.4 Computation of f^{succ}_{latest} 85
7.2.5 Time-dependent disjunctive propagation 85
7.3 Implementation and complexity 85

8 Experimental evaluation 88
8.1 Comparison of TDTSP and TSP solutions 88
8.2 Experimental setup 90
8.3 Evaluation of the effects of taking time-dependency into account 93
8.4 Scaling of the different models 95
8.4.1 `NoOverlap-TDTSP` versus `TDNoOverlap-TDTSP` 95
8.4.2 `Classic-TDTSP` versus `TDNoOverlap-TDTSP` 97
8.4.3 Comparison of convergence of `NoOverlap-TSP` and `TDNoOverlap-TDTSP` 102
8.5 Tests on other instances 106

9 Conclusion **110**

A Backward IGP algorithm **113**

Chapter 1

Introduction

We live in a time of big urban centers and big data. We have more information collected than ever before and, as a consequence, a harder time processing all this data into something useful. This is exactly the aim of the Optimod'Lyon project: leveraging data collected from the city in order to improve traffic conditions and therefore reduce travel times for users of the transportation network as well as carbon emissions. This thesis comes as a part of one sub-project of Optimod'Lyon called Smart Deliveries.

Optimod'Lyon (http://optimodlyon.fr/en/) is a three year project supported by the French agency for the environment (ADEME), in which eight industry partners, among which IBM, four academic institutions, among which LIRIS, and the metropolitan authority of Lyon (called Métropole de Lyon) work together to improve urban mobility through better collection, processing and distribution of mobility information. The flagship application of Optimod'Lyon is a real-time, predictive, multi-modal journey planner. This application lets travelers plan their journey through a combination of car, public transportation, walk and bike sharing with the best available information. The expectation is to enable a reduction of 1% of the car modal share on the city, resulting in saving 25 kTe of carbon emissions per year and significant traffic improvements during rush hours, at a very reasonable cost compared to the amount of public works that are usually needed to obtain similar levels of decongestion.

Despite environmental concerns, there is a large portion of road traffic that will not lend itself easily to modal transfer. According to a french report about goods transportation [31], 35% of road trips carry goods rather than people. If we account for the important portion of movements that involve moving both goods and persons to deliver a value added service (such as maintenance and construction visits, catering, sales visits), an even larger portion of existing road traffic needs to be accounted for. We call this portion

of the traffic we focus on "professional urban mobility".

Businesses are also acutely aware and affected by the cost of mobility and have direct incentives to improve it. For instance, fuel represents 18% of the total costs of a small freight operator employing 200 persons (undisclosed personal communication). Current estimates (CERB, 2007) consider that 20 minutes lost in urban traffic amount to about eight euros for such an operator, which represents, by-and-large, its gross financial margin for a half-day vehicle tour (also called vehicle round). We can therefore consider there is a synergy of goals between public authorities and the private businesses in curbing congestion and optimizing trips. The shared objective of both public and private stakeholders is to tend towards the minimization of vehicle-kilometers and time spent on the road. Smart Deliveries stems from the observation that a large portion of professional mobility demand is actually planned and can, therefore, be optimized leveraging the traffic information obtained from the city.

Smart Deliveries targets planned urban mobility executed as a **tour** - a sequence of places to be visited, often (but not necessarily) departing and returning to the same address, corresponding to the depot. As the name says, this service is aimed for delivery's companies even though any planned moves involving a sequence of visits could use it as well. The theoretical problem modeling the real-world situation of optimizing a delivery tour is the Traveling Salesman Problem (TSP), where one has to find the minimal total distance to travel through a list of cities (going through each city exactly once). The TSP deals with constant distances (or travel times) and is generally a poor representation of reality when travel times vary considerably during the day, as it does in urban areas. For this reason, in this thesis we study the Time-Dependent Traveling Salesman Problem (TDTSP), which is an extension of the TSP that takes into account variations of travel times during the day.

In order to better address the optimization demands of the possible users of Smart Delivers it is necessary to be able to take different kinds of side-constraints into account. Therefore, it is important to develop a solving method flexible enough to integrate other constraints in the main model without hindering performances and ideally in a simple way. This motivated our choice to tackle the problem using Constraint Programming. Some common constraints in the business of deliveries are: time-windows when clients are available for delivery (or other services), regulations concerning break times for drivers, precedence between deliveries (as in the case of pick-up and delivery problems). The most common and important of all probably being the time-window constraint. When this constraint is taken into consideration the problem is called TDTSP with Time-Windows (TDTSPTW).

Since the availability of extensive real-world data in the area of urban transportation is quite recent, the TDTSP has not been much studied in the literature and Constraint Programming (CP) approaches are even rarer. One reason for this is that CP is usually less efficient than Integer Linear Programming or Meta-heuristics (Local Search, Evolutionary Algorithms, Ant Colonies) for pure (non time-dependent) vehicle routing problems. On the other hand, Constraint-Based Scheduling [10], that is the application of CP to scheduling problems, is one of the biggest industrial success of CP and has shown that CP technologies can be very efficient for solving temporal problems. A variety of specialized variable types (interval variables, sequence variables) and related global constraints and search algorithms have been developed until recently [56, 57, 58] to improve the expressiveness and efficiency of CP-based models involving temporal domains. Other than that, CP is also a flexible framework, ideal for taking the side-constraints commonly present in this context into account in an efficient manner.

Thesis contributions In this context, the main contributions of the PhD thesis are:

- The generation of a new benchmark for the TDTSP that is based on real-world data and made available online
- An in-depth analysis about the modeling of time-dependent travel time functions
- The evaluation of the interests of taking time-dependent travel times into account in the optimization of (delivery) tours in an urban context
- The evaluation of CP as a suitable approach to tackle the TDTSP, the development of a new global constraint and propagation algorithm to model and solve the TDTSP

1.1 Thesis plan

In chapter 2 we give the necessary background to understand the rest of the work presented here. We first introduce the TSP - as a building block to tackle the TDTSP later on - and define some useful concepts to model it, as well as giving an overview of the state-of-the-art approaches to solve it. We then give an introduction to CP, briefly review CP approaches for the TSP and shortly introduce CP Optimizer.

In chapter 3 we define formally the TDTSP(TW), classify vehicle routing problems according to time-dependency and review the literature in three phases (corresponding to motivation, methods and experiments): (1) we present of studies performed in the literature in order to motivate the modeling of real world delivery problems through TDTSPs instead of its constant time version, the TSP (2) we list solving approaches for the TDTSP as well as to the more general Time-Dependent Vehicle Routing Problem (TDVRP) - where visits have to be attributed to a whole fleet of vehicles instead of just one as in the TDTSP and (3) we overview the instances commonly used for testing performances of different solving techniques for the TDTSP and TDVRP (with Time-Windows), how they are generated (randomly or from real-world data) and their main characteristics.

Next, in chapter 4 we move from theory to application and describe what are the steps to transform the real-world problem (data) in instances that serve as input for optimization models. We then describe how we generated our set of benchmark instances.

In the chapter that follows, 5, we delve deeper into the modeling of travel time functions and give some new algorithms to calculate alternate versions of those functions respecting certain properties that are useful in our solving technique.

Then, in chapter 6, the modeling of the TDTSP with CP is studied. A first model, extending the CP model previously presented (in the background chapter 2) for the TSP, is given and its limitations are pointed out through an example. A second model solving some of the issues of the first model is given as well as its performance on the same example. The limitation of this second model gives rise to the development of a dedicated global constraint, which is presented in details in the subsequent chapter.

In chapter 7, the model using the new constraint (`TDNoOverlap-TDTSP`) is presented, followed by a description of the internal propagation of the constraint. By the end of the chapter, implementation and complexity of the constraint are discussed.

In the sequence, in chapter 8, we present an experimental study that aims at answering two questions: (1) how does the quality of solutions found with a TSP model and with a TDTSP model compare, i.e., what is the effect of integrating time-dependency in the optimization model? (2) how do the different CP models presented in this thesis compare in terms of performance and how do they scale-up when the size of instances grows? We end this chapter with a description of other instances considered for testing our method and the work that has been done in this direction. Final conclusions and future perspectives are discussed in the last chapter.

1.2 List of publications

National Journal

2013 - Optimisation du fret et des déplacements professionnels planifiés dans le projet OptimodLyon. T. Baudel, P. Aguiar Melgarejo, C. Solnon, L. Jacques, J. Coldefy. Revue TEC 219, pp. 2-5.

International Conference

2015 - A Time-Dependent No-Overlap Constraint: Application to Urban Delivery Problems. P. Aguiar Melgarejo, P. Laborie, C. Solnon. Twelfth International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming (CPAIOR 2015)

International Workshops or Conferences that select papers from abstracts

2015 - A Constraint Programming Approach for the Time-Dependent Traveling Salesman Problem with Time-Windows. P. Aguiar Melgarejo, P. Laborie, C. Solnon. 27th European Conference on Operational Research (EURO 2015).

2015 - Optimizing Urban Freight Deliveries: From Designing and Testing a Prototype System to Adressing Real Life Challenges. T. Baudel, L Dablanc, J. Ashton, P. Aguiar Melgarejo. 9th International Conference on City Logistics (City Logistics 2015)

2014 - A Time-Dependent No-Overlap Constraint: Application to Delivery Problems. P. Aguiar Melgarejo, P. Laborie, C. Solnon. Doctoral Program of the 20th International Conference on Principles and Practice of Constraint Programming (CP 2014) , pp. 1-7.

2012 - Global and reactive routing in urban context: first experiments and difficulty assessment. P. Aguiar Melgarejo, T. Baudel, C. Solnon. Workshop on Optimization and Smart Cities.

National Workshops or Conferences that select papers from abstracts

2015 - A Time-Dependent No-Overlap Constraint: Application to Delivery Problems. P. Aguiar Melgarejo, P. Laborie, C. Solnon. 16ème confrence ROADEF Société Franaise de Recherche Oprationnelle et Aide à la Décision (ROADEF 2015)

2015 - Une contrainte de No-Overlap dépendant du temps : Application aux problèmes de tournées de livraison en milieu urbain. P. Aguiar Melgarejo, P. Laborie, C. Solnon. Onzièmes Journées Francophones de Programmation par Contraintes (JFPC 2015).

Chapter 2

Background

The first part of this chapter gives the necessary background to understand the mathematical problem addressed in this thesis, the TDTSP, which is an extension of the TSP. Therefore, in the first section (2.1), the TSP and preliminary graph theory concepts are formally defined. The TDTSP is described in the following chapter.

The second part of the chapter concerns our approach to solve the TDTSP, Constraint Programming (CP). In section 2.2, we give an introduction to Constraint Programming and in section 2.3, CP approaches for the TSP are briefly presented. In the last section (2.4), the CP solver used in this thesis (CP Optimizer) is described in what concerns modeling and search, in the context of scheduling problems.

2.1 TSP

The Traveling Salesman Problem (TSP) is an NP-hard problem and one of the most studied problems in combinatorial optimization as it appears as a substructure in many problems but also has several applications in its pure form. The TSP can be stated as the problem of finding the shortest way in which to travel through a list of cities and back to the starting point without going through any of the cities (other than the departing one) twice. Nowadays, the most common applications of the problem are in the areas of transportation, planning and logistics but it has also been used for DNA sequencing [62] and chips manufacturing [54], for example. The TSP is a specific case of the more general Vehicle Routing Problem (VRP), where a whole fleet of vehicles has to be routed in an optimal way. In the following we give some preliminary definitions in order to formally define the TSP and VRP.

Definition 1 (Graph) *A graph G is defined by a couple (V, A) such that V is a finite set of vertices, and $A \subseteq V \times V$ is a set of arcs. If there are arcs between all pairs of vertices (i.e., $A = V \times V$) then the graph is said complete. In this thesis, we consider directed graphs such that arcs are oriented, i.e., arc $(i, j) \in A$ is different from arc (j, i), and we call them graphs for short.*

Let $G = (V, A, c)$ be a complete graph, to each arc $(i, j) \in A$ such that $i \neq j$, is associated a cost function $c : A \rightarrow \mathbb{R}^+$. This function may represent an actual cost (for example, a transportation cost), the time or distance necessary to go from i to j, or a combination of different costs.

Definition 2 (Path) *A path in $G = (V, A)$ is a sequence of vertices $P = (v_1, ..., v_k)$ such that*
$(v_i, v_{i+1}) \in A, \forall\ i \in \{1, ..., k-1\}$.

Definition 3 (Cost of a path) *Given a graph $G = (V, A, c)$, the cost $C(P)$ of a path $P = (v_1, ..., v_k)$ on graph G is defined as the sum of the costs of the arcs in the path:*

$$C(P) = \sum_{i=1}^{k-1} c(v_i, v_{i+1}) \tag{2.1}$$

Definition 4 (Hamiltonian path) *A Hamiltonian path is a path where every vertex of the graph appears exactly once.*

Definition 5 ((Hamiltonian) cycle) *A cycle in a graph is a closed path, a path where the last vertex is the same as the first. A Hamiltonian cycle is a cycle where every vertex of the graph appears exactly once except for the first vertex, which is also the last.*

Definition 6 ((A)TSP) *The TSP is the problem of finding a Hamiltonian cycle in $G = (V, A, c)$ of minimum cost (as in definition 3). We call it asymmetric TSP (ATSP) when G has asymmetric costs, i.e., when $c(i, j) \neq c(j, i)$ for some $i, j \in V$.*

The ATSP is actually more realistic from a transportation point of view where distances (roads) and travel times are rarely symmetric (Fig. 2.2). A generalization of the TSP is the VRP, where multiple vehicle tours are optimized at once, as in Definition 7.

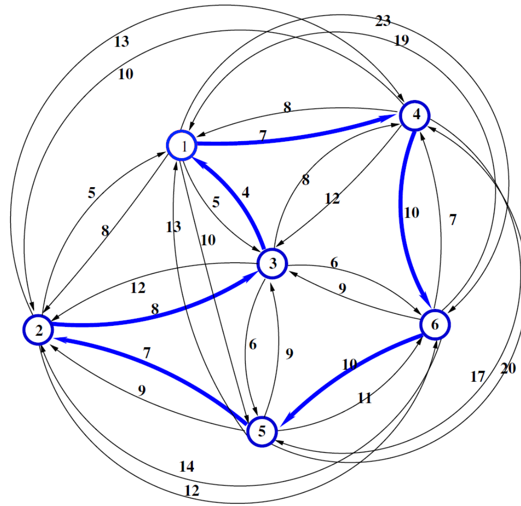


Figure 2.1: Example of an optimal solution to a TSP highlighted in blue

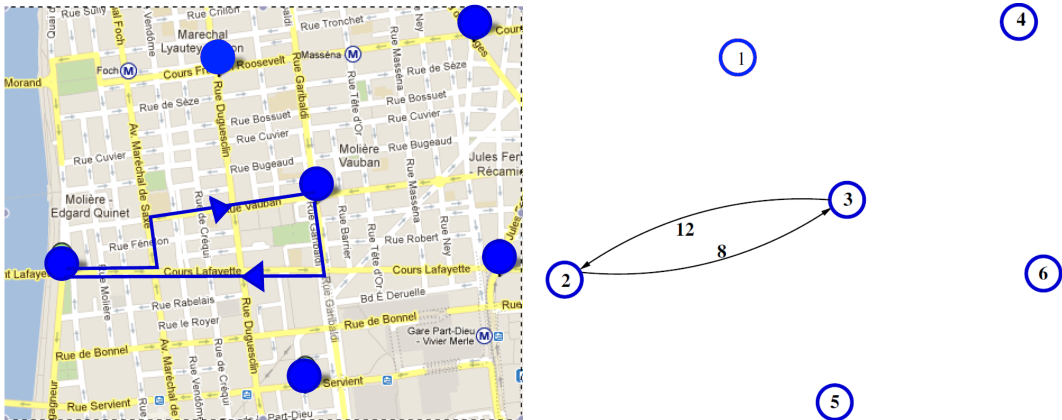


Figure 2.2: Example of asymmetry in the road-network, on the left, and in the corresponding graph representation (with edge costs equivalent to the length of the paths), on the right

Definition 7 (VRP) *Given a graph $G(V, A, c)$, a depot d in V, a number $m \geq 1$ of vehicles and a cost function $c : A \rightarrow \mathbb{R}^+$, the VRP is the problem of finding m cycles such that*

- *each cycle starts and ends at the depot d*
- *each vertex v in $V \backslash \{d\}$ occurs exactly once in one cycle*
- *the sum of the m cycle costs (as in definition 3) is minimum.*

In the TSP, a solution for the problem is often called tour or route. For the VRP, one solution is made out of multiple tours or routes. The number of vehicles m is not always fixed in the VRP and this number is sometimes minimized as a first objective (before optimizing route costs). Nodes of the graph are often clients with demands and vehicles frequently have capacity constraints (Capacitated-VRP).

State-of-the-art exact approaches for solving the TSP are based on Integer Linear Programming (ILP)[8], the most notable example being the special-purpose TSP solver Concorde [25], that can solve pure symmetric TSPs on 10,000 or more vertices in a reasonable time (the largest to date having 85,900 cities). Nevertheless, this approach is not capable of handling asymmetric instances or instances containing other constraints (side-constraints), common in real-life problems. This limitation comes from the fact that Concorde works by adding cuts and upper bound heuristics specific to the (symmetric) TSP, which no longer work when side-constraints are added. Current exact methods able to solve the ATSP with constraints cannot scale beyond a few hundred vertices.

In the next section we give an introduction to Constraint Programming and how it has been applied to solve the ATSP so far.

2.2 Introduction to Constraint Programming

Constraint programming (CP) provides high level languages that allow one to describe (model) a problem in a declarative way by means of constraints, that is, properties of the solution to be found. The user might as well have to specify how the solver should search for solutions but some solvers have automatic search available. The problem is then automatically solved by embedded algorithms.

In this section we start by explaining what is a Constraint Satisfaction Problem (CSP), in subsection 2.2.1. In order to solve CSPs with constraint programming two main parts have to be considered, modeling and search. An example of modeling language (OPL) is given in subsection 2.2.2 and search and filtering are presented in 2.2.3. In 2.2.4, the concept of global constraints is introduced. Global constraints have as goal to ease the modeling of a CSP and/or improve the search for a solution in CP. And finally, in subsection 2.2.5, Constrained Optimization Problems are introduced. For this thesis, the OPL language and C++ were used to model COPs, and IBM CP Optimizer (CPO) was used to solve them.

2.2.1 Constraint Satisfaction Problem

A constraint satisfaction problem (CSP) is described by a finite set of variables $x \in X$ ranging over specific domains $D(x)$ and satisfying a finite number of constraints $c \in C$. Variables usually range over integer, real, binary or non-numeric domains.

A constraint $c(x_1, ..., x_k)$ is a relation on the domains of the variables involved, i.e., $c(x_1, ..., x_k) \subseteq D(x_1) \times ... \times D(x_k)$. It can be viewed as a requirement that states which combinations of values from the variable domains are admitted. The number k of variables involved is the arity of the constraint, a constraint on two variables is called binary and, more generally, a constraint on n variables is called n-ary.

Constraints can be defined in **intension**, when they can be described by a predicate. For example, $x_1 \neq x_2$ or $|x_1 * x_2| < |x_3|$. Or they can be defined in **extension**, when we enumerate explicitly what tuples satisfy it or not. If $D(x_1) = \{1, 2, 3, 4\}$ and $D(x_2) = \{3\}$ expressing $x_1 < x_2$ in extension means defining it by $\{(1, 3), (2, 3)\}$.

Given a set of variables $X' \subseteq X$, an **assignment** or **instantiation** of X' is a tuple $\tau \in \Pi_{x \in X} D(x)$. It is complete if $X' = X$ and partial otherwise. Given a subset of variables $X'' \subseteq X$, and a variable $x \in X''$, we note $\tau|_{X''}$ the restriction of τ to variables of X'', and τ_x the value of x in τ. Given an assignment τ of $X' \subseteq X$ and a constraint c defined on a set of variables $X'' \subseteq X'$, τ satisfies c if $\tau|_{X''} \in c$, and τ violates c otherwise.

A **solution** to the problem is a complete instantiation such that all constraints in C are satisfied simultaneously.

The same problem can usually be modeled by different CSPs. We illustrate this fact on the n-queens problem, the problem of placing n queens onto a $n \times n$ chessboard in a way that queens cannot attack each other. A solution for this problem with $n = 8$ queens is given in figure 2.3.

One way of modeling this problem in CP is to define two types of variables, col_i and row_i for all $i \in \{1, ..., n\}$, specifying the column and row of queen i thus varying in the domain $D = \{1, ..., n\}$. The following constraints are defined:

- $col_i \neq col_j \ \forall \ i, j \in D | \ i \neq j$ (columns must be different)
- $row_i \neq row_j \ \forall \ i, j \in D | \ i \neq j$ (rows must be different)
- $row_i + col_i \neq row_j + col_j$ and $row_i - col_i \neq row_j - col_j \ \forall \ i, j \in D | \ i \neq j$ (diagonals must be different)

This first model has $2n$ variables.

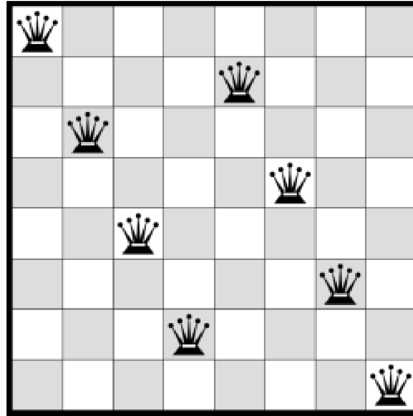


Figure 2.3: A solution of the 8-queens problem

Since there must be one and only one queen per column or row we can eliminate one of the two sets of variables, let us say columns, by supposing that queen k is in column k, i.e. row_k gives the row of queen in column k. The model then becomes:

- $row_i \neq row_j \; \forall \, i, j \in D | \; i \neq j$ (rows must be different)
- $row_i + i \neq row_j + j$ and $row_i - i \neq row_j - j \; \forall \, i, j \in D | \; i \neq j$ (diagonals must be different)

The second model has the advantage of having less variables (with the same domains as in the previous model) and less constraints. The search space of a CSP is the set of all complete assignments $\Pi_{x \in X'} D(x)$. For example, the search space of the first model for the n-queens problem contains n^{2n} tuples, whereas the search space of the second model contains n^n tuples.

The n-queens problem illustrates how a single CSP can be modeled in different ways and how this affects the effectiveness in which it will be solved. By reducing the search space, as we saw in the example, or by allowing a more (or less) efficient propagation of constraints, as we explain in Section 2.2.3.

2.2.2 Example of modeling language: OPL

The Optimization Programming Language (OPL) is a modeling language that supports both mathematical and constraint programming, it uses IBM ILOG CPLEX Optimizer to solve mathematical programming models and IBM ILOG CP Optimizer to solve CP models. Similar to AMPL, OPL's syntax is close to the mathematical notation of optimization problems.

The structure of an OPL model contains the following ingredients:

- input values and constants definition
- definition of decision variables and their domains
- a function to minimize or maximize (for CP models it is optional)
- inside a block **subject to** come all the constraints
- pre and post-processing or flow control instructions can be given in **execute** blocks written in JavaScript

Let us give an example of a CP model in OPL for the n-queens problem:

```
int n=...;
range N= 1..n;
dvar int row[N] in N;

subject to {
  forall(i in N, j in N : i != j){
    row[i]!= row[j];
    row[i] + i != row[j] + j;
    row[i] - i != row[j] - j;
  }
}
```

In 2.2.4 we will see a more efficient way of expressing these constraints.

2.2.3 Filtering and search

The simplest way to search for a solution is called **generate and test** and it consists in enumerating every complete instantiation in turn and checking whether it satisfies all the constraints. To generate all complete instantiations we construct a search tree starting from an empty instantiation (no variables are instantiated) at the root. We then choose at each node in the tree a non assigned variable x_i and a value $v_i \in D(x_i)$ to instantiate thus extending the current partial instantiation.

Constraint programming proposes to solve CSPs by combining search and propagation techniques. For combinatorial problems the number of nodes to explore in the search tree is exponential so an exhaustive search without pruning is not an option, for this reason CP combines search with propagation of constraints. Through propagation the solver can save time by not

visiting some nodes where the values assigned to the variables would cause inconsistencies.

The propagation of a constraint filters the domains of its variables by removing values that cannot belong to any solution of the CSP (values inconsistent with at least one constraint). During propagation these filtering algorithms are repeatedly called until they no longer alter the domains of variables (which must happen in a finite number of calls since the domains are finite). A pioneering work for constraint propagation has been done in 1972 by Waltz for a scene drawing application [85]. Since then, many different constraint propagation algorithms have been proposed. The reader may refer to [11, 61] for more information.

Different filtering algorithms can be proposed for each type of constraint, these algorithms can differ in their ability to filter constraints or in their time complexity. The goal is to have the best ratio between time complexity and filtering efficiency and, of course, the cost of calls of the filtering algorithms at each node must be less than the time required by the search procedure to exclude the same values by testing.

A constraint $c(X')$ defined for the variables in X' is said **inconsistent** if $c(X') = \emptyset$ and consistent otherwise. A value $a \in D(x)$ is **consistent with** $c(X')$ iff $x \notin X'$ or there exists a tuple $\tau \in c(X')$ such that $\tau_x = a$. Constraint $c(X')$ is said **arc consistent** if and only if for all $a \in D(x)$, such that $x \in X'$, a is consistent with $c(X')$.

A filtering algorithm is said to establish arc consistency if it removes all the values of the variables involved in the constraint that are not arc consistent with the constraint. For example, let us consider the constraint $x + 4 = y$ with $D(x) = \{1, 3, 4, 5\}$ and $D(y) = \{4, 5, 8\}$. An arc consistent filtering must eliminate from $D(x)$ all values not corresponding to a value in $D(y)$ and vice-versa. We see that 3 and 5 are not consistent values for x because 7 and 9 are not possible values for y. In the same way we eliminate 4 from $D(y)$ because there is no 0 in $D(x)$. Hence, an arc consistent filtering for this constraint must give $D(x) = \{1, 4\}$ and $D(y) = \{5, 8\}$.

The **backtracking** algorithm tests consistency of partial instantiations by only checking constraints concerning at least one fixed variable and verifying whether or not the assigned value is consistent with the constraint. In this way the algorithm can stop the search sooner in certain branches as we see in figure 2.4.

In **forward checking**, once a variable x is fixed to v we eliminate (temporarily) all values from the non instantiated variables' domains that are inconsistent with $x = v$. For this reason we do not need to check consistency of the current variable with the previously fixed ones since they are necessarily consistent. This method allows to prune branches leading to failure

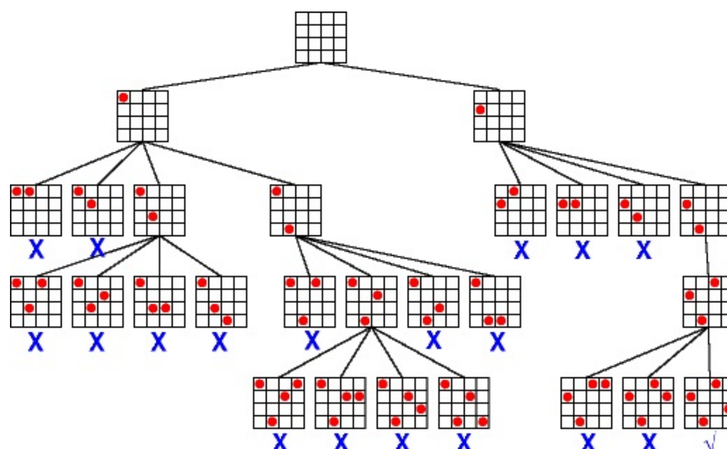


Figure 2.4: Backtracking for the second model of the 4-queens problem (figure by R. Barták [14])

earlier than backtracking as we can see in figure 2.5. This happens because in forward checking we can discard a branch as soon as the domain of a non instantiated variable becomes empty (there is no value consistent with the current partial instantiation), in backtracking this branch would be further searched until the algorithm tried to fix the inconsistent variable.

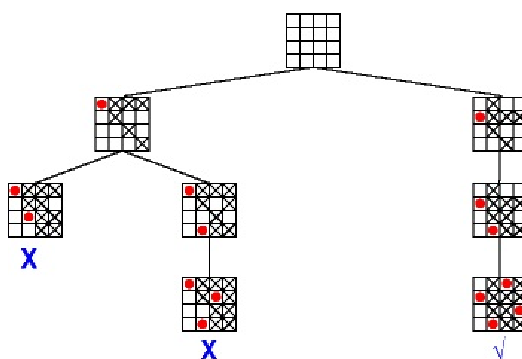


Figure 2.5: Forward checking for the second model of the 4-queens problem (figure by R. Barták [14])

Look ahead or **maintaining arc consistency** (MAC) goes further than forward checking and establishes full arc consistency, meaning that it checks for consistency not only between the recently assigned variable and the non instantiated but also amongst the non instantiated. This approach permits to detect conflicts between future variables (non yet instantiated),as we can see in figure 2.6. But this knowledge comes at the price that a lot more filtering

work has to be done. As said before we need to see if testing inconsistent nodes can be faster than eliminating them previously (as for establishing arc consistency) to choose the best approach.

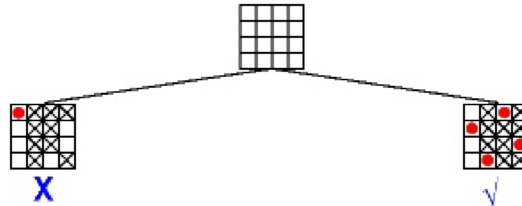


Figure 2.6: Look ahead for the second model of the 4-queens problem (figure by R. Barták [14])

The exploration of the search tree can be guided through variable and value **ordering heuristics** [74], they define in what way the search will choose the next variable to fix (to assign a value to) and what value should it choose, respectively. Examples of heuristics are: choosing the variable with the smallest (biggest) domain, choosing first variables of type A and then those of type B, assigning the smallest (biggest) values in the domain, etc.

2.2.4 Global constraints

To ease the modeling of a CSP and/or improve the search for a solution in CP, global constraints [82] were introduced. Global constraints have one or more of the following as goal:

- to be able to model constraints that are difficult, and sometimes impossible, to express through basic constraints (like linear or logical)
- to reduce the complexity of the filtering algorithm or of the propagation phase
- to implement a more efficient filtering algorithm (one that eliminates more inconsistent values)
- to provide some structural properties in the model that can be exploited by the search

One of the most common global constraints amongst CP solvers is the *allDifferent* constraint. It states that the values of the variables concerned must be pairwise distinct. This constraint could be alternately represented by a set of binary constraints. For every pair of variables in the set of variables

that must be all different we create a constraint stating that the values of this pair must be different.

To exemplify the power of global constraints suppose we have a CSP with 3 variables x_1, x_2, x_3 and an *allDifferent* constraint involving these variables with $D(x_1) = \{a, b\}$, $D(x_2) = \{a, b\}$ and $D(x_3) = \{a, b, c\}$. Establishing generalized arc consistency for the global constraint removes the values a and b from the domain of x_3, while arc consistency for the set of binary constraints does not delete any value. The filtering algorithm for this constraint, proposed by [75], creates a matching between variables and the values in their domains and eliminates edges that cannot be in a maximum matching.

The model for the n-queens problem presented previously, with the *allDifferent* constraint, becomes:

```
int n=...;
range N= 1..n;
dvar int row[N] in N;
dvar int diag1 in N;
dvar int diag2 in N;

subject to {
  forall(i in N) {
    diag1[i] == row[i]+i;
    diag2[i] == row[i]-i;
  };
  allDifferent(queen);
  allDifferent(diag1);
  allDifferent(diag2);
};
```

This model introduces new variables ($diag1$ and $diag2$), and the corresponding search space has n^{3n} tuples. However, this search space is explored much more efficiently thanks to the *allDifferent* constraint propagation.

2.2.5 Constrained Optimization Problem

A constrained optimization problem (COP) is a CSP P with variables $\{x_1, ..., x_k\}$ together with an objective function $f : D(x_1) \times ... \times D(x_k) \longrightarrow \mathbb{R}$ that must be either maximized or minimized. To solve this kind of problem with CP the solver looks for a feasible solution, computes the corresponding objective value, and then adds a new constraint stating that future solutions must have

a better value of objective than it. This process is repeated until the system becomes unfeasible, in which case the last solution found is optimal (or the COP is unfeasible).

For COPs it is possible to use algorithms to calculate, at each node of the search tree, a bound of the objective function. In this way branches can be cut when a bound that is worse than the best solution found so far is calculated. This method is known as branch-and-bound.

When a new solution is found by the branch-and-bound algorithm there are two main options, one can either continue a **depth-first search** (DFS) in the current tree or **restart** a new branch-and-bound procedure. The advantage of the latter being that the heuristic choices can rely on the result of the last propagation which should lead to a better exploration of the search tree. The drawback is that this might lead to re-exploring some parts of the search tree.

The calculation of bounds can be done by any kind of algorithm (not necessarily another CP model). This allows, for example, to combine CP with mathematical programming techniques, as Lagrangian relaxations that can be efficiently calculated. Different papers using this kind of combined techniques are cited in the next section, proposing CP approaches for the TSP.

2.3 CP for the TSP

In 1997, Caseau and Laburthe [23] proposed a set of propagation and branching techniques to improve the performance of constraint programming and make it a state-of-the-art technique for solving small TSPs (of up to 30 vertices). Recent works [17, 36] have shown that Constraint Programming (CP) is competitive with state-of-the-art special-purpose TSP solvers, like Concorde, for medium-size instances. A review and evaluation of the different global constraints that have been used in the literature to address the TSP with CP is given in [36]. Another recent successful application of CP (combined with Lagrangian relaxation) was proposed to a generalization of the TSP in [22] and also shown to be competitive with state-of-the-art exact algorithms.

In [17], the authors develop a special filtering for the *Weighted-Circuit* global constraint (which maintains a circuit - as in Def. 5 - in a weighted graph $G = (V, A, w)$) and compare its performance to the following CP model for the TSP (which we extend to solve the TDTSP, in chapter 6). In this model n represents the number of nodes in V.

$$
\begin{aligned}
\mathit{intVar} \quad & position[1..n] \in 1..n && (2.2)\\
& next[1..n] \in 1..n && (2.3)\\
& prev[1..n] \in 1..n && (2.4)\\
\text{minimize} \quad & \sum_{i \in 1..n} w(i, next[i]) && (2.5)\\
\text{subject to} \quad & position[1] = 1 && (2.6)\\
& \mathit{allDifferent}(position) && (2.7)\\
& \mathit{allDifferent}(next) && (2.8)\\
\forall i \in 2..n: \quad & position[i] = next[position[i-1]] && (2.9)\\
& \mathit{allDifferent}(prev) && (2.10)\\
& \mathit{inverse}(prev, next) && (2.11)\\
& next[1] < prev[1] && (2.12)
\end{aligned}
$$

The reason why this model is a good comparison basis for other CP approaches is because it only uses "simple" global constraints (as *allDifferent* and *inverse*), proposed in most CP solvers.

2.4 Scheduling and CP Optimizer

Temporal domains occur very often in optimization problems. While a temporal domain can be considered isomorphic to an integer domain, time presents important characteristics which should be leveraged to improve the efficiency and convenience of the use of CP when dealing with such domains.

- time domains are too large to be enumerated efficiently
- time is monotonous
- time presents some cyclical and multi-scale aspects (hours, days, weeks, months...) which need to be taken into consideration for the expression of constraints

Over the years, a variety of specialized variable types (intervals) and related global constraints have been developed [13, 57, 58] to improve the expressiveness and efficiency of CP based programs involving temporal domains.

A typical example of a problem that involves temporal domains is the scheduling of tasks on a single machine that in CP Optimizer can be modeled with the global constraint *NoOverlap*. This constraint states that tasks

performed by the same machine cannot overlap in time. In the same way, this constraint can be used to model the TSP by constraining the visits performed by the same salesperson to happen in a sequence. More details about how to model the (TD)TSP with this constraint will be given in chapter 6.3.

Interval variables Generally speaking, CP Optimizer handles *optional* interval variables that is, interval variables that are associated with a Boolean status depending on their presence/absence. For simplicity, we assume here that all interval variables are present.

An interval variable a is a decision variable that represents a task or activity with unknown start and end times, i.e. whose domain $dom(a)$ is a subset of $\{[s, e)|s, e \in Z, s \leq e\}$. An interval variable is said to be fixed if its domain is reduced to a singleton, i.e., if $\hat{a}$ denotes a fixed interval variable then $\hat{a} = [s, e)$. In this case, s and e are respectively the **start** and **end** of the interval and $d = e - s$ its length.

2.4.1 Precedence graph

In order to support the propagation of *NoOverlap* constraints, CP Optimizer uses a *precedence graph* structure [39, 55]. This structure is a directed acyclic graph with nodes representing the interval variables of the *NoOverlap* constraint and arcs the precedence relations between interval variables. Different types of precedence relations are distinguished in the precedence graph, as illustrated on Figure 2.7.

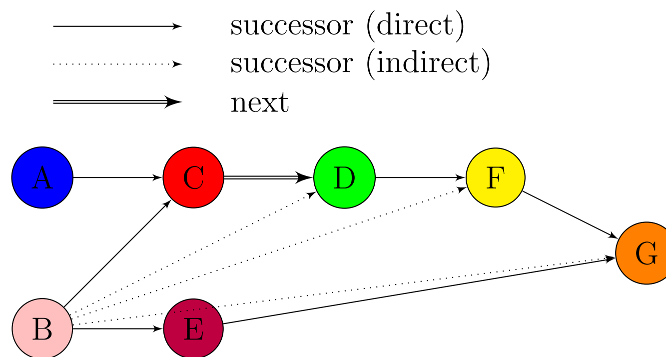


Figure 2.7: Example of precedence graph

A *next* arc between two nodes i and j means that interval variable j always comes next to i in any solution, that is, the position of j in the sequence is

equal to the position of i plus one. A *successor* arc arc between two nodes i and j means that interval variable j will come after i in a solution, that is, the position of j in the sequence is strictly greater than the position of i. Of course, a *next* arc subsumes a *successor* arc. For instance a sequence value $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G$ is compatible with the precedence graph of Figure 2.7. The precedence graph automatically maintains the *transitive reduction* of the *successor* arcs as a set of *direct successor* arcs. Figure 2.7 shows the direct successors of node B (namely C and E) as well as its indirect successors (D, F and G). Note the difference between the notion of *next* and the one of *direct successor*: the notion of direct successor depends on the current topology of the graph and does not imply that the two nodes are next to each other in all the solutions. For instance C is a direct successor of A but it is not next to A in solution $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G$.

The *precedence graph* structure is incrementally maintained when new *next* and *successor* arcs are added during the search (as decisions or because of constraint propagation). For instance the insertion of a *successor* relation $C \rightarrow E$ on the precedence graph of Figure 2.7 would produce the precedence graph of Figure 2.8. We see that the precedence $C \rightarrow E$ has been strengthened as a precedence $D \rightarrow E$ as node E cannot be scheduled between C and D because of the *next* arc between C and D. We also see that because of the path $B \rightarrow C \rightarrow D \rightarrow E$ the successor relation between B and E which used to be *direct* is now an *indirect* one.

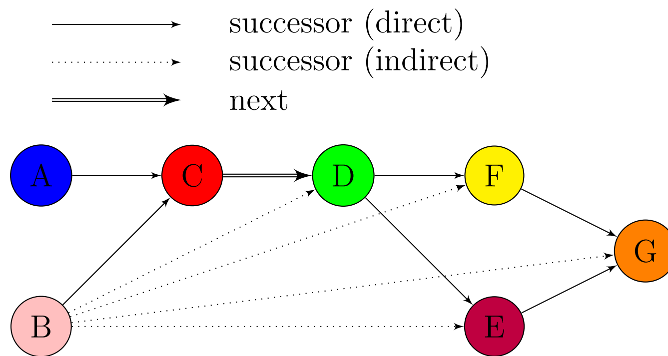


Figure 2.8: Example of precedence graph propagation

The precedence graph structure provides iterators to traverse the set of all successors or all direct successors of a node with a complexity in $O(n)$ if n is the size of the iterated set.

2.4.2 Search

To find a solution, the CP Optimizer search functionality implicitly generates combinations of values for decision variables by means of constructive strategies. These strategies are executed and guided towards optimal solutions in order to converge rapidly. The default search of CP Optimizer uses a variety of strategies and uses the most appropriated one depending on the model structure and on constraint propagation. The two most important ingredients being Self-Adapting Large Neighborhood Search [56] and Failure Directed Search [84], which are briefly presented here.

Self-Adapting Large Neighborhood Search combines several ingredients which are fundamental to its efficiency and robustness:

- *Large Neighborhood Search (LNS)*: by freezing some features of a solution and focusing on re-optimizing the unfrozen features the LNS framework provides a general and efficient traversal of the search space. Compared with Tree Search, it avoids being stuck with wrong early decisions. It is more flexible than Local Search for complex problems involving many types of constraints and resources.

- *Partial Order Schedules (POS)*: a POS is a directed graph G($\mathcal{A}$, $\mathcal{E}$) where the edges in $\mathcal{E}$ are precedence constraints between activities with the property that any temporal solution to the graph is also a resource-feasible solution. Algorithms for transforming a fully instantiated solution into a POS are described in [73, 45].

- *Neighborhoods*: the LNS uses a portfolio of large neighborhoods exploiting the temporal dimension of the problem (time-windows) or the structure (for instance topology of the precedence constraint network) (examples are *RandomizedNHood*, *TimeWindowNHood*, *TopologicalNHood*). They are all based on the initial generation of a POS constructed from a completely instantiated solution where activities have fixed start times and end times.

- *Completion strategy*: some completion strategies use a linear relaxation of the problem and, doing so, has a global vision of the ideal position of activities in time would there be no resource limitation. In the context of LNS where only a part of the POS is unfrozen, this relaxation tends to be very informative as most of the resource constraints are still captured by frozen precedence arcs of the POS. The branching scheme of the strategy allows to exploit constraint propagation and

better explore the bottom of the search tree which clearly is a plus compared to more classical non-backtracking greedy algorithms.

- *Learning*: the re-enforcement learning scheme, although quite simple, ensures a quick convergence on the most effective neighborhoods, completion strategies and their associated parameter values. Learning is a key factor in the robustness of the approach.

Failure Directed Search (FDS) operates under the assumption that the current problem is infeasible, or alternatively, if there is a solution then it is hard to find (heuristic methods already failed to find it). Therefore it supposes that it will explore the whole search space (to prove infeasibility or optimality) or at least a significant part of it (before a solution is found).

With this assumption in mind, FDS gives up on the idea of guiding the search towards possible solutions. It does exactly the opposite: it drives the search into conflicts in order to prove that the current branch is infeasible. Choices that fail the most are preferred. From two branches of a choice the one that fails the most is preferred. It is the well-known first-fail principle but applied also on the branch ordering.

2.5 Discussion

In this chapter we started with preliminary definitions necessary to formally model the TSP which, together with the VRP, describes the real-world problem of delivering multiple nodes of a network with (one or more) vehicles. In urban contexts though, estimating travel times in a constant way as it is done in the TSP and VRP is unrealistic as travel times tend to vary a lot during the day. In the following chapter the time-dependent version of the TSP, the TDTSP, is introduced and the literature concerning this problem is reviewed.

In this thesis we chose to address the TDTSP with Constraint Programming, which we introduced in the second part of this chapter. A brief review of how CP has been used to tackle the TSP shows that CP can be competitive with state-of-the-art methods for the TSP, at least for not too large instances. In spite of this fact, no CP approaches have been investigated in the literature for the TDTSP. The fact that delivery problems usually have many side-constraints, often complicated to model or integrate in meta-heuristic or MIP approaches is a good call for CP, as one of CP's strengths is the ease of integration of constraints. Another reason to pick CP is its good

performance for scheduling problems which is an important dimension of the TDTSP.

Chapter 3

TDTSP(TW)

In the case of urban centers, considering travel times to be constant is clearly a poor model of reality, the biggest cities having huge variations of travel times during the day. According to TomTom's traffic index [3], based on data from 2015, Lyon has a congestion level of 26% - corresponding to the increase in overall travel times when compared to a free flow situation - with 52% of increase during the morning peak and 50% during the evening peak.

In the deliveries business, precise information about travel times allows to calculate more reliable schedules and to improve satisfaction of time-related constraints, like time-windows on deliveries. Another important gain of considering time-dependent travel times is that minimizing total travel time has the potential to avoid congested axes and to reduce greenhouse effects as well as transportation costs.

In this chapter we introduce time-dependency in the TSP, we formally define travel time functions and related concepts to be able to define the TDTSP(TW). We then give a classification of routing problems according to time-dependency to clarify which problem we address in this thesis. In the last section a literature review of the TDTSP/TDVRP is given, from three different angles: the motivations for taking time-dependency into account, the approaches proposed to solve these problems and the instances used for experimentation.

3.1 Definition of the problem

The theoretical problem studied in this thesis is the TDTSP, an extension of the TSP where arc costs depend on the time when the arc is traveled. In the TSP we are given a list of locations and the distances between every two of them. The objective is to find the tour minimizing the total traveled

distance while visiting every location exactly once and coming back to the point of departure (depot). In some cases though, the goal is to minimize the total travel time instead of distance or simply to schedule interventions or deliveries in certain time-windows predefined by the client. For these purposes, one needs to know the travel times between consecutive deliveries. To produce somewhat reliable schedules it seems necessary to take travel time variations into account as, in urban zones, variations can be very important in the course of a day.

In order to integrate time-dependent information about travel times we define in Def. 8 travel time functions, also called transition times in what follows.

Definition 8 (Travel Time Function) *Given a graph $G = (V, A)$, a travel time function $f : A \times \mathbb{R}^+ \to \mathbb{R}^+$ is a function such that for a given arc $(v_i, v_j) \in A$, $f(v_i, v_j, t)$ is the travel time from v_i to v_j when leaving v_i at time t.*

The following property (FIFO) plays an important role in the modeling of travel times and in the difficulty of routing problems. Here we simply define it, as it will be further discussed in the following chapters. It states that a vehicle leaving later from the same departure point and traveling through the same path as another vehicle cannot arrive earlier at the destination.

Definition 9 (FIFO property) *A time-dependent travel time function $f : A \times \mathbb{R}^+ \to \mathbb{R}^+$ is said to satisfy the FIFO (First-In First-Out) property iff:*

$$\forall (i, j) \in A, \forall t, t' \in \mathbb{R}^+, t \leq t' \Rightarrow t + f(i, j, t) \leq t' + f(i, j, t')$$

In the case of deliveries and specially of interventions, the duration of visits is generally not the same so we consider that each vertex v is associated with a given duration $\delta(v)$. The notion of Timed-Path extends the notion of path (definition 2) in order to associate times to vertices.

Definition 10 (Timed-Path) *Given a graph $G = (V, A)$, a starting time $\tau \in \mathbb{R}^+$, a travel time function $f : A \times \mathbb{R}^+ \to \mathbb{R}^+$ and a duration function $\delta : V \to \mathbb{R}^+$, a timed-path is a tuple $TP = (P, a, s, e, l)$ such that $P = (v_1, ..., v_k)$ is a path in G, $a : [1, k] \to \mathbb{R}^+$ defines the arrival time $a(i)$ to v_i, $s : [1, k] \to \mathbb{R}^+$ defines the start time $s(i)$ of visit v_i, $e : [1, k] \to \mathbb{R}^+$ defines the end time $e(i)$ of v_i, and $l : [1, n] \to \mathbb{R}^+$ defines the leave time $l(i)$ from vertex v_i. In the general case, times associated with the vertices in P must satisfy the following constraints (illustrated in Figure 3.1):*

$$
\begin{aligned}
a(v_1) &= \tau \\
s(v_i) &\geq a(v_i), && \forall i \in \{1, ..., k\} \\
e(v_i) &= s(v_i) + \delta(v_i), && \forall i \in \{1, ..., k\} \\
l(v_i) &\geq e(v_i), && \forall i \in \{1, ..., k\} \\
a(v_{i+1}) &= l(v_i) + f(v_i, v_{i+1}, l(v_i)), && \forall i \in \{1, ..., k-1\}
\end{aligned}
$$

In some cases, it may happen that waiting is not allowed on vertices. In this case, we have $a(v_i) = s(v_i)$ and $e(v_i) = l(v_i)$. When the path P contains all different vertices, we may note $a(v_i), s(v_i), e(v_i)$ and $l(v_i)$ the times associated with vertex v_i in P.

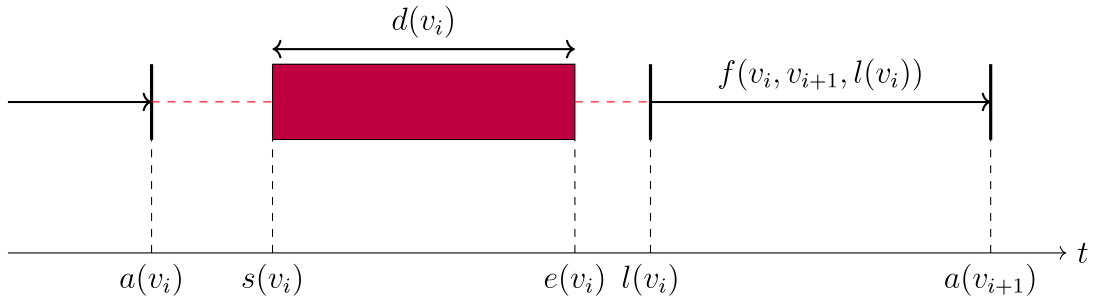


Figure 3.1: Relations between variables of a timed-path

Definition 11 (TDTSP) *Given a graph $G = (V, A)$, a depot $d \in V$, a starting time $\tau \in \mathbb{R}^+$, a travel time function $f : A \times \mathbb{R}^+ \to \mathbb{R}^+$ and a duration function $\delta : V \to \mathbb{R}^+$, the Time-Dependent Traveling Salesman Problem is the problem of finding the timed-path $TP = (P = (v1, ..., vk), a, s, e, l)$ which starts from the depot ($v_1 = d$) and visits each vertex exactly once ($\{v_1, \ldots, v_k\} = V$), and such that the returning time to the depot, $l(v_k) + f(v_k, d, l(v_k))$, is minimal.*

If travel times are not FIFO, waiting at nodes to travel at a more advantageous speed later on might improve the total travel time, but waiting at nodes is not always a possibility since this might imply parking in prohibited areas or require additional costs. Here we assume that waiting is allowed, as in the definition of timed-path, the start time at the node can be greater or equal to the arrival time at the node. The start time of the tour τ could also have been subject to optimization but the only fixed version is considered here.

Travel time functions are not assumed to be symmetrical (i.e. $f(v_i, v_j, t)$ may be different from $f(v_j, v_i, t)$) therefore when we refer to TDTSP we are actually considering the more general asymmetrical version that some call TDATSP. Finally, we can add to the TDTSP time-window constraints, as in definition 12, and define the Time-Dependent Traveling Salesman Problem with Time-Windows (TDTSPTW).

Definition 12 (TDTSPTW) *Given a graph $G = (V, A)$, a depot d in V, a starting time $\tau \in \mathbb{R}^+$, a travel time function $f : A \times \mathbb{R}^+ \rightarrow \mathbb{R}^+$, a duration function $\delta : V \rightarrow \mathbb{R}^+$, and a time-window function $tw : V \rightarrow P(\mathbb{R}^+ \times \mathbb{R}^+)$ such that $tw(v_i) = \{(w_{s1}, w_{e1}), ..., (w_{sp}, w_{ep})\}$ is the set of time-windows associated with vertex v_i, the TDTSPTW is the problem of finding the timed-path $TP = (P = (v_1, ..., v_k), a, s, e, l)$ which starts from the depot ($v_1 = d$) and visits each vertex v_i exactly once during one of its time-window (i.e., there exists (w_{sj}, w_{ej}) in $tw(v_i)$ such that $s(v_i) \geq w_{sj}$ and $e(vi) \leq w_{ej}$), and such that the returning time to the depot, $l(v_k) + f(v_k, d, l(v_k))$, is minimal.*

In the literature two types of (time-window) constraints are usually considered, soft and hard. Shortly said, soft constraints can be violated while adding penalties to the objective and hard constraints have to be satisfied otherwise the problem is considered unfeasible. If soft time-windows are not respected in the final solution, costs of earliness (arriving too early at the client) and/or lateness (arriving too late) might have to be considered. The time-windows used in this work are hard constraints and have to be satisfied as declared by the customer. To ensure that hard time-windows can be respected it is sometimes mandatory to allow waiting at nodes until a time-window becomes available and "service" can start. Otherwise, if waiting is not allowed, the problem might become unfeasible. The TDTSP can be seen as a TDTSPTW where every visit has a unique time-window $(-\infty, \infty)$.

Objective functions In Definitions 11 and 12, we define the objective function as the minimization of the returning time to the depot. Other objective functions might be considered, depending on the final application. For example, the goal might be to minimize CO_2 emissions, the length of the tour or its total cost.

For our application's purpose the ideal optimization goal would be actual cost (so that delivery companies adopt it), with a sub-goal of limiting vehicle emissions (to be beneficial for the city). But the variables that determine the total cost and vehicle emissions are essentially labor costs and fuel usage, both of which are strongly correlated to the actual time spent performing the

tour. Thus, travel time is a good compromise for our purposes, as it happens to be easier to estimate reliably.

In the case where time-windows constraints have to be respected, being too early or too late also impacts costs (parking costs, extra costs for the client). So being able to estimate arrival time at the client reliably, might be a goal in itself.

For the problem of minimizing fuel usage (and therefore CO_2 emissions) two main things have to be taken into account, truck travel speeds and the load being carried by the truck [34]. Since emissions are non-linear in speed [30] (both low and high speeds yield higher emissions) and trucks travel with the traffic time-dependent travel speeds, paths and times where speeds are closer to the emissions-minimizing speed have to be prioritized. Furthermore, the order of deliveries at customers should aim to reduce as much as possible the load of the truck along the way. The diversity of factors that have to be taken into account for this kind of objective function makes it quite complex to optimize.

There is no difference between optimizing total travel time and return time to the depot when waiting is not allowed (or it is allowed and travel times are FIFO), since the end time can be calculated by the sum of travel times and waiting times between vertices. When waiting is allowed (and travel times are not FIFO or there are time-window constraints), minimizing total travel time might imply having larger waiting times (than when minimizing the end time of the tour) which can have larger costs as a consequence, if it implies paying extra time to drivers or if parking costs apply. So, in this thesis, we chose to minimize the end time. The difference in complexity between these two objective functions depends on the model (as well as on the instances).

3.2 Classification of routing problems according to time-dependency

Since the VRP is a generalization of the TSP and also maybe a more frequent application in logistics, an important part of the literature on time-dependent routing addresses the TDVRP rather than the TDTSP. For this reason, in the following sections, literature concerning vehicle routing in general (single or multiple vehicles) is considered. On the other hand, papers concerning applications other than urban routing and scheduling like air traffic control [42] or maritime logistics [51] are not studied here since modeling considerations are very different making their general approach harder to compare. For a

general review of alternative applications of the time-dependent versions of the TSP and VRP, the reader can refer to [43].

It is important to distinguish between the different problems that have been addressed as time-dependent in the literature but do not mean the same thing. Let us divide the problem into five categories concerning their level of 'time-dependency': constant, position-dependent, time-dependent, dynamic and stochastic.

We refer to the TSP (respectively VRP) as the **constant** costs version or the constant problem, sometimes called sequence-dependent in the literature. In this version costs do not depend on time in any degree and only depend on the pair of nodes (origin-destination).

The **position-dependent** version is the problem firstly described by Picard and Queyranne in 1978 [72], when scheduling jobs on a single machine with costs depending on the position of the job in the sequence, i.e., if there are n vertices, then the cost function is defined on $A \times [1, n]$, and $c(i, j, k)$ gives the cost for traveling from i to j when i is the kth visited vertex. Since then, [46, 4, 21, 41] also addressed this version of the problem and used the term time-dependent TSP to describe it even though costs do not directly depend on time.

In the **time-dependent** version, the cost (travel time) from one vertex to the other varies according to the actual departure time from the first vertex. To our knowledge the first authors to define the TDTSP in this way were Malandraki and Daskin in 1992 [66]. Earlier than this, Beasley in 1981 [16] and Ahn and Shin in 1991 [5] defined the equivalent time-dependent version of the VRP. The time-dependent version is limited to predictable changes in costs, for example, due to periodic variations like week days, seasons, holidays, rush hours, construction work, etc. Random events cannot be captured by time-dependent models.

Unpredictable events can be treated either in real-time or in a-priori way. For the **dynamic** version of the problem, the Dynamic TSP (DTSP), costs are updated as they change in real-time and decisions have to be adapted to take changes into account in an online manner. The DTSP was first proposed by Psaraftis [48] in 1988 and the two main approaches used in the literature to solve the problem are evolutionary computation [89, 7] and ant colony optimization [24].

With **stochastic** or probabilistic (a-priori) optimization, on the other hand, one might try to estimate the probabilities of the different possible outcomes and take them all into account when looking for the best solution. Two different problems may be addressed in this case: one may search for a robust solution (such that the probability that the solution becomes unfeasible when executing it with real-time data is lower than a given threshold),

or for a flexible solution (such that, given some online adaptation procedure, the expected cost of the adapted solution with respect to real-time data is optimized). Stochastic travel times are addressed in [59] and [52], for example. Dynamic and stochastic versions of the problem are often considered together, [60] discusses both in depth and addresses different variants of the dynamic and stochastic VRP in his thesis.

Some confusion exists within the vehicle routing literature in what concerns time-dependency: [86, 83] propose a queuing approach (stochastic) for the "dynamic" problem but actually meant the time-dependent problem, different authors referred to the position-dependent problem as time-dependent. For this reason, a clarification of naming conventions seemed important here.

3.3 Literature review

The literature review is organized into three subsections: motivations, methods and experiments. The first subsection reviews the work done to evaluate the interest of taking time-dependency into account for the optimization of vehicles routing problems. In the methods subsection, the different solving approaches used so far to address the TDTSP(TW) and TDVRP(TW) are listed and in the experiments subsection the different kinds of instances used for testing are presented as well as their common characteristics. The two last subsections show that our work fills a gap in what concerns CP approaches and benchmark instances for the TDTSP while the first one motivates the study of time-dependent routing problems.

3.3.1 Motivation: constant versus time-dependent approaches

Different authors studied the effects of taking time-dependency into consideration when studying the TSP or VRP for urban deliveries. One might wonder if constant times can be good enough approximations of reality in such a way that solutions of the constant problem are the same or similar to solutions of the time-dependent problem. To answer this question the common approach in the literature has been to compare solutions of the TD-VRP with solutions to a related constant VRP, where travel times between nodes are averages (or some other constant approximation) of the different travel times during the day. The solution sequence found for the constant problem is then "simulated" using the more realistic conditions and the total travel time (or other objective) is calculated using time-dependent travel times between nodes.

Fleischmann et al. [38] compared constant against time-dependent travel times for seven TDVRP instances (five of which have time-window constraints) ranging between 58 and 786 clients to deliver. Those instances were generated from real-world data coming from the traffic information system of the city of Berlin and vehicle and client's order information were provided by several logistics service providers in the city. Even though their instances come from real data and their solving method does not suffer from considering larger amounts of time steps - they report only small increases in computational times for 5, 10 and 50 time steps - they point to memory as the limiting factor for using 50 time steps or more per travel time function (the paper was published in 2004 but tests were performed years earlier). Total travel times of the optimized sequences (using the varying numbers of time steps from constant to 50) are then simulated using the original (non-agreggated) values of 214 time steps. The main goal of their tests was to compare the different heuristic algorithms (local search) in what concerns the amount of time steps considered for the travel time functions. Since the heuristics do not prove optimality it is not possible to conclude that considering more time steps does not allow to improve results. It may allow nevertheless, to conclude that the heuristics considered by them have similar performances, within the given conditions, for 5, 10 and 50 time steps. On their instances, using constant travel times led to an underestimation of travel times of about 10% and violations of time-windows.

In 2012, Kok et al. [53] developed a realistic speed model, which they applied to road-networks from six US cities, and tested four different congestion avoidance strategies on their Capacitated-VRPTW (VRPTW with capacity constraints for the trucks) instances. The two first strategies are not specifically avoiding congestion actually, and they calculate optimal solutions for the VRP in the first case supposing constant maximum speeds for every arc and in the second case using constant speeds estimated by averaging time-dependent speeds during the day. For strategies 3 and 4, estimated time-dependent travel speeds are used for the TDVRP and the difference between the two is that on top of that strategy 4 calculates time-dependent shortest paths between adjacent nodes of the optimal sequence. Basically, strategies 1 and 2 are constant and 3 and 4 are time-dependent with degrees of "congestion avoidance" increasing from 1 to 4. Kok et al. [53] have 360 Capacitated-VRPTW instances of 15, 50 and 100 customers, truck capacity is set to 55 (judged to be a good trade off not to constrain the length of routes only by time-windows or truck capacity) and start times are fixed. Half of the customers have time-windows with lengths varying randomly from 30 to 90 minutes, service times are either 15 or 30 minutes in the same proportion and customer demands are randomly drawn from 1

to 10. Their primary objective is to minimize the number of vehicles and the secondary is to minimize the end time (called by them, duty time) of all truck drivers. They compare the four different strategies on a number of performance measures, that can be summed up by: number of vehicles, total duty time, total traveled distance and total late time at customers. Averages over all instances are given (results for instances of 100 customers are scaled up from the smaller instances) and results for strategies from 2 to 4 are given in percentages relative to strategy 1. Strategy 2 requires 11.70% more vehicles than 1 and increases the total duty time by 3.34% while reducing the total travel distance by 0.60% and the total lateness by 88.58%. This means that strategy 2 when compared to 1 adds a lot in terms of reliability but also in terms of costs (more vehicles and for longer). Strategies 3 and 4 have quite similar performances with slight improvements using 4. Both require only around 0.5% more vehicles than strategy 1 on average while allowing to eliminate lateness at customers completely and to reduce total duty time by 7.69% and total distance by 1.24% (with strategy 4). The conclusion is that time-dependent strategies have a great impact on the reliability of routes which ultimately has a great impact on costs (of lateness, of hiring drivers for longer, of fuel).

Other authors having performed similar studies are [49, 83, 32, 64], these studies were generally smaller in size or used mostly randomly generated data. Their conclusions in terms of gains of considering time-dependency correspond to our conclusions (presented in Section 8.3) that not taking time-dependent travel times into account has huge impacts on the reliability of solutions in the presence of time-windows. Their estimated gains ranged between 1 and 22%, where smaller gains corresponded to "less time-dependent" situations.

3.3.2 Methods : existing approaches for solving the TDTSP and TDVRP

In this section we give an overview of the different techniques that have been used in the literature to tackle the TDTSP and TDVRP (with and without time-windows). In Figure 3.3.2 a classification of approaches per type of problem is given. The branch that describes TDTSP approaches is probably more complete than the one that describes TDVRP approaches as we are focusing on the TDTSP in this thesis.

Complete approaches for those problems are all based on MIP, and consist mainly on branch-and-cut. A MIP formulation for the TDVRP and its adaptation for the TDTSP is given by [66] but the authors do not discuss

how to solve this formulation, they propose several simple heuristics for that purpose. For the TDTSPTW and TDVRPTW, [6] and [78] propose a transformation into graphical ATSPs but their approach depends on the tightness of time-windows in order to produce a compact graphical representation. Heuristic approaches are more varied and consist for the most part on adaptations from existing heuristics for the TSP/VRP, the majority being based on local search or meta-heuristics.

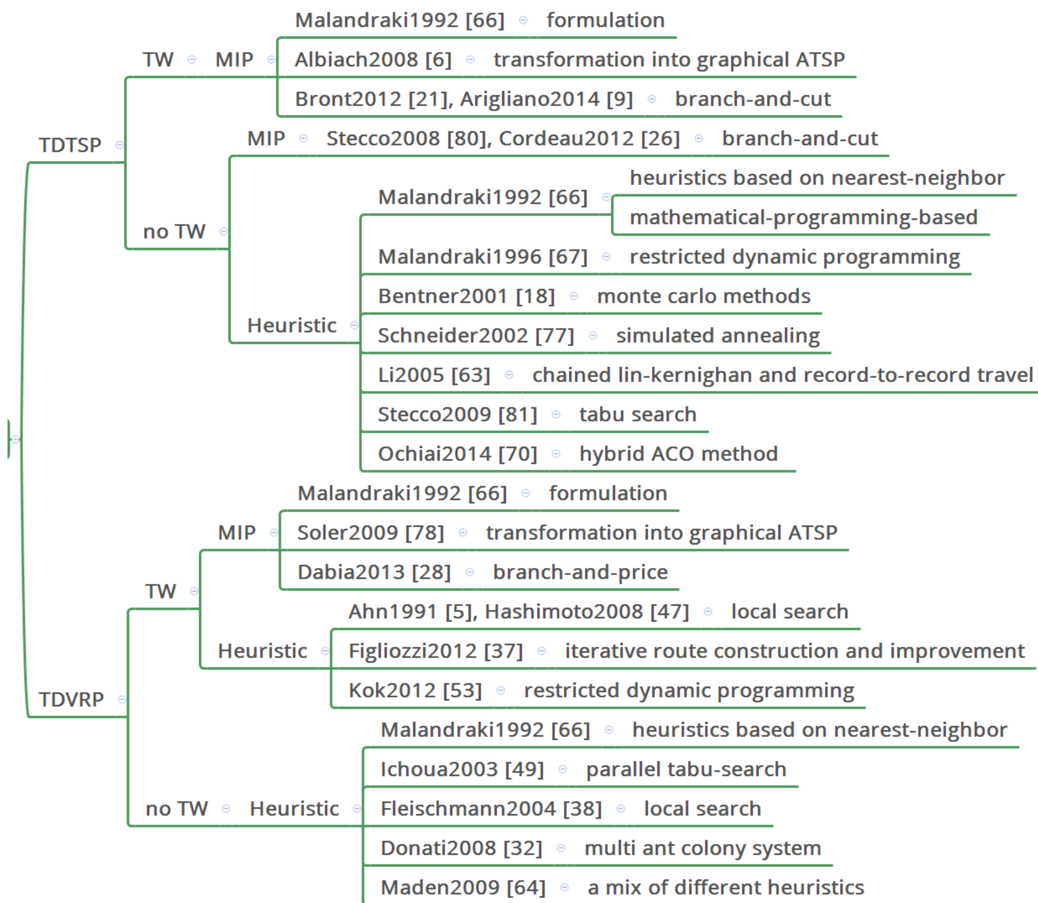


Figure 3.2: Tree of approaches proposed in the literature for the TDTSP and the TDVRP, with and without time-windows (TW and no TW, respectively)

Heuristics present the disadvantage that adding new constraints is not as straight-forward as it usually is with CP (and sometimes with MIP) approaches. On the positive side, some heuristics can be adapted for the TDTSP without considerably increasing complexity, as described by [5]. As

one can see in Fig. 3.3.2, we couldn't find any heuristic approaches for the TDTSPTW but a great variety of heuristics for the TSP have been adapted for the TDTSP (without time-windows).

It is not simple to judge how MIP approaches would behave in the presence of a larger number of time steps (higher granularity in the discretization of time) as the papers listed here under MIP approaches did not test their solution with a number of time steps larger than eight. It would be interesting to see more discussions concerning how MIP models would scale with a larger number of time steps. Certain constraints can also be very hard to express accurately with this kind of approaches, so even though it is an exact approach that can be very fast in certain situations it also seems to come with important limitations in this context.

To the best of our knowledge, the only paper using a Constraint Programming approach for a time-dependent problem is [51], treating two scheduling problems with time-dependent task costs. No papers address the TDTSP or TDVRP with CP.

It seems very hard to compare different methods as they might not address exactly the same problem (one might have fixed start times as the other may not, one might consider FIFO travel time functions and the other general functions) and the lack of a common benchmark also makes comparisons difficult.

3.3.3 Experiments: instances generation and characteristics

In this section we describe instances used across the related literature in order to figure out common characteristics and desirable features of a benchmark. We can distinguish two main types of test sets, instances artificially generated simulating one or two traffic peak-hours and instances based on real traffic data.

Artificially generated instances Two main groups of instances artificially generated should be highlighted here: instances based on Solomon's 100-customer Euclidean problems [79] - used by [49, 47, 28, 32] mainly for studying the TDVRP - and TDTSP instances developed by [26, 9] that are, to the best of our knowledge, the only ones (other than ours) currently available online. Papers [80] and [28] point to instances no longer available online.

Ichoua et al. [49] first adapted Solomon's instances in 2003 using three different scenarios of speed variation. Arcs of the original instances are classified into one of three categories (fast, medium and slow) and three time-step

speeds are associated with each kind giving 3×3 speed matrices for each scenario. The geographical distribution of customers in the original instances follows three different patterns: randomly distributed, clustered and mixed (some random, some clustered). Instances are further differentiated by number of customers that can be delivered by one vehicle, some having very tight time-windows and others, very large. The number of clients being served per instance can vary between 25 and 100. With a lot of similarities to Ichoua's instances but with some particularities, the instances generated by [26] are described in details in section 8.5 as we tested our method on them. The sizes of their instances in terms of nodes to be visited varies between 15 and 40.

Only in rare cases in the related literature tests were performed on instances with more than 100 nodes (both real-world and artificial are listed):

- TDTSP instances - [18, 77, 63] used adapted version of BIER127 which is a TSPLIB95 [76] problem with 127 nodes; [70] up to 200 customers;
- TDVRP instances - [38] tested on four instances of more than 100 with 371, 672, 761 and 786 deliveries (using 15, 18, 20 and 84 vehicles, respectively), two of which with time-windows; [5] up to 200 customers.

Most randomly generated instances use travel times or speeds that simulate one or two traffic peaks during the day, having only between 3 to 5 time steps, [6] and [21] consider 8 time periods of about 1 hour each.

Real-world instances Instances using real-world data were used by [38, 35] in Germany, [64] in the UK, [32] in Italy and [70] in Japan. In most cases, authors modeled the road network including real-world data and then proceeded to extracting the corresponding instances to be optimized (as exemplified in the following chapter), the only exception being [70] which calculates shortest paths and tours in an integrated manner. Another common trait between these test instances is that they all have a much more detailed granularity of time than those artificially generated, dealing with a number of time steps on the order of one or two hundreds and of lengths varying from 5 to 15 minutes.

When time-window constraints are considered they are usually unique per client and present for either 50% of customers or 100%. Questions concerning the choice between speeds or times are discussed in section 5.1.1.

3.4 Discussion

In this chapter we have set the formal basis to study the TDTSP. First, formal definitions of the TDTSP and of the most common side-constraint, the Time-Window constraint, were given and the choice of the objective function to be optimized was discussed. Then, a classification of routing problems according to time-dependency was given in order to clarify which works in the related literature address the issue (of time-dependency) in a similar way and are, therefore, comparable with our study.

Once the exact problem we are addressing is clearly defined, the related literature was reviewed in three subsections: the interest of studying the time-dependent problem rather than the constant one, the methods used to solve the time-dependent problem and the types of instances used for experiments.

In the first subsection we present the studies done in the literature comparing constant to time-dependent approaches. The goal of these studies is to evaluate the potential gains of considering time-dependent information but also to evaluate what is the impact of using constant information in the real-world feasibility of the solutions found. All conclude that time-dependency is important to take into account and perhaps even more so in the presence of time-windows. We also performed this kind of study during this thesis, using the benchmark instances we generated, our results are reported in section 8.3.

In the second subsection the approaches used in the literature to tackle time-dependent routing problems (TDTSP, TDVRP) were listed. We point out that CP approaches are rare or non-existing, we could only find one paper [51] treating a similar problem to the TDTSP with CP. Exact MIP approaches seem to be more restricted in terms of the number of time-steps in the modeling of travel time functions they can take into account. To the best of our knowledge there are no papers using MIP approaches in the literature reporting tests with a larger number of time-steps than 8.

In subsection 3.3.3, instances were differentiated between artificially generated simulating traffic peak-hours and based in real-world data. They were then analyzed in terms of number of nodes (customers) and of number of time steps of the travel times/speeds functions[1]. The number of customers is usually between 15 and 100 but some cases with larger amounts were listed. The number of time-steps varied with respect to how instances were generated, for artificial ones the number of time-steps considered tends to be quite small (most commonly around 3 or 4) while for realistic ones they tend to

[1] Note that we will discuss the choice to use times or speeds in the model more in detail in section 5.1.1

be larger, in the order of hundreds. None of the authors studying real world data made their instances available and only one of the artificially generated benchmarks is currently accessible online. The available instances only consider up to 40 customers and no time-windows which are limitations when one looks at the common requirements here described. In that sense, our benchmark instances presented in the next chapter might fill a gap in what concerns the ease of evaluation and comparison of TDTSP related methods.

Chapter 4

A new benchmark for the TDTSP derived from real-world data

In the context of the Optimod project we had access to real traffic data measured from 630 sensors installed in the main axes of Lyon for 6 years. In this chapter we describe how we generated a TDTSP(TW) benchmark from this data and present the modeling choices we made with respect to time-dependent travel times functions. A TDTSP instance in our benchmark consists of a list of visits with their corresponding duration times, a time-dependent travel time function and, in the case of the TDTSPTW, time-window constraints.

Here we distinguish between the city's road-network and the TDTSP instance graph. The **road-network** is a graph where vertices represent intersections between different streets (or different segments of the same street) and edges correspond to street segments. The TDTSP **instance graph** is an aggregated version of the road network where vertices represent delivery addresses and edges correspond to shortest paths connecting two different addresses.

In the first section of this chapter, we give a brief description of how the travel speeds used later in this chapter were estimated. In the second section, all the steps to transform the road-network graph into an instance graph are explained in details. Finally, in the third section, the benchmark instances generation is described.

4.1 Estimation of travel speeds

In this section, we give some background on what kinds of measuring of traffic are done and how travel speeds are predicted from the collected data.

There are two main ways in which travel speeds on a given street can be measured:

- Data coming from sensors inside vehicles: as GPS, Floating car data (data coming from cellphones)
- Inductive-loop traffic detectors: electromagnetic systems buried in the traffic lane that are able to detect vehicles passing over it or stopped within the loop

The first option is limited as not enough data is available, so in this thesis we use data coming from (a total of 630) inductive-loops installed across the city of Lyon. These loops measure density (vehicles/km) and flows (vehicles/hour), one measure every 6 minutes, and from these values it is possible to estimate speeds using the fundamental diagram of traffic flow. The three graphs shown in Fig. 4.1 are related by $flow = speed * density$.

In practice, we used tables given in an internal report from OptimodLyon that allow to obtain average travel speeds for all possible ranges of occupancy rates and different types of streets (according to their vehicle capacities, speed limit, etc.). For streets without sensors estimations are made through an interpolation of data from neighboring streets, taking the streets' directions into account in the calculations (for example, neighboring streets going in opposite directions do not impact estimations for one another).

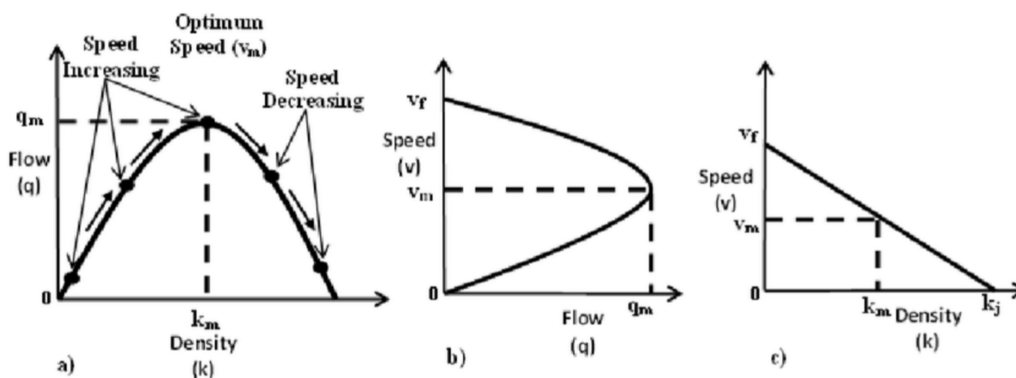


Figure 4.1: Fundamental diagram of traffic flow (from [88])

Then, travel speeds on a given road segment are predicted for every 6-minute time step by calculating the median value of speeds on the same time step of similar days (week days, public holidays, sunny or rainy). More about the predictive model developed specifically for the Optimod'Lyon project can be read at [87], the model described here is a simpler version developed with the only purpose to generate the benchmark instances described later in this chapter.

4.2 From road-network to instance graph

The road-network graph of Lyon is extracted from the Open Street Map database [2]. The extracted graph has about 100000 edges, corresponding to one-way or two-way road segments. The edges are connected through vertices having an average arity of about 2.3. This arity is fairly small, because the graph needs to be quite detailed in order to allow for all possible and compliant road maneuvers when calculating routes.

As indicated in [15], we collected real delivery tours data, with a total of about 10000 tours and an average of 20 visits per tour. After cleaning the data and verifying delivery addresses, we have randomly selected 255 addresses, among the set of addresses occurring in real tours, which are displayed in Fig. 4.2.

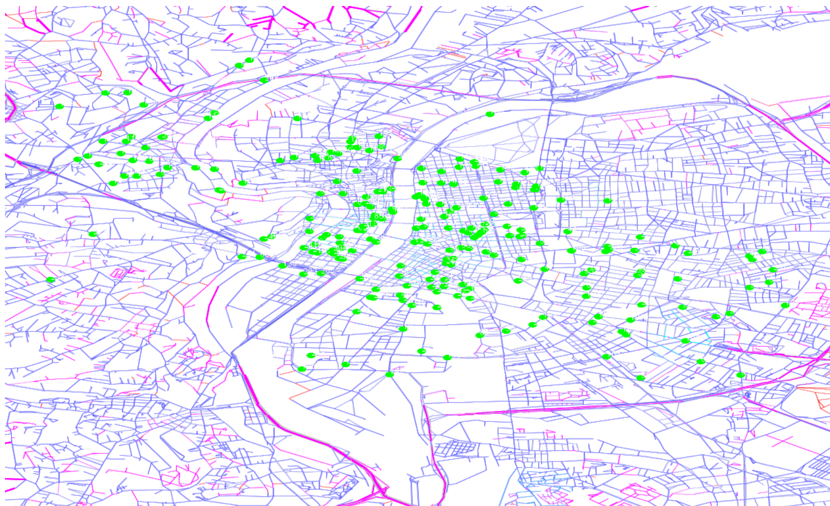


Figure 4.2: Lyon's road-network with 255 delivery addresses

The first step to build the instance graph is to add the 255 delivery addresses as new vertices in the road-network graph, possibly breaking some of the road-network's edges in two, in cases where the address is located

within a street segment. The next step is to compute shortest paths between every couple of delivery addresses, these shortest paths correspond to the edges of the instance graph (Fig. 4.3). Usually, for TSP instances, this step corresponds to finding shortest paths in terms of distances, in a road-network graph where the edge-costs are the lengths of the associated street segments. We, on the other hand, are interested in shortest paths in terms of time, in a context where travel durations are time-dependent. This means that we want to know for each moment t and each pair of addresses (a, b) which path leaving from a at time t arrives the earliest at b. In order to calculate the time-dependent travel times of edges in the instance graph, we need to have time-dependent travel duration information for each edge in the road-network graph. Travel times in the road-network are obtained from speeds estimated from the data collected from sensors, as explained in the previous section.

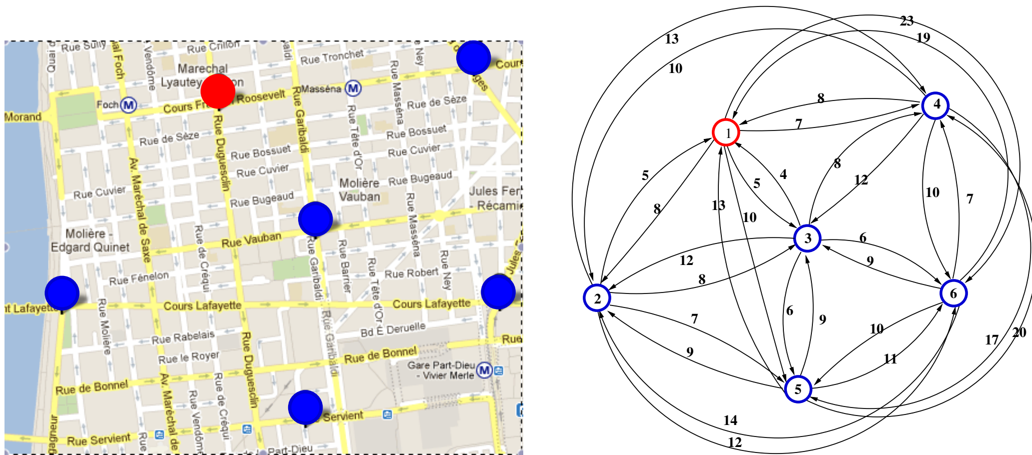


Figure 4.3: Example of first transformation step, on the left, where delivery addresses are added to the road-network graph, and on the right, the instance graph generated with shortest paths costs for every edge

4.2.1 Estimating time-dependent travel times from time-dependent speeds

A predictive model is used to estimate time-dependent travel speeds on the road-network from historic traffic data for each of the 65 time steps of 6 minutes, from 6:00 to 12:30, as described in Section 4.1. From these travel speed functions (which are step-wise functions as illustrated on the left graph of Fig. 4.4) we need to calculate travel times, that will be used to calculate

shortest paths in terms of time in the next step. A simple way of calculating the travel time t_{time} on an edge from a given travel speed t_{speed} (at a certain departure time t_0) would be to divide the length l of the edge by the speed at time t_0: $t_{time}(t_0) = l/t_{speed}(t_0)$. By following this procedure one would obtain a stepwise travel time function, the red function on the right graph of Fig. 4.4. In the example of the figure, a travel time ti is obtained by dividing the length of the edge by the corresponding speed vi with $i \in 0, 1, 2, 3, 4$.

This approach presents the drawback that stepwise travel time functions do not respect the FIFO property (proof in the following chapter 5) and this property is essential for the next step of the extraction of an instance graph from the network-graph (calculating time-dependent shortest paths). The importance of this property is that the computation of time-dependent shortest paths is a polynomially solvable problem in FIFO-networks (networks in which travels happen in a FIFO manner) [50] whereas it is NP-hard for non-FIFO networks [71].

In order to generate travel time functions (from travel speeds) respecting the FIFO property (Def. 9), we use an extension of the algorithm proposed by Ichoua et al. in [49]. The original, referred to as IGP algorithm in what follows, is described in Algorithm 1. Given a time-dependent travel **speed** function on an edge, the IGP algorithm allows to calculate the travel **time** on that edge for a specific departure time. We provide an extension, Alg. 2, of the IGP algorithm that calculates the whole (piecewise linear) travel time function at once, instead of just one travel time value for a given departure time. This algorithm would calculate the black travel time function on the right graph of Fig. 4.4, which respects the FIFO property (proof in the following chapter 5).

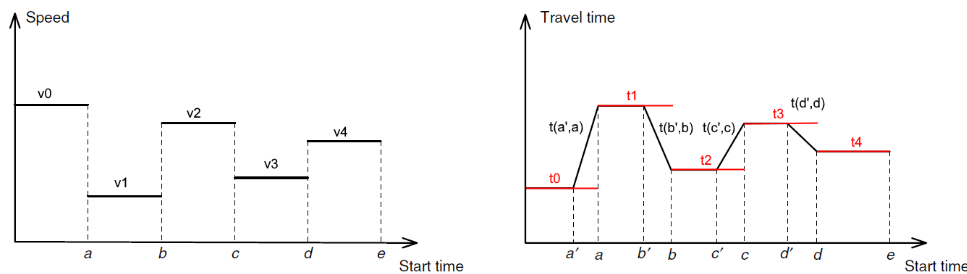


Figure 4.4: A stepwise travel **speed** function on the left transformed in travel **time** functions on the right. A stepwise function in red obtained by dividing edge lengths by speeds and a piecewise linear function in black obtained using the IGP algorithm (figure modified from [28])

For a given departure time, it might happen that the time it takes to travel through an edge is larger than the length of the current step of the speed function, meaning that the speed on that edge might change **during the travel**. The goal of the IGP algorithm is to take these changes into account (they might happen multiple times during one travel) and calculate the travel time left on an edge always using the most recent travel speed. Whereas with the simple method presented in the first paragraph of this section, only the speed occurring during departure time was considered.

The IGP works by iterating over time steps, updating the speed used in the travel time calculation, as soon as the current time (departure time plus travel time so far) crosses the boundary of a time step. For example, in Fig. 4.4, the travel times $t(b', b)$ in the interval $[b', b[$ are calculated using both speeds of the second and third steps, $t(b', b) = \frac{l_1}{v1} + \frac{l_2}{v2}$, where l_i is the portion of the total length l that was traveled using vi, l_i is a function of the departure time. While travel times from a to b' (in this case, constantly $t1$) only require the speed from the second step, $t1 = \frac{l}{v1}$.

Algorithm 1 IGP algorithm

Input: - an edge (i, j) of length L_{ij}
- a number of time steps M, such that for $k \in [1, M]$, $[T_k, T_{k+1}[$ is the kth time interval
- a stepwise speed function for (i, j), $v : M \to \mathbb{R}^+$ such that the speed on (i, j) is equal to $v(k)$ at any time t in $[T_k, T_k + 1[$, for each $k \in [1, M]$
- a starting time $t_0 \in [T_1, T_{M+1}[$

Output: The time needed to traverse (i, j) when leaving from i at time t_0

1: $t \leftarrow t_0$ // t is the current time
2: Let $k_0 \in [1, M]$ be the time step index such that $T_{k_0} \leq t_0 \leq T_{k_0+1}$
3: $k \leftarrow k_0$ // k is the index of the time step that contains t
4: $d \leftarrow L_{ij}$ // d is the portion of (i, j) that remains to be traversed at time t
5: $t' \leftarrow t + d/v(k)$ // t' is the arrival time on j if speed had a constant value equal to $v(k)$
6: **while** $t' > T_{k+1}$ **do**
7: $\quad d \leftarrow d - v(k) * (T_{k+1} - t)$
8: $\quad t \leftarrow T_{k+1}$
9: $\quad t' \leftarrow t + d/v(k + 1)$
10: $\quad k \leftarrow k + 1$
11: **end while**
12: **return** $t' - t_0$

Algorithm 1 must be called for each possible initial time $t_0 \in [T_1, T_M + 1[$, for each edge (i, j). To speed-up this computation, [28] proposes to search for "breakpoints"[1], which are points where the travel time function might change its derivative, and only use the IGP algorithm to calculate travel times corresponding to these breakpoints - instead of adopting a more naive approach like iterating over all possible starting times and using the IGP algorithm for each of them. The resulting travel times, corresponding to each of the calculated breakpoints, are then connected by linear pieces giving a continuous piecewise linear function as result (this part is omitted from the algorithm as it is straightforward). In Fig. 4.4, the breakpoints of the travel time function are $a', a, b', b, c', c, d', d$.

To understand how Algorithm 2 looks for breakpoints, we observe that, for a given starting time t_0, if more than one time step has to be used in the calculation of the travel time then for all subsequent starting times (belonging to the same time step as t_0) there will be a slope - increasing or decreasing, depending on the speed function. As long as only one time step speed is required the travel time remains constant and as soon as a new time step is needed in the IGP algorithm the slope might change and therefore a new break-point has to be calculated.

Algorithm 2 calculates (and returns) the set of breakpoints β corresponding to speed function v (on edge (i, j) with M time-steps), $IGP((i, j), M, v, t_0)$ returns the travel time for departure time t_0 using the IGP algorithm (Alg. 1). In what follows we suppose $(i, j), M, v$ fixed and call the algorithm with only the departure time: $IGP(t_0)$. For departure time t_0 the related arrival time is given by $t_0 + IGP(t_0)$. To find breakpoints corresponding to a fixed time step $ts = [T_k, T_{k+1} - 1]$ (T_{k+1} is considered in the subsequent iteration), the algorithm looks at the time step boundaries crossed between $T_k + IGP(T_k)$ and $T_{k+1} - 1 + IGP(T_{k+1} - 1)$, the arrival time if leaving at the start of ts and at the end, respectively. For each boundary T_h crossed, the departing time (somewhere inside time step ts) must be calculated, to do so a "backward" version of the IGP algorithm was developed. This version is given in Appendix A and the departing time obtained with the backward algorithm is denoted by $BackIGP((i, j), M, v, T_h)$.

[1] They do not provide a detailed procedure for this purpose (search for breakpoints).

Algorithm 2 Extension of IGP algorithm

Input: - an edge (i, j) of length L_{ij}
- a number of time steps M, such that for k in $[1, M]$, $[T_k, T_{k+1}[$ is the kth time interval
- a stepwise speed function for (i, j), $v : M \rightarrow \mathbb{R}^+$

Output: The set β of breakpoints for edge (i, j) and speed function v

1: $\beta \leftarrow \emptyset$
2: **for all** $k \in \{1, .., M\}$ **do**
3: $\quad \beta \leftarrow \beta \cup T_k$
4: $\quad t_b \leftarrow T_k$ // the first time of time-step k
5: $\quad t_e \leftarrow T_{k+1} - 1$ // the last time of time-step k
6: $\quad k_b \leftarrow k' : T_{k'} \leq t_b + IGP(t_b) < T_{k'+1}$ // the time-step of the arrival time when leaving from i at time t_b
7: $\quad k_e \leftarrow k'' : T_{k''} \leq t_e + IGP(t_e) < T_{k''+1}$ // the time-step of the arrival time when leaving from i at time t_e
8: $\quad$ **for all** $h \in \{k_b, .., k_e\}$, the set of time-steps between k_b and k_e **do**
9: $\quad\quad t_h \leftarrow BackIGP(v, T_h)$ // the time of departure in order to get to arrive at time T_h
10: $\quad\quad \beta \leftarrow \beta \cup t_h$
11: $\quad$ **end for**
12: **end for**
13: **return** β

4.2.2 Time-dependent shortest paths calculation

As mentioned earlier, finding time-dependent shortest paths is polynomially solvable in FIFO-networks and a straight-forward adaptation of Dijkstra's point-to-point shortest path algorithm to the time-dependent case is enough to solve the problem. The time-dependent adaptation of Dijkstra, Algorithm 3, was proposed by [33], which at the time did not notice it only worked in cases where travel times respect the FIFO property.

Given a graph $G(V, A, f)$ (where $f : A \rightarrow \mathbb{R}^+$ is a function giving the arc costs), a source vertex $s \in V$ and a destination $d \in V$, Dijkstra's classic algorithm (for the static case) maintains an array of tentative distances $T[u] \geq t(s, u)$ for each vertex, where $t(s, u)$ is the shortest distance to go from s to u. The algorithm visits (or settles) the vertices of the road network in the order of their distance to the source vertex and maintains the invariant that $T[u] = t(s, u)$ for visited vertices. When a vertex u is visited, its outgoing arcs (u, v) are relaxed, i.e., $T[v]$ is set to $\min(T[v], t(s, u) + f(u, v))$, where $f(u, v)$ gives the distance of arc (u, v). Dijkstra's algorithm terminates

when all target vertices are visited. In the time-dependent version of Dijkstra's algorithm presented here, the only change is that function f - usually the weight or distance of an arc in the graph - becomes a time-dependent function as highlighted in red in Alg. 3.

Algorithm 3 Time-Dependent Dijkstra

Input: a graph $G = (V, A, f)$ with arc costs $f : A \rightarrow \mathbb{R}^+$, an initial vertex $s \in V$, a set of final vertices $D \subseteq V \backslash \{s\}$, a departure time τ

Output: an array $T : V \rightarrow \mathbb{R}^+$ such that for all $v \in D$, $T[v]$ is the shortest time to go from s to v when leaving s at time τ

```
1: T[v] ← ∞ for all v ∈ V
2: T[s] ← 0
3: Q ← V
4: nbDone ← 0
5: while nbDone ≠ |D| do
6:    u ← argmin_{v∈Q}{T[v]}
7:    if u ∈ D then nbDone++
8:    remove u from Q
9:    for all (u, v) ∈ A do
10:      if T[v] > T[u] + f(u, v, τ + T[u]) then
11:        T[v] ← T[u] + f(u, v, τ + T[u])
12:      end if
13:   end for
14: end while
15: return T
```

Shortest travel times starting from every address are then calculated, for every 6-minute time step, using Alg. 3, which we will call $TDDijkstra$. Stepwise travel time functions are generated by using the beginning of each time step (at minute 0, 6, 12,...) as departure time input (τ) for $TDDijkstra$ and using the output of the algorithm as the travel time function's value for the corresponding step, as described in Algorithm 4. The output of $TDDijkstra(G(V, A, f), \tau, s, d)$ (with source s, a set of destinations d, and departure time τ) gives the travel time from s to every destination in d when leaving s at time τ, whereas the output of Alg. 4 is a stepwise (6-minute time steps) travel time function $f'(s, d, t)$ (from source s to destination d). Since for a given time step $ts_k = [T_k, T_{k+1}[$, the travel time is constant, we use $f'(s, d, ts_k)$ to refer to the (constant) travel time of all departure times belonging to the interval ts_k.

Algorithm 4 Time-Dependent Travel Times Calculation

Input: $G(V, A, f)$, s, d

1: **for all** $k \in 1, ..., M$ **do**
2: $ts_k \leftarrow [T_k, T_{k+1}[$
3: $\tau \leftarrow T_k$
4: $f'(s, d, ts_k) \leftarrow TDDijkstra(G(V, A, f), \tau, s, d)$
5: **end for**
6: **return** $f'(s, d, t)$

As we will see later, our solving approach for the TDTSP can handle piecewise linear functions but in the benchmark we decided to model travel times as stepwise functions in order to simplify its usage. Piecewise linear functions could have been generated from the same data by using more sophisticated algorithms for the time-dependent shortest path problem as the ones described in [29]. The fact that we use stepwise travel times (therefore, not respecting the FIFO property anymore) is further discussed in the following chapter, along with a proof of the fact the stepwise travel time functions are not FIFO.

4.2.3 Generation of dilated travel time functions

One limitation of our calculation of shortest paths is that it does not take into consideration the time spent in vertices in the path (corresponding to intersection delays) but we know from experience that the time it takes to traverse a crossroad or to turn left, for example, is an important factor in the augmentation of travel times during rush hours. For a more detailed discussion about route travel time estimation the reader can refer to [15] (in the context of Optimod'Lyon, using real-world data) and to chapter 5 of [20].

Since our model tends to underestimate the travel time in congested areas, we generated two additional versions of the function with a dilatation of travel times of respectively 10% and 20% centered on the average travel time. This means that values above the average are augmented of 10% or 20% and values below the average are decreased by the same amount. So we end up with 3 functions: $T00$ (the original one), $T10$ and $T20$. This correction method might reduce travel times at less congested moments of the day below their real values but the main point here is to try to better approximate total variation of travel times instead of travel times per se.

The travel time functions given in the benchmark are naturally represented as stepwise functions. Figure 4.5, displays the 3 different functions ($T00$, $T10$ and $T20$) for two different pairs of addresses (corresponding to

two different edges of the instance graph).

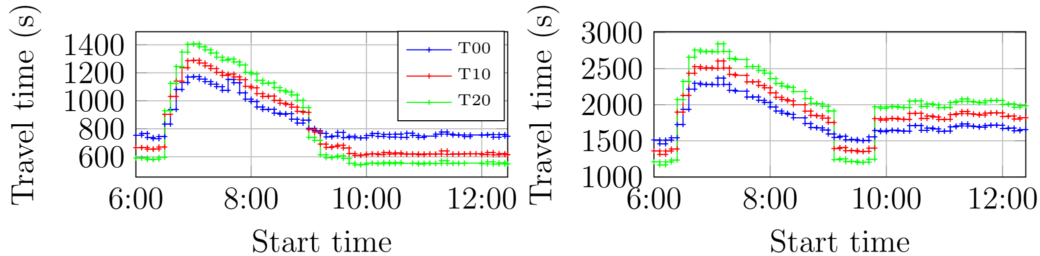


Figure 4.5: Examples of time-dependent travel time functions between two adresses

Since our instances with 100 visits require a total travel time longer than 6.5 hours and that we have not computed[2] travel times for after 12:30, we created travel time functions with twice the previous number of times steps by taking the symmetrical values with respect to 12:30. We suppose that morning rush hours are repeated similarly by the end of the evening, the resulting functions have 130 time steps, ranging from 6:00 to 19:00.

4.3 Benchmark instances generation

Given the graph G with 255 vertices, corresponding to real delivery addresses, and travel-time functions between all pairs of vertices computed as described previously, we generated a benchmark for the TDTSP. In order to study scale-up properties, we consider different problem sizes $n \in \{10, 20, 30, 50, 100\}$, where n is the number of deliveries (vertices). For each problem size n, we randomly generated 500 different instances. Each instance is obtained by randomly selecting n vertices, among the 255 vertices of G, and randomly generating a visit duration $d(v_i) \in [60s, 300s]$ for each selected vertex.

For the generation of instances with time-window constraints we simply add time-windows to the previously generated instances. To ensure that all instances are feasible (i.e., have at least one tour that satisfies all time-window constraints) the time-window constraints have been defined in the four following steps:

[2]Benchmark generation was done in a preliminary phase in the thesis. Different computers and systems were used subsequently which made it complex to come back and generate new travel times.

1. We generate a feasible solution (not necessarily a good one) to a TDTSP instance without time-windows. Let s_i and e_i denote the start and end time of visit i in the solution.

2. For each visit i in the solution, we randomly select the number of time-windows $\in \{0, 1, 2\}$ with respective probabilities 0.50, 0.25 and 0.25.

3. For each time-window we randomly select its duration in $[3600, 7200]$, given in seconds, where $dTW1$ and $dTW2$ denote the durations of the first and the second time-window if applicable.

4. We generate the first time-window so that it contains $[s_i, e_i[$ and the second time-window in the remaining space without overlapping.

In step 1, to avoid adding biases to the instances with time-windows, half of the generated instances come from initial solutions to the original TDTSP instance and the other half come from solutions to the associated TSP instance (generated by simply taking constant travel times, corresponding to the mean of the time-dependent travel times, in the TDTSP instance). The solution is found by stopping the CP search, for the (TD)TSP model, after finding three solutions.

In step 4, the start and end times of the time-windows are selected in such a way that the visits in the solution belong to a time-window (so the solution generated in 1 is still feasible). An example of placement of time-windows for a given solution is displayed in Fig. 4.6. The procedure to place the time-windows is as follows.

The rule for placing the start of the first time-window denoted $sTW1$ is:

- if $s_i > \max(0, e_i - dTW1)$ then pick $sTW1$ randomly in the interval $[0, s_i - \max(0, e_i - dTW1)]$
- else $sTW1 = \max(0, e_i - dTW1)$

Given a value $delta$ of minimal distance from the first time-window, $solVal$ the value of the objective function of the solution found in step 1 and $horizon = 1.5 * solVal$, the rule for placing the start of the second time-window denoted $sTW2$ is:

- if $sTW1 > 0.75 * solVal$ and $sTW1 - dTW2 - delta > 0$ then pick $sTW2$ randomly between 0 and $sTW1 - dTW2 - delta$
- else if $(\max(0, sTW1 - dTW2 - delta) + \max(0, horizon - dTW2 - eTW1 - delta)) > 0)$ then pick $sTW2$ randomly between 0 and $\max(0, sTW1 - lTW2 - delta) + \max(0, horizon - dTW2 - eTW1 - delta)$

- else $sTW2 = -1$

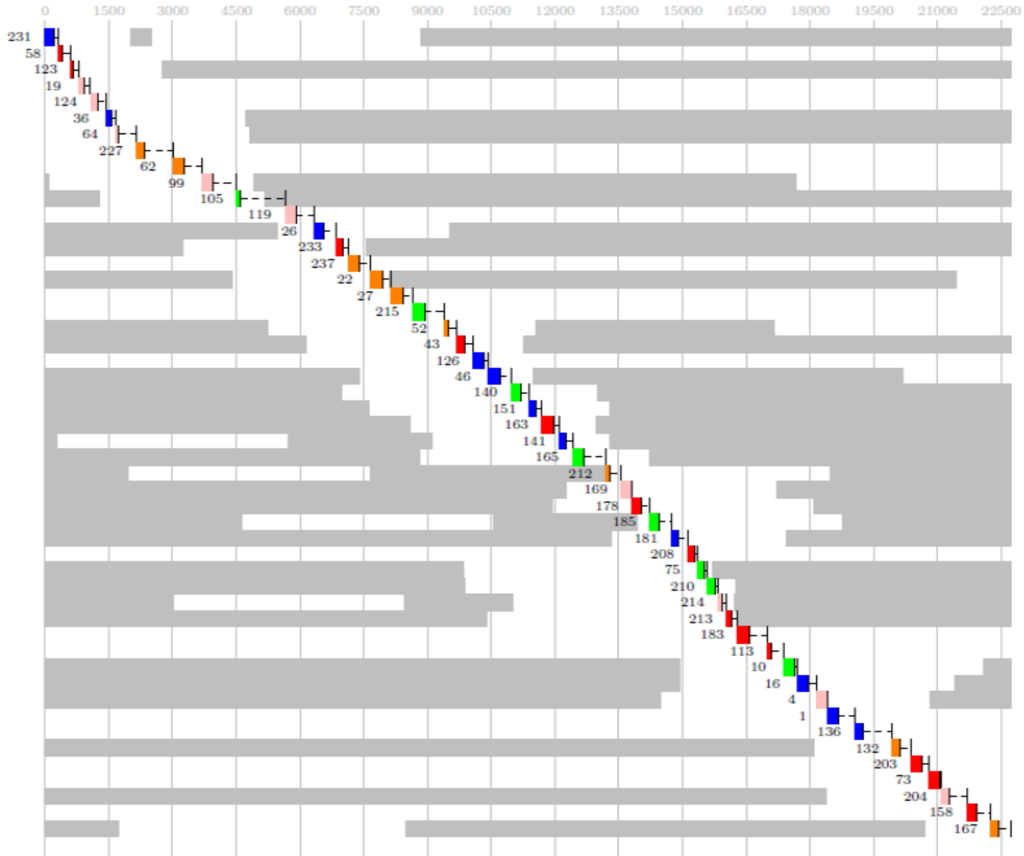


Figure 4.6: Example of generated instance with 50 visits where gray zones represent forbidden delivery times and white zones correspond to the authorized time-windows. The colored boxes are as long as the visit duration and the dashed lines represent transition times between visits

A selection of the benchmark instances presented in this chapter is available at `http://liris.cnrs.fr/christine.solnon/TDTSP.html`. The procedure to select 20 instances per size to use in the benchmark is described in section 8.2. A larger set of 60 instances per size (only for sizes 10, 20 and 30 and without time-windows) was used in a previous study [68] and the results obtained as well as the instances are also given in the same web page.

4.4 Discussion

In this chapter we described how to transform real-world data to be able to use it as input for TDTSP optimization models in general. This is an essential step in any real-world application but it is not frequently mentioned in the literature of routing problems. The methods we used are good enough to generate realistic travel times, and therefore realistic time-dependent instances, but are not state-of-the-art and definitely not suitable for large scale applications. The implementation of more efficient algorithms would take longer and were not the focus of the thesis.

In the previous chapter we reviewed the literature related to the TDTSP and TDVRP in what concerns test instances and showed that having an available benchmark for the TDTSP(TW) might be of great value for the community studying routing and scheduling problems in urban context. One of our contributions with this thesis was the generation of the benchmark presented in this chapter, which is the first benchmark for the TDTSP(TW) with realistic travel times to be made available online. In the second part of this chapter the instances' generation process was described in details. There is no doubt that routing problems are vast and applications and models may vary a lot but we hope that this benchmark can fill a gap in what concerns the study of the TDTSP(TW).

Two main modeling choices concerning travel time functions were made for the benchmark instances described in this chapter:

- to represent them as stepwise functions as we judged that this representation might be easier to integrate in other approaches, even though in our approach we take (the more general) piecewise linear functions into account.
- to give travel **times** instead of travel **speeds** (sometimes used in the related literature) to represent an instance. This choice is discussed in the following chapter.

Chapter 5

Modeling time-dependent travel time functions

The modeling of travel times plays an essential role in the optimization of vehicle routing problems. A model too precise may be complex to integrate in the optimization process while a too simple model may yield a solution far from optimal or infeasible in real conditions. Therefore, the level of precision in the representation of travel times should be calibrated according to the optimization method. In this chapter we discuss the mathematical representation of time-dependent travel time functions as well as how to transform these functions in such a way that they respect some interesting properties. It is important to notice that, in our method, these transformations are used only internally, in the propagation algorithms presented in chapter 7. **We do not use them to modify the problem solved**, we aim to develop a model flexible and general enough to address different applications of the TDTSP that do not necessarily have to satisfy the properties described in this chapter.

The chapter is divided into two sections, the first one concerning the FIFO property and the second one concerning the triangular inequality property. In both sections, the importance of those properties is discussed in the context of vehicle routing optimization as well as how to transform travel time functions to satisfy these properties. The first section also contains a discussion about the choice of speeds or times as arc costs for the instance graph.

5.1 The FIFO property

In the time-dependent road-network context the FIFO property (Def. 9), also known as non-passing property, states that two identical vehicles traveling

through the same path must not pass each other. Here, the "same path" might be defined in two different ways, the same edge in the instance graph or the same path in the road-network graph. Noting that **an edge in the instance graph can represent different paths in the road-network** at different times of the day. This has implications on the choice of times or speeds to model travels on the instance graph, which will be discussed in subsection . In the following subsections, properties of FIFO travel time functions are described and used to explain how to transform a non-FIFO travel time function into a FIFO function. Finally, a review of the main methods proposed in the literature for this kind of transformation is given.

Most authors (for instance, [5, 49, 38, 37]) argue that the FIFO property is essential to be taken into account since it is a better model of reality. But, in reality, traffic will probably not vary as predicted and the FIFO property may no longer hold. In the case of an accident, for example, a vehicle that gets stuck on traffic will arrive later than another one that leaves after but takes an alternative path (that would have been longer if travel times were as predicted). Despite it being questionable whether FIFO travel times are really a better model of reality, many methods depend on the satisfaction of this property to work.

The TDTSP instances we have generated (as described in the previous chapter) do not necessarily satisfy the FIFO property due to the fact that travel time functions are stepwise functions. For the sake of generality, we do not make the FIFO hypothesis in our model, we accept as input time-dependent travel time functions that do not necessarily respect this property. On the other hand, it is important to be able to compute functions that respect this property for internal propagation of our constraint, as explained later in chapter 7.

5.1.1 Travel times or travel speeds?

The discussion of whether it is equivalent, or not, to use times or speeds to model time-dependent travels appears quite often in papers concerning time-dependent vehicle scheduling. Even though the information that is needed for scheduling and most times for optimization (when the objective is to minimize total travel times) are travel times, using time-dependent travel speeds on the instance graph and calculating specific travel times only when needed was first proposed by Hill and Benton [19] in 1992. Their method of deriving travel times from travel speeds did not produce FIFO travel times though, this question was addressed by Ichoua et al. [49] in 2003.

As mentioned previously, Ichoua et al. [49] developed Alg. 1 to transform stepwise travel speed functions in time-dependent travel time functions re-

specting the FIFO property. In their paper, though, they transform travel speed functions given for the **instance graph** edges and not for the road-network graph as we did. They say that there are no differences between associating travel times or speeds to edges since one can be calculated from the other but the problem is that only one shortest path (of the road-network) can be considered if travel speeds are associated to edges of the instance graph. Of course, as it happens frequently in papers studying the (TD)TSP, some simplifications have to be made in order to focus on solving the theoretical problem and the first steps of modeling reality are often overlooked. Nevertheless, their algorithm (IGP) is one of the most cited for time-dependent travel time modeling in vehicle routing context and it can also be used for a more accurate modeling of reality as proposed by us in the previous chapter.

In 2004, Fleischmann et al. [38] also pointed out that using travel speeds is a more limiting choice since it only allows to consider one path for a certain edge in the instance graph (the distance must be fixed in order to compute the travel time on the edge). As mentioned in the section introduction, an edge in the instance graph can represent different paths in the road-network at different times of the day and this possibility can be captured by calculating time-dependent shortest paths yielding time-dependent travel times. In this sense, if one wants to use travel speeds for edges of the instance graph, to be able to model different shortest paths of the road graph, it would be necessary to also keep a time-dependent distance function, in such a way that the correct travel times can be calculated later on. For this reason, we choose to use time-dependent traveling times instead of speeds as edge's cost function, in the instance graph.

Recently, in 2013, Ghiani and Guerriero [44] presented a method allowing to find, for any continuous piecewise linear travel time model, satisfying the FIFO property, what they called its "IGP parameters": a stepwise travel speeds function and a constant length (distance), for each arc (i, j) of the instance graph. They concluded from this that the drawback presented by [38] was wrong. We propose a different way of seeing things, we agree with [38] in that associating travel speeds to the instance graph does not allow to consider different shortest paths in the road-network unless, of course, a function with the distances of the different shortest paths occurring during the day is kept as well, for every arc. But it is also true that IGP parameters can always be found as [44] have shown, i.e., once a FIFO travel times function has been calculated one can find a stepwise speeds function and an arc length (constant during the whole day) that produces FIFO travel times, when using the IGP algorithm. On the other hand, the speeds calculated in this way **do not correspond to actual speeds** on the shortest path taken in the road-network. Ghiani and Guerriero recognize this by calling

them "dummy" speeds. In the same way, the arc length obtained as an IGP parameter is fake and does not correspond to the length of a path in the road graph.

As a final observation, in some contexts it might be more useful to know the time-dependent travel speeds in the graph as in the case of minimizing CO_2 emissions, where travel speeds information is required to estimate fuel usage, as pointed out by [34]. Ultimately, the choice of modeling with travel speeds or times depends on the exact problem being studied (as different objective functions and constraints are possible) as well as on the chosen solving technique, which might deal better with one option or the other.

5.1.2 Properties of FIFO functions

Depending on how the time-dependent travel times were calculated it is possible that the generated functions do not respect the FIFO property. To study the conditions required for a function to be FIFO we first define the arrival time function (associated with a given time-dependent travel time function) as the function giving the arrival time at the destination for every departure time at the source (this function is also called ready time by [28]).

Definition 13 (Arrival time function) *The arrival time function* $f_{arr} : A \times \mathbb{R}^+ \rightarrow \mathbb{R}^+$ *associated with a time-dependent travel time function* f *is:*

$$f_{arr}(i, j, t) = t + f(i, j, t), \forall (i, j) \in A, \forall t \in \mathbb{R}^+$$

Let us assume a fixed arc (i,j) and note f(i,j,t) as f(t) for short. From definitions 9 (FIFO) and 13 it is straight-forward to note that the arrival time function has to be monotonically increasing (nondecreasing) for the travel time function to be FIFO i.e., $\forall t \leq t', f_{arr}(t) \leq f_{arr}(t')$.

From a simple manipulation of Def. 9, a FIFO function f should respect $\forall t \leq t', \frac{f(t')-f(t)}{t'-t} \geq -1$, it is easy to conclude from there that continuous pieces of the travel time function must respect $\frac{df(t)}{dt} \geq -1$. For discontinuous points, decreases in the value of the function are not FIFO but increases do not disrespect the property: if t is the "decreasing" discontinuity point, leaving at $t + \epsilon$ allows to arrive earlier than leaving at $t - \epsilon$ for ϵ sufficiently small (Fig. 5.1). Typically, stepwise functions are FIFO if they are nondecreasing, which is never the case of real-world travel time functions (as travel times tend to decrease after rush hours, for example).

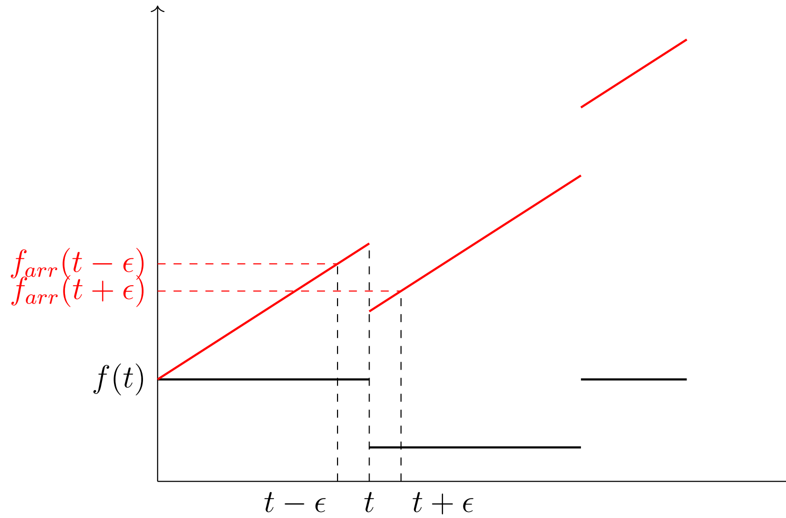


Figure 5.1: Example of non-FIFO decreasing step. The function in red f_{arr} gives the arrival times associated with the travel time function in black f

To render stepwise functions FIFO we need to introduce (non-constant) linear pieces and the resulting function is piecewise linear. Piecewise linear functions that do not respect the FIFO property can be transformed in the same manner but they stay piecewise linear after the transformation. They are the simplest (non-constant) structure one can use that is "invariable" through FIFO transformation and they are also easily integrated in our solving approach.

5.1.3 FIFO transformation

In 1992, Malandraki and Daskin [66] said that if waiting is permitted at nodes then stepwise travel time functions behave as piecewise linear functions when the steps are decreasing. This is also the main idea we used in Alg. 5 to make travel time functions respect the FIFO property. Ichoua et al. [49] subsequently argued that this procedure "induces useless waiting at nodes" but in reality travel times do not change in a stepwise manner and smoothing functions by simulating waiting at nodes is only a way of thinking about how travel times would likely behave in reality. This smoothing method does not imply that vehicles should actually wait before traveling through an arc.

In what follows we present the algorithm developed to transform general piecewise linear functions to respect the FIFO property. Differently from the IGP algorithm (Alg. 1) presented in the previous chapter, our algorithm calculates a FIFO travel time function from a (piecewise linear) travel time

function, while the IGP takes travel speeds as input. The function returned by the algorithm is piecewise linear but not necessarily continuous since only decreasing times need to be changed in order for the function to be FIFO. We do not transform stepwise travel time functions into **continuous** piecewise linear functions because we wanted to ensure a **minimal** set of changes that would still produce a FIFO function, since these FIFO functions are used for propagation but not for verification of solutions. Basically, the transformed function in our case has to be a FIFO function that is the closest possible lower bound of the original function so, if increasing discontinuities were also smoothed, the bound would only get further from the original function.

The algorithm works by replacing decreasing discontinuities and linear pieces with a slope smaller than -1 by linear pieces with a slope of exactly -1, where the slope of a linear piece between two points (x_1, y_1) and (x_2, y_2) is $\frac{y_2 - y_1}{x_2 - x_1}$. The interpretation is to simulate waiting at the departure vertex whenever it can reduce the travel time on the edge. In those cases, the new travel time becomes the waiting time plus the future (smaller) travel time taken. In figure 5.2 an example of an execution step of Alg. 5 is shown, the red dashed lines represent the FIFO function that is being calculated (piece by piece) from the non-FIFO one in dark lines.

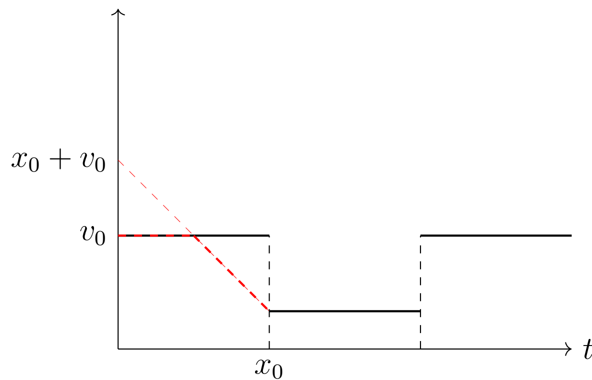


Figure 5.2: Transformation of a stepwise function into a FIFO function

Algorithm notation

We suppose a fixed arc (i, j) and f a piecewise linear function and $f(i, j, t)$ is simplified to $f(t)$. Each time interval $p_k = [t^k_{min}, t^k_{max})$ on which the function is linear is called a **piece**. Since p_k is open on t^k_{max}, by abuse of notation, for a given k, we write $f(t^k_{max})$ for $\lim_{x \to t^k_{max}} f(x)$. For instance, if f is not continuous on $t^k_{max} = t^{k+1}_{min}$ then $f(t^k_{max}) \neq f(t^{k+1}_{min})$. Finally, $linear((t, v), (t', v'))$ denotes the linear function defined by (but not limited by) the two points

(t, v) and (t', v') and $min(f_1, f_2)$ returns the minimum function, the function assuming the minimum value between f_1 and f_2 at every point. The notation $f \restriction_p$ means that f is restricted to interval p and therefore all operations are done only in this interval.

Algorithm 5 Calculate f_{FIFO}

Input: a piecewise linear function f, composed of ν linear pieces such that for each piece $k \in [1, \nu]$, $p_k = [t^k_{min}, t^k_{max}[$ is the kth time interval on which f is linear

1: $f_{FIFO} \leftarrow f$
2: **for all** pieces $p_k = [t^k_{min}, t^k_{max}[$ of f, with $k \in [1, \nu - 1]$ **do**
3: $\quad x_0 \leftarrow t^{k+1}_{min}$
4: $\quad v_0 \leftarrow f(x_0)$
5: $\quad$ Let s_k be the slope of p_k
6: $\quad$ **if** $v_0 < f(t^k_{max})$ **or** $s_k < -1$ **then**
7: $\quad\quad f_{FIFO} \restriction_{[0,x_0)} \leftarrow \min(f_{FIFO} \restriction_{[0,x_0)}, linear((0, x_0 + v_0), (x_0, v_0)))$
8: $\quad$ **end if**
9: **end for**
10: **return** f_{FIFO}

Literature on FIFO transformation methods Since Ahn and Shin [5] proposed in 1991 to use FIFO travel times to enhance the performance of existing heuristics, the majority of approaches for the TDTSP work based on the validity of this property. Most recent approaches adopt the IGP method, i.e., they use speeds on the instance graph and calculate travel times only when needed, for a specific departure time.

The only approach similar to ours (that we could find) was Fleischmann's [38], which also had access to real-world data and modeled travels on the instance graph using travel times calculated from shortest paths on the road-network. Differently from us, their focus was on obtaining continuous piecewise linear FIFO functions, in such a way that the inverse function always exists. Even though for our constraint propagations we also need to calculate some sort of inverse function in order to do "backward" propagations (i.e., propagate the latest leave time from the previous visit in order to arrive at the current visit at a certain time), we can calculate an adapted version from a non-continuous piecewise linear FIFO function (described in chapter 7). Their FIFO transformation method is illustrated in Figure 5.3, in [38] they describe how to calculate parameters δ_{ijk}. The algorithm they use to calculate the FIFO transformation takes a departure time as input and calculates the corresponding FIFO travel time as output.

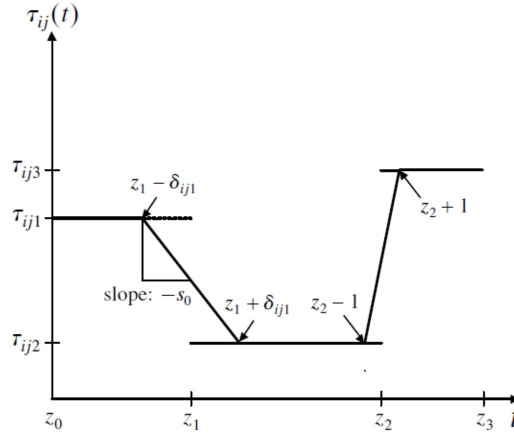


Figure 5.3: Fleischmann's FIFO transformation [38]

Those are the two main methods proposed in the literature to calculate FIFO travel times: the IGP algorithm described in the previous chapter that transforms stepwise speeds into piecewise linear travel times and Fleischmann's transformation that, similarly to our algorithm, takes stepwise travel time function and calculates a piecewise linear FIFO travel time function from it.

5.2 The triangular inequality property

Another property that is usually expected for TSP instances is that they respect the triangular inequality, both because it seems reasonable to expect road networks to satisfy it and because some heuristics make use of this property to reduce the search space and can only provide performance guarantees in its presence. In the context of road networks this property makes sense since paths between any two points are calculated to be the shortest possible. In the context of time-dependent routing though not much has been said about this property other than some authors mentioning whether the property is used as an hypothesis or not (for instance, [67, 21] mention the triangular inequality is not assumed in their time-dependent approach).

Once again, we do not make this hypothesis in our models but we need to be able to calculate time-dependent travel times satisfying the triangular inequality (from the original ones) for propagation purposes. So here we formally define an extension of this property to the time-dependent case and present an algorithm that takes a time-dependent instance and turns it into an instance satisfying the time-dependent triangular inequality (so that the two instances have the same optimal solution).

In general, if there exists a shortest path from i to j, shorter than the direct arc, then the triangular inequality does not hold. It means that there is at least one vertex k such that passing through k allows to arrive faster at j. When travel times vary with the departure time, the triangular inequality has to be adapted in order to take time-dependency into account. The main difference is that the time to departure from the intermediate vertex k to the final destination j must now be at least the departure time from i plus the travel time from i to k. The departure from k could also happen later, if waiting is allowed, for this reason the minimum travel time with waiting time included is considered in Def. 14.

Definition 14 (Time-dependent triangular inequality) *A time-dependent travel time function $f : A \times \mathbb{R}^+ \rightarrow \mathbb{R}^+$ is said to satisfy the triangular inequality property if and only if $\forall i, j, k \in V, \forall t, \delta \in \mathbb{R}^+$, then:*

$$f(i, j, t) \leq f(i, k, t) + f(k, j, t + f(i, k, t) + \delta) + \delta$$

For the same reasons as for the FIFO property, time-dynamic road-networks have no reason to respect the triangular inequality (whenever unpredicted events take place). On the other hand, if the distances of the road-network respect the triangular inequality and the instance graph was generated by calculating time-dependent shortest paths, one could expect that the time-dependent triangular inequality would be valid on the instance graph. If it is not valid on the instance graph then there is one k such that it is shorter (takes less time) to go from i to j going through k first but this contradicts the fact that the path from i to j is the time-dependent shortest path.

The only problem is that our time-dependent shortest paths were calculated per time step (for each couple of vertices (i, j) and each time step $[t1, t2[$, we compute the duration d of the shortest path from i to j when leaving i at time $t1$, and we assume that the duration of the travel from i to j is equal to d during the whole time step $[t1, t2[$). And, in the same way as with the FIFO property, the time-dependent triangular inequality is no longer guaranteed without continuity between different time-steps.

The following algorithm is an extension of the Floyd Warshall All Pairs Shortest Path algorithm [27] to the case of time-dependent functions. The values $f_{min}(i, j)$ and $f_{max}(i, j)$ represent respectively the minimum and maximum travel times assumed by $f(i, j, t)$ and can be easily stored while reading the functions. The *if* in line 4 reduces the number of *min* calculations that the algorithm needs to perform by eliminating paths that clearly cannot be shorter.

The difference between this algorithm and the original Floyd Warshall lies on line 5, where now the time-dependent triangular inequality has to be enforced. In Def. 14, if function f is FIFO then waiting is never advantageous ($\delta = 0$) and the right-hand side becomes simply $f(k, j, t + f(i, k, t))$. Here we consider that the travel time functions given as input are FIFO, if not they can be transformed using the algorithm given in the previous section.

Let $\nu_{i,j}$ be the number of linear pieces of function $f(i, j, t)$. The algorithm calculating the *min* on line 5 (enforcing the triangular inequality) goes through all the pieces of $f(i, k, t)$ and $f(k, j, t)$ to calculate the composed function $f(k, j, f(i, k, t))$ (with $\leq \nu_{i,k} + \nu_{k,j}$ pieces). Then it goes through the pieces of $f(i, k, t)$ and $f(k, j, f(i, k, t))$ to calculate their sum. And finally through the pieces of $f(i, j, t)$ and $f(i, k, t) + f(k, j, f(i, k, t))$ to calculate the minimum. Therefore, the *minimum* calculation runs in $O(\nu_{i,j} + \nu_{i,k} + \nu_{k,j})$ and if ν is the maximum number of steps of any given travel times functions then the Time-Dependent Floyd Wharshall runs in $O(n^3 * \nu)$.

Algorithm 6 Time-Dependent Floyd Warshall

Input: $f(i, j, t)\ \forall i, j \in \{1..n\}$
1: **for all** $k \in \{1, ..., n\}$ **do**
2:   **for all** $i \in \{1, ..., n\}$ **do**
3:     **for all** $j \in \{1, ..., n\}$ **do**
4:       **if** $k \neq i$ **and** $k \neq j$ **and** $i \neq j$ **and** $f_{min}(i, k) + f_{min}(k, j) \leq f_{max}(i, j)$ **then**
5:         $f(i, j, t) \leftarrow \min(f(i, j, t), f(i, k, t) + f(k, j, f(i, k, t) + t))$
6:       **end if**
7:     **end for**
8:   **end for**
9: **end for**

5.3 Discussion

In this chapter we presented two properties: (1) the FIFO property, which is commonly required in time-dependent routing problems (shortest paths problems included) and (2) the time-dependent triangular inequality, which to best of our knowledge is a definition introduced by us since most authors either do not mention this property in the time-dependent context or say that they do not suppose it holds. We insist on the fact that we do not suppose these properties hold in our models since we want to be as general as possible in our modeling but functions satisfying these properties will be automati-

cally created for propagation purposes of our constraint `TDNoOverlap-TDTSP`, presented in chapter 7. Algorithms to turn a time-dependent travel time function into a FIFO function and to make a TDTSP instance respect the time-dependent triangular inequality are presented. The FIFO transformation has been studied by other authors and a comparison of the different methods and approaches to the question is made in the first section.

Chapter 6

Modeling the TDTSP with existing CP concepts

In this chapter we present an adaptation for the TDTSP of the CP model for the TSP given in Section 2.3 and show the limitations of this adapted model through an example. A scheduling model addressing one of the limitations pointed out is then proposed for the TDTSP, using the current constraints proposed by CP Optimizer.

In what follows, we consider that f is a step function where each time-step has the same length l so that f is modeled with a cost matrix T. The input data is :

- A number $n > 0$ of visits, by convention the first vertex in the list of visits is considered as depot and for modeling purposes we duplicate the first visit (the depot) and create a visit $n + 1$ which represents the end of the tour.

- A time horizon $H > 0$, a number of time steps $m > 0$ and a duration $l > 0$ of time steps so that $H = lm$.

- A cost matrix $T : [1, n + 1] \times [1, n + 1] \times [0, m - 1] \to \mathbb{R}^+$ so that the travel time from vertex i to vertex j when leaving from i at time t is given by $T[i][j][\lceil t/l \rceil]$.

- A visit duration vector $D : [1, n] \to \mathbb{R}^+$.

6.1 Extension of the classical CP model for the TSP

We present here a TDTSP model adapted from the classic CP model used to solve the TSP, described in Section 2.3. We added variables $time[i]$, which give the arrival time at each vertex i, and modified constraints to take into account the fact that a duration D_i is associated with every vertex i, and that travel durations are time-dependent.

$$
\begin{array}{rll}
\textit{intVar} & position[1..n] \in 1..n & \\
& next[1..n+1] \in 1..n+1 & \\
& prev[1..n+1] \in 1..n+1 & \\
& time[1..n+1] \in 0..H & \\
\text{minimize} & time[n+1] & \\
\text{subject to} & position[1] = 1 & (6.1) \\
& time[1] = 0 & (6.2) \\
& prev[1] = n+1 & (6.3) \\
& next[n+1] = 1 & (6.4) \\
& \mathit{allDifferent}(position) & (6.5) \\
& \mathit{allDifferent}(next) & (6.6) \\
& \mathit{allDifferent}(prev) & (6.7) \\
& \mathit{inverse}(prev, next) & (6.8) \\
\forall i \in 1..n+1: & next[i] \neq i & (6.9) \\
& prev[i] \neq i & (6.10) \\
\forall i \in 1..n: & position[next[i]] = position[i] + 1 & (6.11) \\
\forall i \in 2..n+1: & position[prev[i]] + 1 = position[i] & (6.12) \\
\forall i \in 1..n+1: & time[i] \geq time[prev[i]] + D[prev[i]] + & \\
& T[prev[i]][i][time[prev[i]]/l] & (6.13) \\
\forall i \in 1..n+1: & time[next[i]] \geq time[i] + D[i] + & \\
& T[i][next[i]][time[i]/l] & (6.14)
\end{array}
$$

Note that the number of positions in a path is $n+1$ since we have to return to the depot. For each visit i, variables: $next[i]$ and $prev[i]$ give the next and previous visits, $position[i]$ gives the position of the visit in the path and $time[i]$ is the time of arrival at i, as shown in Fig. 6.1. Besides constraints at the extremities of the tour to fix initial and end visits and the

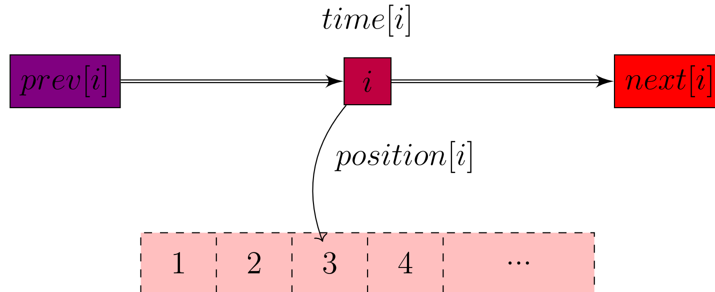


Figure 6.1: Illustration of the classical CP model's variables

start time ((6.1) to (6.4)), *allDifferent* constraints ((6.5) to (6.7)) are posted on each group of variables, whereas $prev/next$ variables are linked with an *inverse* constraint (6.8), which enforces $next[prev[i]] = i$ for all i. Constraints (6.9) and (6.10) prevent cycles of size 1 and constraints (6.11) and (6.12) state the relations between *position* variables and *next* and *prev* variables. The relation between time and relative positions of visits is modeled with constraints (6.13) and (6.14). For a stronger propagation, the term T[...] in these constraints is modeled using a table constraint (see [65] for more on table constraints).

The following redundant constraints were added to the model to help improving the lower bound on the objective term $time[n+1]$. This lower bound is used to prune the current branch of the search tree, each time it is greater than the best known solution. Therefore, improving this bound usually improves the solution process. Instead of propagating the objective function just considering the values of $time[prev[n+1]]$, the redundant constraints allow to have a better estimate of the total cost of the tour by taking all transition times and durations into account. We noticed that these redundant constraints help reducing the number of branches by a factor close to 2 and the CPU time by a factor varying between 1 and 2.

$$time[n+1] \geq \sum_{i \in 1..n} D[i] + \sum_{i \in 1..n} T[i][next[i]][time[i]/l] \tag{6.15}$$

$$time[n+1] \geq \sum_{i \in 1..n} D[i] + \sum_{i \in 2..n+1} T[prev[i]][i][time[prev[i]]/l] \tag{6.16}$$

In the search branching scheme used to compare the performance of the propagation in chapter 8.4.2, we use a search that builds the sequence of visits in a chronological order. In order to keep track of positions in the

sequence, we added a new set of variables $atPosition[j]$ that represent the vertex at the jth position in the sequence. In Figure 6.1, $position[i] = 3$, in this case $atPosition[3]$ should have the value i. Variables $atPosition$ are related with the rest of the model thanks to the following constraints:

$$
\begin{array}{rl}
\textit{intVar} & atPosition[1..n] \in 1..n+1 \\
\text{constraints} & \textit{allDifferent}(atPosition) \\
 & \textit{inverse}(position, atPosition) \\
 & atPosition[1] = 1 \\
 & next[atPosition[n]] = n+1 \\
 & atPosition[n] = prev[n+1] \\
\forall j \in 1..n: & next[atPosition[j]] = atPosition[j+1] \\
\forall j \in 1..n: & prev[atPosition[j+1]] = atPosition[j]
\end{array}
$$

6.2 Limitations

In the context of TDTSP, time variables $time[i]$ representing the dates of a visit are very important and their domain should be as tight as possible for two main reasons: the value of the travel time depends on the value of $time$ (which in turn affects the domains of the related visits through propagation) and it also allows for more propagation in the presence of time-window constraints. An important limitation of the model presented above is the weakness of the propagation between temporal variables $time$ and sequencing variables (like $next$ and $prev$). For instance, it should be clear from their formulation that constraints like (6.13) and (6.14) would benefit from some more global reasoning over the travel time between i and $prev[i]$ (resp. between i and $next[i]$). Furthermore, reasoning only locally on direct successors of a visit ($next$, $prev$) may miss some important propagation as illustrated by the following example.

A visit a is called a *successor* of another visit b if a comes somewhere after b in the path, it is called *next* of b if it is visited exactly after b. It is possible to see that, with the previous model, all the propagation is done reasoning with direct neighbors of a visit ($next$, $prev$).

We can show that reasoning with successors (besides prev/next variables) allows to obtain tighter bounds on the time of visits, as soon as the problem is asymmetric[1] (in the sense that reversing a solution may change its total

[1]Some common causes of asymmetry are: asymmetric travel times (like time-dependent travel times), time windows constraints or precedences between visits.

travel time or its feasibility). For simplification we will work here with a TSP example.

Consider the following slightly asymmetric TSP problem where D is the depot (which we separate into departure and arrival nodes, D_d and D_a, to make things more clear), A, B, C are visits and distances of matrix T are shown directly on the graph, in Figure 6.2. We suppose that the upper-bound of the objective is 100, therefore only two paths are feasible (D_d, B, C, A, D_a) and (D_d, C, B, A, D_a), each with a total length of 100, as shown in Figure 6.2. Given these two feasible solutions, the tightest possible domains of *prev* and *next* variables can be seen in Table 6.1.

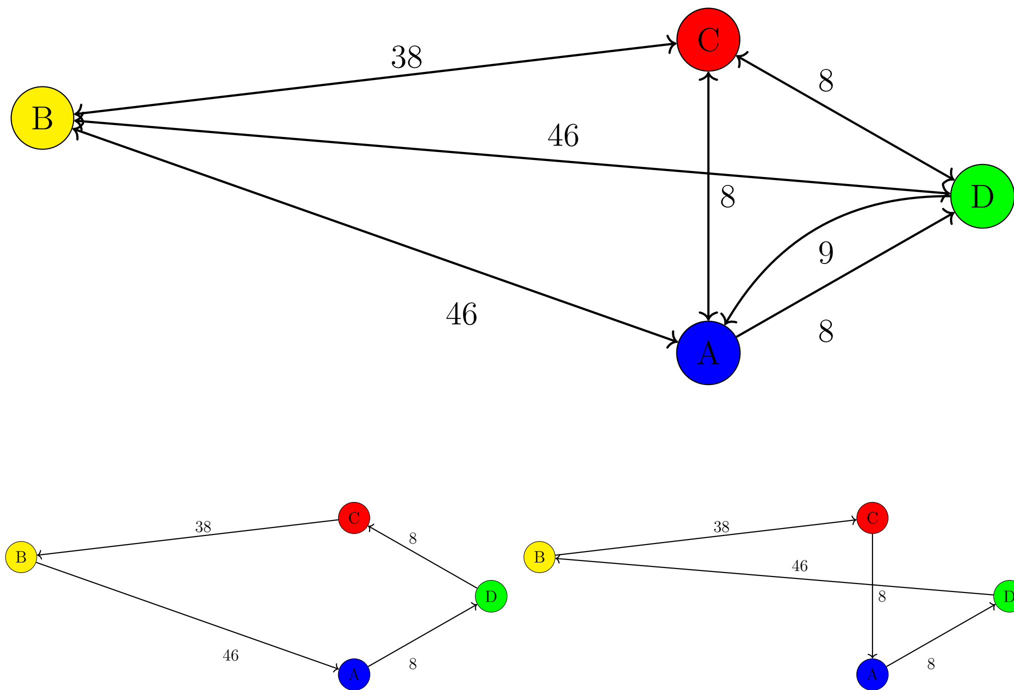


Figure 6.2: Instance graph with travel distance T (at the top) and possible tours with cost 100 (at the bottom)

visit	dom(next)	dom(prev)
D_d	{B,C}	
A	$\{D_a\}$	{B,C}
B	{A,C}	$\{C, D_d\}$
C	{A,B}	$\{B, D_d\}$
D_a		{A}

Table 6.1: Domains of variables *next* and *prev*

In what follows we use $dom(a)$ to refer to the domain of a variable a, $\underline{a}$ for the smallest value in its domain and $\bar{a}$ for the biggest. If a is fixed then $\underline{a} = \bar{a}$ and $dom(a)$ is a singleton.

If time bounds are computed using only $prev/next$ variables, the best we can do to exploit the constraints on time (6.17) and (6.18) (adapted from (6.13) and (6.14) for the non-time-dependent case)

$$time[next[i]] \geq time[i] + T[i][next[i]] \tag{6.17}$$

$$time[i] \geq time[prev[i]] + T[prev[i]][i] \tag{6.18}$$

boils down to the following formulas to compute $\underline{time}[X]$, the minimum value in the domain of $time[X]$. Those formulas have to be calculated until a fixed point is reached (meaning that $\underline{time}[X]$ stays the same for all possible nodes $X \in \{A, B, C, D_a\}$). The departure time $time[D_d]$ is fixed to 0 in the model so it doesn't need to be taken into consideration here.

$$\underline{time[A]} = \max(\underline{time[A]}, \min(\underline{time[B]} + T[B][A], \underline{time[C]} + T[C][A])) \tag{6.19}$$

$$\underline{time[B]} = \max(\underline{time[B]}, \min(\underline{time[C]} + T[C][B], \underline{time[D_d]} + T[D_d][B])) \tag{6.20}$$

$$\underline{time[C]} = \max(\underline{time[C]}, \min(\underline{time[B]} + T[B][C], \underline{time[D_d]} + T[D_d][C])) \tag{6.21}$$

$$\underline{time[D_a]} = \max(\underline{time[D_a]}, \min(\underline{time[A]} + T[A][D_a])) \tag{6.22}$$

For node A, for example, $\underline{time[A]}$ is updated using (6.19): the maximum between the current domain lower bound $\underline{time[A]}$ and a possible better bound, calculated from taking the minimum possible arrival time when coming from the nodes in the domain of $prev[A]$. In this case, the fixed point is achieved in the first iteration and gives $\underline{time[A]} = 16$, $\underline{time[B]} = 46$ and $\underline{time[C]} = 8$.

Let's now look at how we could propagate by also considering (indirect) successors. The distances in matrix T satisfy the triangle inequality[2] so calculating the cost of the path (D, A, B, D) gives a lower bound of all possible tours where A comes before B (and start and finish at the depot). Let us suppose then that A comes before B i.e., that B is a successor of A. In this case,

[2]If the triangle inequality is not satisfied, one can easily pre-compute a smaller transition time corresponding to the length of the shortest path (using Floyd-Warshall algorithm) to provide a lower bound on travel times. That is what the `NoOverlap-TDTSP` constraint of CP Optimizer is doing internally.

we can calculate a lower bound of the optimal tour by calculating the cost of (D, A, B, D): $c(D, A, B, D) = T[D][A] + T[A][B] + T[B][D] = 101$. Since this lower bound is already higher than the current objective upper-bound (100) then B must be visited before A and we can infer a precedence $B \rightarrow A$. This corresponds to the so-called disjunctive constraint in scheduling. With this, we know that A cannot start before $\underline{time[A]} = T[D][B] + T[B][A] = 92$, which is a lot better than the value of $\underline{time[A]} = 16$ found when reasoning only with *prev* and *next* variables.

If the TSP was purely symmetric it would not have been possible to deduce any successor links (indirect precedence) since any solution would be reversible and give the same cost. This type of reasoning is interesting as soon as solutions are asymmetric, which is usually the case for time-dependent travel times.

6.3 A scheduling model for the (TD)TSP

In order to integrate this kind of reasoning we use the concepts of *interval* and *sequence variables* in CP Optimizer, as described in Section 2.4. Each visit i is modeled as an interval variable denoted $visit[i]$, this variable contains the start and end time of each visit which can be accessed with the integer expressions *startOf*() and *endOf*() and also its *type*, which in this case corresponds to the visit's location. It is thanks to *type* that we can create two *visit* variables at the same location (in this case, the depot). The tour is modeled as a sequence variable over the set of visits called *tour*. The start time of the next visit in the sequence can be accessed as a variable in the model with the syntax *startOfNext*(). The sequence variable maintains a precedence graph to propagate temporal relations between visits (see 2.4.1) and the vertices of this graph are the *visit* variables associated to each visit.

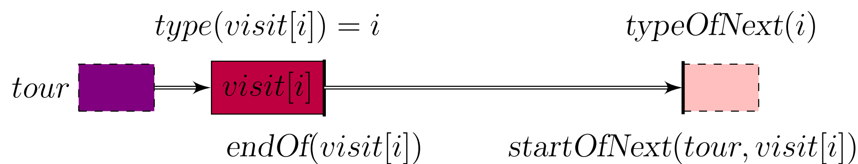


Figure 6.3: Illustration of the scheduling model's variables

In CP Optimizer the `NoOverlap-TDTSP` constraint is used to enforce a **minimal** transition time between vertices on the precedence graph. The `NoOverlap-TDTSP` constraint has different kinds of propagation between variables (vertices) in the graph given the type of arc connecting them (next or

successor). The difference in propagation is further explained in the following chapter. Since the transition times (costs) matrix T is time dependent and the `NoOverlap-TDTSP` constraint only takes constant transition times as input, we need to calculate the minimum value that T can assume for every pair of vertices. We define matrix $minT : [1, n+1] \times [1, n+1] \rightarrow \mathbb{R}^+$ as the minimum travel time from vertex i to vertex j: $min_{k \in 0,..,m} T[i][j][k]$. The scheduling model for the TDTSP using the `NoOverlap-TDTSP` constraint is as follows:

$$
\begin{aligned}
\text{intervalVar} \quad & visit[i \in 1..n+1] \, \text{size} \, D[i] \\
\text{sequenceVar} \quad & tour \, \text{in all}(i \in 1..n+1) \; visit[i], \\
& for(i \in 1..n) \; type(visit[i]) = i, \\
& type(visit[n+1]) = 1 \\
\text{minimize} \quad & startOf(visit[n+1]) \\
\text{subject to} \quad & \text{first}(tour, visit[1]) \\
& \text{last}(tour, visit[n+1]) \\
& NoOverlap(tour, minT) && (6.23) \\
\forall i \in 1..n : \quad & step[i] = endOf(visit[i])/l && (6.24) \\
\forall i \in 1..n : \quad & startOfNext(tour, visit[i]) \geq \\
& endOf(visit[i]) + T[i][typeOfNext(i)][step[i]] && (6.25)
\end{aligned}
$$

Constraint (6.24) is only a simplification of notation so that constraint (6.25) is more readable. Constraint (6.25) is equivalent to constraint (6.14) used in the time-dependent extension of the classical CP model for the TSP. This constraint is used to guarantee that time-dependent transition times between visits are respected. While `NoOverlap-TDTSP` connects information on sequence (successors) with time information (though only lower bounds, minimal transition times). It should also be noted that, in the real model, the expression $T[i][typeOfNext(i)][step[i]]$ in constraint (6.25) cannot be written straight-forwardly like this, as the element constraint in CP Optimizer does not support arrays with more than one index being a variable. But the same thing can be expressed by generating a two-dimensional transition times matrix $T2D$, such that $T2D[i][j + t * n] = T[i][j][t]$ and replacing constraint (6.25) by:

$$
\begin{aligned}
\forall i \in 1..n : \quad & startOfNext(tour, visit[i]) \geq \\
& endOf(visit[i]) + T2D[i][typeOfNext(i) + step[i] * n])
\end{aligned}
$$

This model is a lot shorter than the previous one presented in this chapter, this is mainly due to the power of expression and complexity level of variables and constraints in this scheduling model. In the results chapter (8), a comparison of propagation and efficiency of the different models is performed and we can see the role of having a more global reasoning while modeling and solving.

Let's see how this model would propagate on the example described in Section 6.2. Here the travel times are time-**independent** so they are entirely captured by the minimal travel time $minT$ (in this particular context, constraints (6.24) and (6.25) are useless). The *NoOverlap* constraint would build the transition distance between pairs of successors (called $ttSuccessor[i, j]$ here) in the precedence graph. As the transition time already satisfies the triangle inequality, matrix $minT$ can directly be used as transition distance between successors ($ttSuccessor[i, j] = minT[i][j]$) and since visits durations are not taken into account in the example the start and end of any visit are the same. Using this matrix, the disjunctive constraint would discover that B cannot be a successor of A, because: if A came before B then B would have to start at $startMin(B) = 55$ ($= endMin(A) + ttSuccessor[A, B] = 9 + 46$) or after, but B cannot finish after $endMax(B) = 54$ ($= maxTourCost - ttSuccessor[B, D] = 100 - 46$). The propagation would then add the successor relation $B \rightarrow A$ into the precedence graph, leading to the propagation on the visit times described in the end of Section 6.2.

6.4 Discussion

In this chapter we start by extending the classical CP model for the TSP to model the TDTSP and we also present extra variables and constraints meant to strengthen the level of propagation of this model. In the second section we present the limitations of the extended version of the classical model, namely:

1. it does not make use of precedence relations other than next/previous - indirect successors are not taken into account for propagation
2. it has weak propagation between sequence related variables and time related variables.

In the third section we give a scheduling model that addresses the first limitation but not the second. To tackle the second limitation we extended the *NoOverlap* constraint to be able to take time dependent transition times into account. The new constraint obtained from this extension is named *TDNoOverlap* and is presented in details in the following chapter.

Chapter 7

The *TDNoOverlap* constraint

In the previous chapter we saw that the *NoOverlap* constraint allows to enforce a minimal (constant) transition time between vertices on the precedence graph. In this chapter, we extend *NoOverlap* into a *TDNoOverlap* constraint to take into account time-dependent transition times. In section 7.1, the model using the new constraint is presented and, in the following sections, details about the propagation and implementation of the constraint are given.

7.1 CP model with *TDNoOverlap*

The *TDNoOverlap* constraint is an extension of *NoOverlap* to time-dependent costs. It has two parameters: a sequence variable, that contains the interval variables that must not overlap, and a time-dependent cost function, that defines the time-dependent transition costs between interval variables.

The scheduling model sketched below uses the new *TDNoOverlap* constraint (7.1) instead of constraints from 6.23 to 6.25, used in the scheduling model presented in the previous chapter. The new model becomes very short as the *TDNoOverlap* constraint captures the global structure of the problem.

$$
\begin{array}{rl}
\text{intervalVar} & visit[i \in 1..n+1]\,\text{size}\,D[i] \\
\text{sequenceVar} & tour\,\text{in all}(i \in 1..n)\ visit[i] \\
\text{minimize} & \mathit{startOf}(visit[n+1]) \\
\text{subject to} & \text{first}(tour, visit[1]) \\
 & \text{last}(tour, visit[n+1]) \\
 & \mathit{TDNoOverlap}(tour, T) \qquad (7.1)
\end{array}
$$

7.2 Propagation of *TDNoOverlap*

In the same way as for *NoOverlap*, the *TDNoOverlap* constraint operates on a sequence variable (an ordered list of interval variables) which maintains a precedence graph with next and successor types of arc between interval variables. To propagate the bounds of the interval variables' domains we need lower bound functions for the time-dependent transition time functions. Given two interval variables i and j, such that there exists a next arc or a successor arc going from i to j, we define two lower bound functions for each type of arc (these functions are precisely described in the following sections):

1. $f^{next}_{earliest}(i, j, t_d)$ and $f^{succ}_{earliest}(i, j, t_d)$ are the transition times such that **the arrival time at** j when leaving i at time t_d **is the earliest** possible.

2. $f^{next}_{latest}(i, j, t_a)$ and $f^{succ}_{latest}(i, j, t_a)$ are the transition times such that **the departure time from** i **is the latest** possible to allow arriving at j at time t_a at the latest.

To propagate the earliest time for j (as in Fig. 7.1), *TDNoOverlap* tightens the lower bound of $time[j]$ so that it is greater than the lower bound of $time[i]$ plus the duration of i and the smallest transition time from i to j, for both types or arc, *next* (Eq. 7.2) and *succ* (Eq. 7.3):

$$\underline{time[j]} \geq \underline{time[i]} + D[i] + f^{next}_{earliest}(i, j, \underline{time[i]} + D[i]) \qquad (7.2)$$

$$\underline{time[j]} \geq \underline{time[i]} + D[i] + f^{succ}_{earliest}(i, j, \underline{time[i]} + D[i]) \qquad (7.3)$$

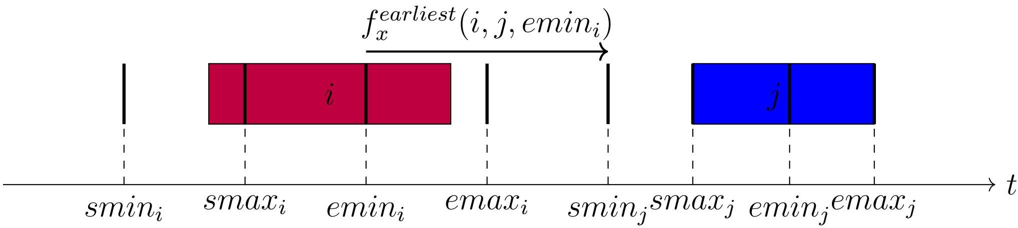


Figure 7.1: Propagation of the earliest start of j given the earliest end of i

To propagate the latest time for i (as in Fig. 7.2), *TDNoOverlap* tightens the lower bound of $time[j]$ so that it is greater than the lower bound of $time[i]$ plus the duration of i and the smallest transition time from i to j, for both types of arc, *next* (Eq. 7.4) and *succ* (Eq. 7.5):

$$\overline{time[i]} + D[i] \leq \overline{time[j]} - f^{next}_{latest}(i, j, \overline{time[j]}) \tag{7.4}$$

$$\overline{time[i]} + D[i] \leq \overline{time[j]} - f^{succ}_{latest}(i, j, \overline{time[j]}) \tag{7.5}$$

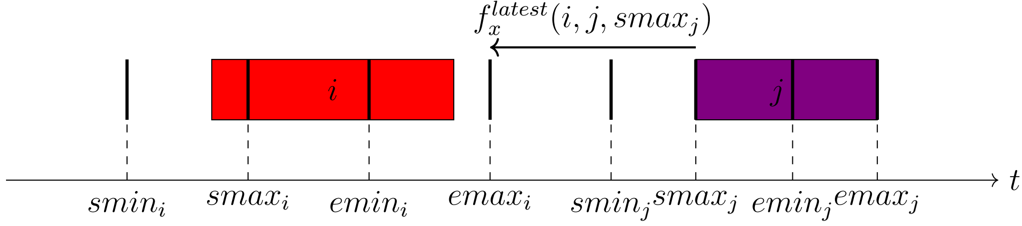


Figure 7.2: Propagation of the latest time end of i given the latest start of j

Now we introduce the formal definitions of the bounding functions and explain how to calculate them.

7.2.1 Computation of $f^{next}_{earliest}$

Here we consider a next arc $i \Rightarrow j$ in the precedence graph, i.e., interval j is just after interval i in the sequence variable. The earliest arrival time at j, if we leave i at time t_d, is defined using the transition time function:

$$f^{next}_{earliest}(i, j, t_d) = \min_{t \geq t_d}\{f(i, j, t) + t - t_d\} \tag{7.6}$$

In $f^{next}_{earliest}$ we check if leaving from visit i later (waiting in place) allows to arrive at j sooner. If the transition times satisfy the FIFO property, waiting is never advantageous.

It follows from Def. 9 and from Eq. (7.6) that if f satisfies the FIFO property then $f^{next}_{earliest}$ and f are equal. Although the FIFO property generally holds in practice, our approach does not assume that f satisfies the FIFO property for three reasons: (1) stepwise functions do not satisfy it because of the discretization, (2) imprecision in data acquisition and time-dependent travel time calculations may introduce non-FIFO effects and (3) the constraint has to be general enough to take other applications into account and the FIFO property is specific to network routing.

If f is a stepwise or a piecewise linear function, $f^{next}_{earliest}$ is a piecewise linear function. So, as we need in any case to handle piecewise linear functions in the propagation, in our implementation of the *TDNoOverlap* constraint we

decided to treat the more general case where the input function f is a piecewise linear function. In Algorithm 5 (given in section 5.1) we describe the method used in pre-solve phase to calculate $f^{next}_{earliest}$ for each pair of vertices in the graph. This algorithm transforms general piecewise linear functions into functions respecting the FIFO property.

7.2.2 Computation of f^{next}_{latest}

Like in 7.2.1, we consider a next arc $i \Rightarrow j$ in the precedence graph, and we define the latest departure time from i in order to arrive at j at time t_a or before as follows:

$$f^{next}_{latest}(i,j,t_a) = \min_{t+f(i,j,t)\leq t_a} \{t_a - t\} \tag{7.7}$$

Since $f^{next}_{earliest}$ already gives us the minimum transition time from a given time it is clear that the minimum in Equation (7.7) is satisfied for the biggest t' such that $t' + f^{next}_{earliest}(i,j,t') \leq t_a$. Then, calculating f^{next}_{latest} comes down to finding this t'.

Algorithm 7 describes the method used in a presolve phase to compute f^{next}_{latest} for a couple (i,j) of interval variables. In the algorithm, we call $arrivalTime$ the arrival time function associated with $f^{next}_{earliest}$. The main idea is to calculate the inverse function of $arrivalTime$ ($arrivalTime^{-1}$) and to subtract it from the identity function to get the corresponding transition times.

This is done by "inverting" piece by piece of the function in such a way that we can easily address the problem that the inverse of $arrivalTime$ is not immediately a function. This happens because $arrivalTime$ is not necessarily continuous or strictly increasing (but it is always non-decreasing) and therefore the image of a given t under $arrivalTime^{-1}$ can be empty or have more than one value. When the image has more than one value the algorithm takes only the maximum value (line 8) and if it is empty the previous value assumed by the inverse function is taken (line 9).

We suppose a fixed arc (i,j) and f a piecewise linear function defined on the time domain $T = [t_{Min}, t_{Max}) \in \mathbb{R}$ and we use the notations defined in section 5.1.In the implementation, we used the class of piece-wise linear functions provided by CP Optimizer[1]. If ν is the number of pieces of the function, this class allows for a random access to a given piece with an average complexity of $O(log(\nu))$. Furthermore, when two consecutive pieces of the

[1]Namely, `IloNumToNumSegmentFunction`.

function are co-linear, these pieces are automatically merged so that the function is always represented with the minimal number of pieces.

Algorithm 7 Calculate f^{next}_{latest}

Input: a piecewise linear FIFO function $f^{next}_{earliest}$, composed of ν linear pieces such that for each piece $k \in [1, \nu], p_k = [t^k_{min}, t^k_{max}[$ is the kth time interval on which $f^{next}_{earliest}$ is linear.

Output: a piecewise linear function f^{next}_{latest}

1: $arrivalTime \leftarrow linear((0,0),(1,1)) + f^{next}_{earliest}$
2: $f^{next}_{latest}(t) \leftarrow 0$
3: **for all** $k \in \{0, ..., \nu\}$ **do**
4: $x_0 \leftarrow t^k_{min}$
5: $x_1 \leftarrow t^k_{max}$
6: $v_0 \leftarrow arrivalTime(x_0)$
7: $v_1 \leftarrow arrivalTime(x_1)$
8: $f^{next}_{latest} \upharpoonleft_{[v_0,v_1)} \leftarrow \max(f^{next}_{latest}, linear((v_0, x_0), (v_1, x_1))) \upharpoonleft_{[v_0,v1)}$
9: $f^{next}_{latest} \upharpoonleft_{[v_1,+\infty)} \leftarrow \max(f^{next}_{latest}, x_1) \upharpoonleft_{[v_1,+\infty)}$
10: **end for**
11: $f^{next}_{latest} \leftarrow linear((0,0),(1,1)) - f^{next}_{latest}$
12: **return** f^{next}_{latest}

7.2.3 Computation of $f^{succ}_{earliest}$

Now we consider a successor arc $i \rightarrow j$ in the precedence graph, i.e., the interval variable j occurs after the interval variable i in the sequence. To estimate the earliest possible time of arrival at j if we leave i at time t_d or after we have to check if we can arrive faster at j by passing through other vertices. Let $\wp^{i,j}_{\tau,f}$ be the set of all timed-paths from i to j starting after time τ with travel time function f. We have:

$$f^{succ}_{earliest}(i, j, t_d) = \min_{p \in \wp^{i,j}_{t_d,f}} t(j, p) - t_d$$

where $t(j, p)$ is the start time of j in path p.

If there exists a shortest path from i to j, shorter than the direct arc, then the triangular inequality extended to the time-dependent case (as in Def. 14 in section 5.2) does not hold. It means that there is at least one vertex k such that passing through k allows to arrive faster at j. The algorithm we use to calculate $f^{succ}_{earliest}$ is Algorithm 6, given in section 5.2, which calculates time-dependent travel times respecting the time-dependent triangular inequality.

We use $f^{next}_{earliest}$ as input travel time function to the algorithm so that waiting at intermediate vertices to possibly go faster is already taken into account.

7.2.4 Computation of f^{succ}_{latest}

The second type of propagation on successor arcs is based on the estimation of the latest departure time from i in order to arrive at j at time t_a or before, given by:

$$f^{succ}_{latest}(i, j, t_a) = \min_{p \in \wp^{i,j}_{t,f}, t(j,p) \leq t_a} t_a - t$$

The reasoning for calculating f^{succ}_{latest} is exactly the same as the one used for f^{next}_{latest} and the algorithm is the same too (Algorithm 7), except that we use as input $f^{succ}_{earliest}$ instead of $f^{next}_{earliest}$.

7.2.5 Time-dependent disjunctive propagation

Classical propagation algorithms used in constrained-based scheduling can be extended to time-dependent transition times. In our implementation of the *TDNoOverlap* constraint we extended the disjunctive reasoning [12]. As soon as two visits i and j are such that one of the conditions below is satisfied then it is clear that it is not possible to visit j before i and thus, we can add a successor arc $i \rightarrow j$ in the precedence graph:

$$\underline{time[j]} + D[j] + f^{succ}_{earliest}(j, i, \underline{time[j]} + D[j]) > \overline{time[i]}$$

$$\overline{time[i]} - f^{succ}_{latest}(j, i, \overline{time[i]}) - D[j] < \underline{time[j]}$$

This extended disjunctive reasoning helps discovering new arcs in the precedence graph that are themselves propagated as described in subsection 7.2.1.

7.3 Implementation and complexity

Propagation can originate from an event on an interval (in the model, the variables *visit*) or on the sequence (*tour*). Changes in the domain of a certain interval have to be propagated on its neighbors in the precedence graph and new arcs on the graph cause domains of related intervals to be propagated. For a given interval variable i, we note: $smin_i$ its minimum possible start (at a given point during search), $smax_i$ its maximum possible start, $emin_i$ its minimum possible end, $emax_i$ its maximum possible end.

This values are related as in Figure 7.3. Furthermore, since durations of intervals (dur_i) are fixed in this problem, the relations between start and end bounds $emin_i = smin_i + dur_i$ and $emax_i = smax_i + dur_i$ hold. We say that (i, j) is an outgoing arc for i and an incoming arc for j to address arcs generally, without having to specify whether it is a next or successor arc.

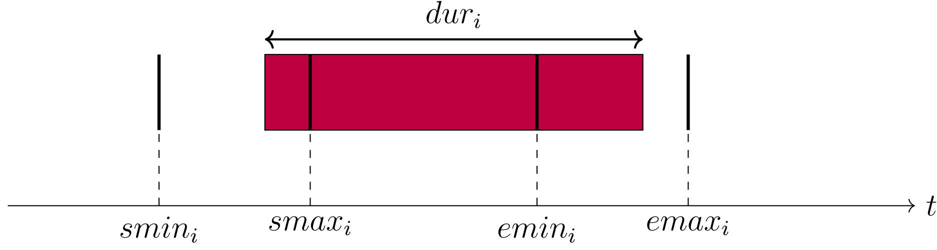


Figure 7.3: Bounds of the domain of interval variable i

In what follows we list the three main types of events and how they are propagated, the disjunctive propagation is treated separately afterwards.

1. Change in the domain of an interval i:

   - Propagate $emin_i$, for each outgoing arc (i, j) (of type $X \in \{next, succ\}$), set $smin_j = emin_i + f^X_{earliest}(i, j, emin_i)$

   - Propagate $smax_i$, for each incoming arc (j, i) (of type $X \in \{next, succ\}$), set $emax_j = smax_i - f^X_{latest}(j, i, smax_i)$

2. New successor arc (i, j):

   - Propagate $emin_i$, set $smin_j = emin_i + f^{succ}_{earliest}(i, j, emin_i)$

   - Propagate $smax_j$, set $emax_i = smax_j - f^{succ}_{latest}(i, j, smax_j)$

3. New next arc (i, j):

   - Propagate $emin_i$, set $smin_j = emin_i + f^{next}_{earliest}(i, j, emin_i)$

   - Propagate $smax_j$, set $emax_i = smax_j - f^{next}_{latest}(i, j, smax_j)$

Disjunctive propagation When the domain of an interval variable i in the precedence graph is modified, the following conditions are tested for every incoming arc (j, i) or outgoing arc (i, j) such that i and j are not simultaneously successors of each other (an iterator is available for this):

- if $emin_j + f^{succ}_{earliest}(j, i, emin_j) > smax_i$ then the successor arc $i \to j$ is added to the precedence graph

- if $smax_j - f^{succ}_{latest}(i, j, smax_j) < emin_j$ then the successor arc $j \to i$ is added to the precedence graph

From an implementation point of view, one of the most critical functions in the propagation is the computation of the values of functions $f^X_Y(i, j, t)$, with $X \in \{next, succ\}$ and $Y \in \{earliest, latest\}$, for a given pair (i, j) and a given time value t. These functions are computed once, before the beginning of the search, but they are accessed very often. In our implementation, piecewise linear functions are implemented using skip lists to permit a random access to $f^X_Y(i, j, t)$ in $O(log(\nu))$ in average, where n is the number of segments of the function [2]. We think that the performance could be much improved by exploiting some support or cache to the last position of the function accessed for a given pair (i, j).

The complexity of the *TDNoOverlap* constraint is dominated by the complexity of maintaining the precedence graph and the disjunctive propagation. The worst-case complexity of the full-fledged propagation is quadratic with respect to the number of interval variables (i.e., visits in our application).

[2]See documentation of class *IloNumToNumSegmentFunction* in CPLEX Optimization Studio [1]

Chapter 8

Experimental evaluation

This chapter is divided into five sections, as follows. The first section introduces preliminary concepts and notation necessary for the chapter, in the following section the experimental setup is given in terms of models used for experiments, selection of benchmark instances and hardware. The third section studies the interests of taking time-dependency into account for the optimization of delivery sequences by comparing the performance of TSP and TDTSP solutions performed under the same traffic conditions. Section 8.4 compares the model using *TDNoOverlap* with the other CP models presented in this thesis and studies their scale up behavior. In the last section, we present the only other set of TDTSP instances available online and briefly compare *TDNoOverlap* with their method on those instances.

8.1 Comparison of TDTSP and TSP solutions

For the comparisons performed in this chapter, some concepts and notations should be explained beforehand for better comprehension. Mainly in what concerns the comparison of solutions for the TSP and solutions for the TDTSP.

Generating a TSP instance from a TDTSP instance Starting from a TDTSP instance, with time-dependent travel time function T, we calculate for each pair of visits (i, j) the **median travel time** over the set of travel times between i and j for all time steps of the optimization horizon. This gives us a constant travel time function T_{Median} which, along with the set of visits, defines a TSP instance. The choice of median travel times was made to generate TSP solutions that reflect real conditions to a larger degree than simply taking minimum, maximum or average travel times occurring

during the day. The quality of TSP solutions certainly varies according to the constant travel times chosen but here only the case with median travel times was studied.

The method chosen to find an optimal solution for the TSP using T_{Median} travel times was the scheduling model presented in section 6.3 without constraint 6.25 (the one enforcing time-dependent transition times). We call this model `NoOverlap-TSP` and it was the model performing the best for the TSP, among the CP models presented here.

Comparing objective values of TSP and TDTSP solutions The following notation is introduced:

- opt_{TSP}, the optimal sequence found by the TSP model (using T_{Median})
- opt_{TDTSP}, the optimal sequence found by the TDTSP model (using T)
- $obj_{T_{Median}}(opt_{TSP})$, the optimal objective value of the TSP (using T_{Median})
- $obj_T(opt_{TSP})$, the objective value of opt_{TSP} evaluated using time-dependent travel times function T
- $obj_T(opt_{TDTSP})$, the optimal objective value of the TDTSP

The relationship between $obj_{T_{Median}}(opt_{TSP})$ and $obj_T(opt_{TSP})$ is not fixed, meaning that either one can produce a longer total travel time. On the other hand, $obj_T(opt_TSP)$ is an upper bound of $obj_T(opt_TDTSP)$, i.e., $obj_T(opt_{TSP}) \geq obj_T(opt_{TDTSP})$ is always valid, otherwise opt_{TDTSP} would not be the optimal solution of the TDTSP. Figure 8.1 gives an example of a TSP solution evaluated with T_{Median} (the costs used to find opt_{TSP}) and the same sequence evaluated in a time-dependent context, using T.

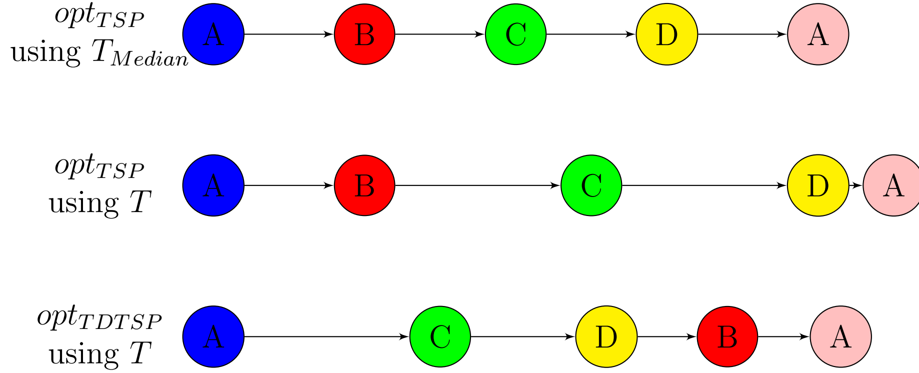


Figure 8.1: Example of TSP optimal solution evaluated using constant (median) travel times on the top and using time-dependent travel times in the middle. On the bottom, the optimal TDTSP solution

Relative gain of time-dependency The difference between $obj_T(opt_{TSP})$ and $obj_T(opt_{TDTSP})$ allows us to quantify the time-dependency of an instance: the bigger the difference, the more time-dependent the instance is. We define the relative gain (in terms of objective value) as:

$$relative_gain = \frac{obj_T(opt_{TSP}) - obj_T(opt_{TDTSP})}{obj_T(opt_{TDTSP})}$$

The relative gain measures how much travel time, as percentage of the total travel time of opt_{TDTSP}, can be saved by considering time-dependent data when optimizing the tour.

8.2 Experimental setup

List of models The following models are used for the experiments presented in this chapter:

1. `Classic-TDTSP` model for the TDTSP, the time-dependent extension of the classical CP model for the TSP, presented in section 6.1.

2. `NoOverlap-TDTSP` model for the TDTSP, the scheduling model using the *NoOverlap* constraint described in section 6.3.

3. `NoOverlap-TSP` model for the TSP, a constant version of `NoOverlap-TDTSP` using T_{Median}.

4. `TDNoOverlap-TDTSP` model for the TDTSP, the model using the *TDNoOverlap* constraint as shown in section 7.1.

Instances selection To simplify the benchmark described in section 4.3, 20 instances are picked (out of the 500 randomly generated) for each size $\{10, 20, 30, 50, 100\}$. To do so we first analyze how the different instances represent the impacts of taking time-dependency into account. The set of instances with time-window constraints was generated after this selection process, therefore, this section concerns only instances without time-windows. The procedure described in section 4.3 to generate instances with time-windows was performed using as input the smaller set of instances selected hereafter.

One can imagine that in certain cases time-dependency might not change the optimal solution of the problem, for example: if the area concerned by the instance's visits is not heavily affected by traffic; if all paths between every two visits are affected in the same way; if there is only one obvious solution as in the case of visits aligned sequentially in the same street. In those cases solutions can be completely time-independent even if we use time-dependent data for optimization. On the other extreme of the spectrum we have situations that are so time-dependent that solutions calculated not taking time-dependency into account can give: unreliable delivery times; optimistic estimations of total travel time, sometimes leading to extra working hours to the drivers; or unfeasible schedules, if time-windows are imposed. In order to select 20 instances that are representative of these different possible behaviors (very time-dependent, not time-dependent at all), we proceed to study the degree of time-dependency of each instance.

Ideally, to be able to calculate accurate relative gains and therefore, accurate degrees of time-dependency of our instances, optimal solutions (opt_{TSP} and opt_{TDTSP}) would have to be used. The problem is that, for instances bigger than 20 visits, optimality is hardly ever proved by our CP models (both for the TSP and the TDTSP). So we simply used the best solutions obtained in a given time limit and estimated the relative gains from them. To look for the best solution in the time limit, the models `NoOverlap-TSP` and `TDNoOverlap-TDTSP` are run on restart mode. For sizes 10 to 30 a time limit of 30 minutes was used, for 50 and 100, the limit was 1 hour. As it will be shown in section 8.4, the model for the TSP is lighter, in terms of constraints to propagate, and has a tendency of converging faster than the TDTSP model at the beginning of the search. This means that occasionally, solutions found by `NoOverlap-TSP` had better obj_T value than those found by `TDNoOverlap-TDTSP` (since optimality of neither was guaranteed), giving negative relative gains.

For each problem size n, a smaller set of 20 instances was selected, to be representative of the different types of estimated gains between TSP and TDTSP. These instances were picked randomly within three main groups of

estimated gains (largest, intermediate, negligible/zero/negative) with equal probability for each group. Also, as can be seen in Fig. 8.2, gains increase when traffic variations are more important during the day (represented by the different matrices $T00$, $T10$ and $T20$). Therefore, to avoid conflicts in the classification of instances into one of the three categories of gains, they were classified according to their performance when using $T10$ as travel times.

In Fig. 8.2 we can see the cactus plot of the relative gains for all 500 instances of size 10. This figure shows that for more than 10% of the $T00$ instances, the gain is greater than 5%, whereas for 40% of the instances it is equal to 0%. Note that a gain of 5% is considered as very important in our context. Furthermore, real-world delivery problems usually have time-window constraints in which case it is mandatory to consider time-dependent data in order to have reliable results, as shown in the next subsection. As expected, the gain tends to increase when using functions $T10$ and $T20$ with larger amplitude of traffic variation (to more than 13% and 21%, respectively).

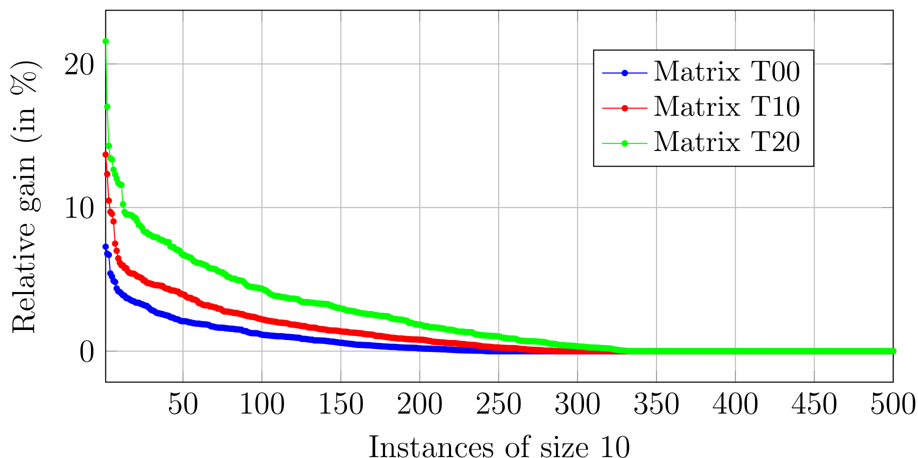


Figure 8.2: Relative gain of TDTSP, instances ordered by decreasing gain

A similar behavior was observed for larger problems with 20 and 30 visits although the gain was slightly smaller, going up to 9% for size 20 and to around 6% for sizes 30, 50 and 100, with some negative results for 50 and 100 since for those sizes optimality is never proved. Those lower gains are probably due to the fact that the peaks of traffic congestion occur between 06:00 and 09:00, which more or less corresponds to the time frame of a 10-visit problem so, for larger problems, part of the route is executed on less congested moments.

The results presented in the sequel of this chapter were obtained by running the tests on an IBM Blade HS22 Type 7870 with an Intel Xeon X5570 (2.93GHz/1333MHz/8MB Cache) processor and 20GB of RAM. For our benchmark we consider an **instance class** to be defined by the number of visits i ($\in \{10, 20, 30, 50, 100\}$), the type of matrix Tj (with $j \in \{00, 10, 20\}$) and the presence or not of time-windows which is indicated by the suffix "TW". The name of an instance class is of the form $i_Tj(_TW)$ and each of the 30 classes $(5*3*2)$ has 20 instances, which gives a total of 600 instances. Reference solutions (best objective values, not sequences) for each of these instances are given online (in the same page as the benchmark instances) and were obtained by letting the `TDNoOverlap-TDTSP` model run for 2 hours on restart mode. Solutions proved to be optimal are marked with $*$.

8.3 Evaluation of the effects of taking time-dependency into account

In this section a comparison similar to the one done in the previous section is made. Time-dependent travel times are used to simulate how solutions found by a TSP model (using median cost values) would behave in "real" conditions. A solution for the TSP consists in a sequence of visits and hereafter, a TSP solution is said **feasible** if it still respects the problem's constraints when time-dependent travel times are applied. Since every sequence is a (TD)TSP solution the only way in which it can become unfeasible with different travel times is if it no longer satisfies the problem's **side-constraints**, in this case, the time-window constraints. A solution with an objective value greater than the function time-span of 13 hours is considered infeasible, this can also be seen as a time-window at the depot, to ensure that tours respect the driver's work hours.

The effects of time-dependency are studied in two paragraphs, for instances without and with time-windows. For both cases, we compared the solutions found by `NoOverlap-TSP` and `TDNoOverlap-TDTSP`, running in restart mode, with a time limit of 1 hour.

Instances without Time-Windows For instances without time-windows the TSP can always find feasible solutions to the TDTSP instances. To make sure that comparisons are "correct", only instances to which both models are capable of proving optimality were considered (up to size 20, for instances without time-windows). If the compared solutions are not optimal, nothing can be said about the relative gains of the instance, in terms of time-

dependency, since they can even be negative when calculated for non-optimal solutions.

In Table 8.1, one can see that, as expected, taking time-dependency into account reduces total travel time in average and the relative gains between the two seem to grow with the dilatation of travel times, i.e., for the different matrices $T00$, $T10$ and $T20$. Also, as pointed out in the previous section, relative gains tend to decrease from size 10 to 20. The smallest gain of all instances of size 10 is 0 and the biggest is 22.57%. And for instances of size 20, the minimum gain is also 0 but the maximum is only 6.94%, which as mentioned earlier is still considered a quite significant gain in the context of urban deliveries.

Inst Class	Avg Gain	StDev Gain	Min Gain	Max Gain
10_T00	3.00	2.61	0	7.27
10_T10	5.69	4.20	0	13.68
10_T20	10.14	5.03	1.53	22.57
20_T00	1.19	0.97	0	3.03
20_T10	2.24	1.68	0	4.96
20_T20	4.33	1.60	0.80	6.94

Table 8.1: Average (relative) gains, standard deviation, minimum and maximum gains per instance class, in percentage

Instances with Time-Windows For instances with time-windows the TSP model is not always able to find a feasible solution in the time limit of 1 hour. For this reason, in table 8.2 we see a column giving the number of instances solved by the TSP model, where solved means that at least one feasible solution has been found - feasibility (using time-dependent travel times) is checked for every solution found in the search for the optimal TSP solution with model `NoOverlap-TSP`. For the best feasible solution found per instance we calculate the average of their relative gains and standard deviation (when it applies). Optimality is only proved for instances up to 30 vertices with both models. As instance size grows and with the increase of amplitudes in the variations of travel times ($T00 < T10 < T20$) it gets harder for the TSP model to find feasible solutions for the TDTSPTW and even for feasible solutions the average gains are a lot larger than those of instances without time-windows, which shows that in the presence of time-window constraints it is crucial to consider time-dependent travel times in the calculation of delivery routes.

Inst Class	Solved (/20)	AvgGain	StDevGain	MinGain	MaxGain
10_T00_TW	10	19.31	35.99	0	94.96
10_T10_TW	9	20.93	36.18	0	99.01
10_T20_TW	10	21.60	33.56	0	99.24
20_T00_TW	10	12.92	15.91	0	50.01
20_T10_TW	7	20.35	19.49	0	50.23
20_T20_TW	4	7.57	6.77	3.26	17.59
30_T00_TW	10	8.62	18.50	0.34	60.91
30_T10_TW	5	16.89	25.74	0.66	62.43
30_T20_TW	1	5.50	0	5.50	5.50

Table 8.2: Number of instances solved by `NoOverlap-TSP` per instance class (with time-windows). Average gains, standard deviation, minimum and maximum gains given in percentage

8.4 Scaling of the different models

In this section experiments are organized into three subsections each studying how `TDNoOverlap-TDTSP` compares with the other models presented in this thesis. The two first subsections compare the `TDNoOverlap-TDTSP` model with `NoOverlap-TDTSP` and `Classic-TDTSP` models, respectively. The last one analyzes the speed of convergence of `NoOverlap-TSP` and `TDNoOverlap-TDTSP` and proposes a way of accelerating the search for optimal solutions by combining the two.

8.4.1 `NoOverlap-TDTSP` versus `TDNoOverlap-TDTSP`

Different sets of instances were generated along the thesis in order to test the performance (and proper functioning) of `TDNoOverlap-TDTSP` - all based on the same original 255 delivery nodes and time-dependent travel times described in chapter 4. For the comparisons between `NoOverlap-TDTSP` and `TDNoOverlap-TDTSP` a preliminary set of instances was used (different from the one selected in section 8.2). This set contains 10 instances per size (15, 20, 50, 100), it is without time-window constraints, and it is used with travel times matrix $T00$ only as it is previous to the generation of dilated travel times ($T10$ and $T20$).

This comparison is meant to answer how much one can expect to gain in terms of time and also in terms of quality of solution, if the search time is fixed, by integrating the time-dependent travel times directly into the constraint. Or if instead, the overhead of adding time-dependent reasoning

in the propagation of *NoOverlap* would be too much and only slow down the search. For this purpose, average convergence curves were generated for all 4 sizes of instance. A **convergence curve** for a certain instance gives, for each CPU time t, the gap between the values of the current solution (sol_{cur}) (the best solution found by the search up to instant t) and the reference solution (sol_{ref})[1]. The gap is calculated similarly to relative gains:

$$gap = \frac{obj_T(sol_{cur}) - obj_T(sol_{ref})}{obj_T(sol_{ref})}$$

In figures 8.3 and 8.4, the average convergence curves for all sizes are given. For sizes 15 and 20, `TDNoOverlap-TDTSP` is capable of proving optimality for all instances (within 120 seconds) while `NoOverlap-TDTSP` is not even though it comes closer than 5% of the optimal solution within 20 seconds in both cases. For sizes 50 and 100, none of the models is capable of proving optimality but `NoOverlap-TDTSP` is in average 20% further than `TDNoOverlap-TDTSP` from the reference solution for size 50 and 40% further for size 100 (within 180 seconds).

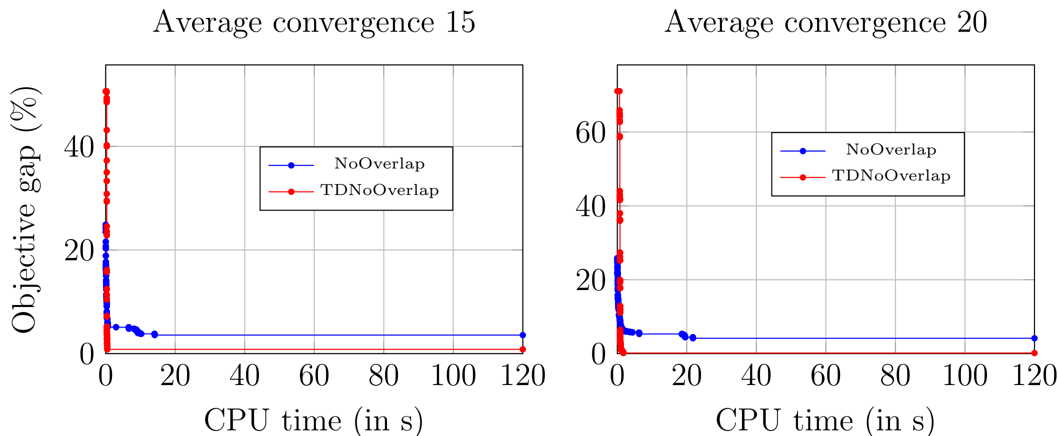


Figure 8.3: Average convergence of `NoOverlap-TDTSP` and `TDNoOverlap-TDTSP` for sizes 15 (left) and 20 (right)

[1]The reference solution is the best solution found in a long running time with the best method, not necessarily proven to optimal

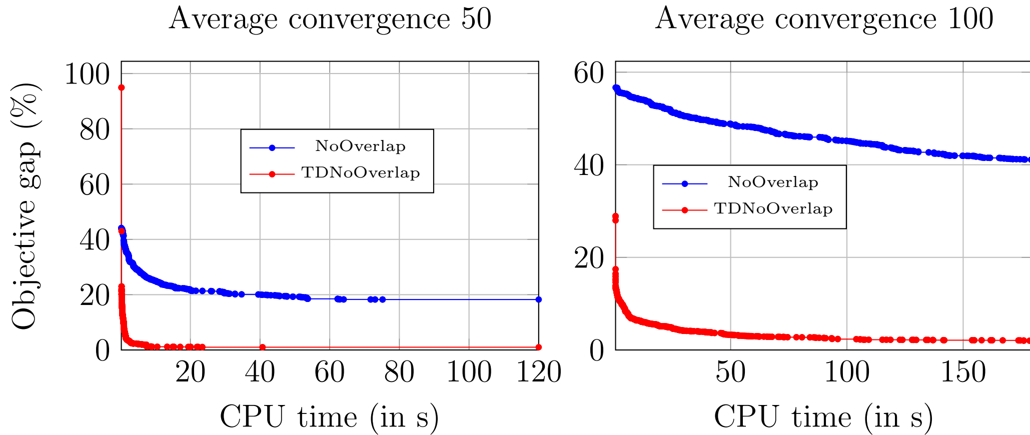


Figure 8.4: Average convergence of `NoOverlap-TDTSP` and `TDNoOverlap-TDTSP` for sizes 50 (left) and 100 (right)

Those results showed very clearly that even with the non-optimized code added for the time-dependent propagation of `TDNoOverlap-TDTSP` the overhead of propagation was more than compensated by the reduction of the search space, in such a way that `TDNoOverlap-TDTSP` improves solutions considerably faster than `NoOverlap-TDTSP`. From this point on we considered that testing with the `NoOverlap-TDTSP` model was unnecessary since `TDNoOverlap-TDTSP` outperforms it.

8.4.2 `Classic-TDTSP` versus `TDNoOverlap-TDTSP`

In chapter 6, we pointed out the limitations of the extended `Classic-TDTSP` model, here we compare the filtering of this model and `TDNoOverlap-TDTSP`. For this purpose, the same depth first search strategy is used for both models so that we can estimate the impact of constraint propagation on the number of branches of the complete search tree. Search heuristics follow a chronological scheduling of visits and choice of the nearest visit in terms of transition time first, given that the earliest date of the previous visit is known. For the `Classic-TDTSP` model, this means that the search first fixes the variables $atPosition[i]$ for $i = 1, 2, ...n$. However, as the search strategy is not static, the search tree is different (one tree is not a sub-tree of the other). We could have tested on a static search strategy but this would have resulted in a more "artificial" type of search. We measured the number of branches and the CPU time of the two models on the 20 instances of size 10 and for each of the 3 functions $T00$, $T10$, $T20$, giving 60 tests at the end.

Comparison was performed only on instances of size 10 because the `Classic-TDTSP` model is not able to solve the larger problems to optimality in a reasonable time. For those 60 tests, all solved to optimality with both models, the left side of Fig. 8.5 shows the comparison of the number of branches of the search tree explored by each approach while the right side compares the CPU times needed to complete the exploration of the search tree. Instances are ordered per function (20 instances with function $T00$ first, then all with $T10$ and finally with $T20$) so one can notice a certain pattern in the difficulty of resolution that is repeated for each function. Figure 8.6 shows the same tests described previously but on instances with time-windows.

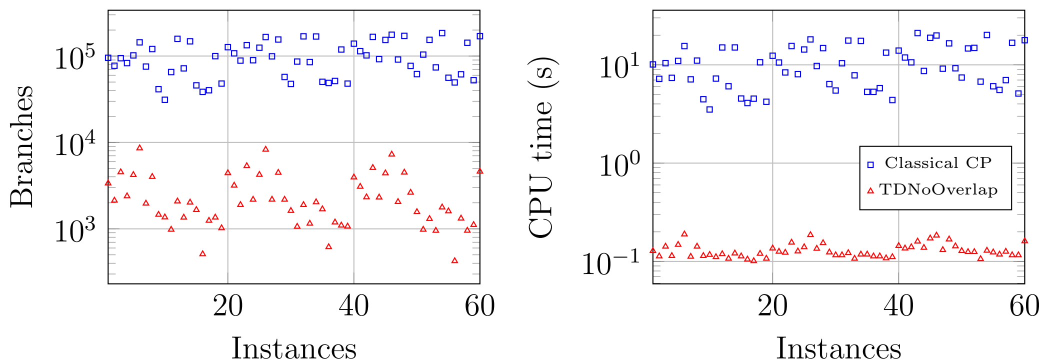


Figure 8.5: Comparison of number of branches (left) and CPU time (right)

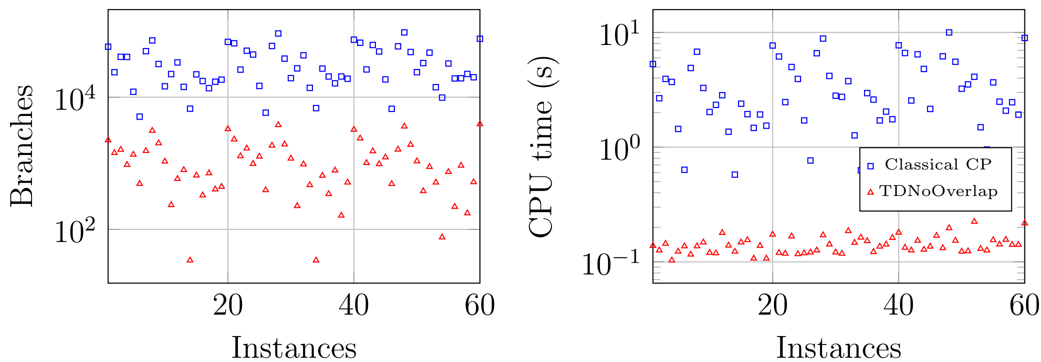


Figure 8.6: Comparison of number of branches (left) and CPU time (right) with Time-Windows

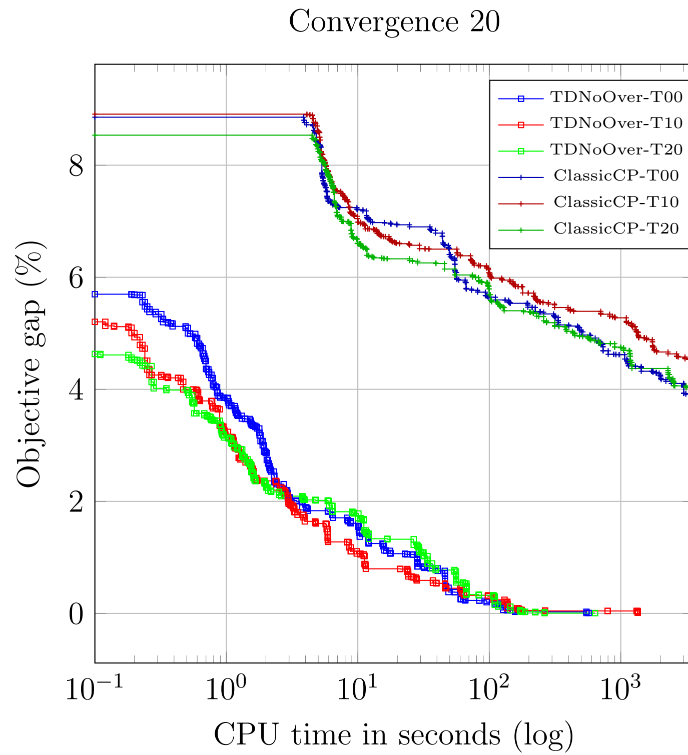
Not only the `TDNoOverlap-TDTSP` model propagates a lot more (about 50 times fewer branches) but it also finds better solutions faster than the

classical CP model. For 10-sized instances the search is about 100 times faster on average for instances without time-windows and more than 10 times faster for instances with time-windows (the problems with time-windows are easier to solve due to reduced search space).

For instances of sizes 20 and more, we used the automatic search of CP Optimizer, which is more sophisticated than depth first search and is able to find better solutions much faster. The same search heuristics as above was used but, depending on the type of model (`Classic-TDTSP` or `TDNoOverlap-TDTSP`), the automatic search of CP Optimizer uses different meta-heuristics. This makes the comparison difficult to analyze precisely, since we cannot separate the effects of propagation (shown in figure 8.6) and search.

For those instances, `Classic-TDTSP` is not capable of proving optimality within the 1-hour time limit. The `TDNoOverlap-TDTSP` model finds and proves the optimal solution for all 120 instances of size 20 (with and without time-windows) and it proves optimality for 48 instances of size 30 with time-windows but cannot prove optimality for instances of this size without time-windows. On the other hand, it finds the reference solutions (or solutions with the same value) way before the 1 hour time-limit for sizes 20 and 30 as we can see in the average convergence graphs of figures 8.7 and 8.8. These graphs show the average of convergence curves over the 20 instances of an instance class.

In the average convergence graphs we see that `Classic-TDTSP` does not find a first solution immediately, the log scale makes it seem longer than it actually takes, but this time corresponds to an initialization of the automatic search by CP Optimizer. From analyzing the results of `Classic-TDTSP` and `TDNoOverlap-TDTSP` for size 10 that already indicate the superiority of `TDNoOverlap-TDTSP` to then looking at the scale-up behavior of both models for sizes 20 and 30 it seems straight-forward to conclude that `TDNoOverlap-TDTSP` does a better and faster job when it comes to solving the TDTSP(TW).



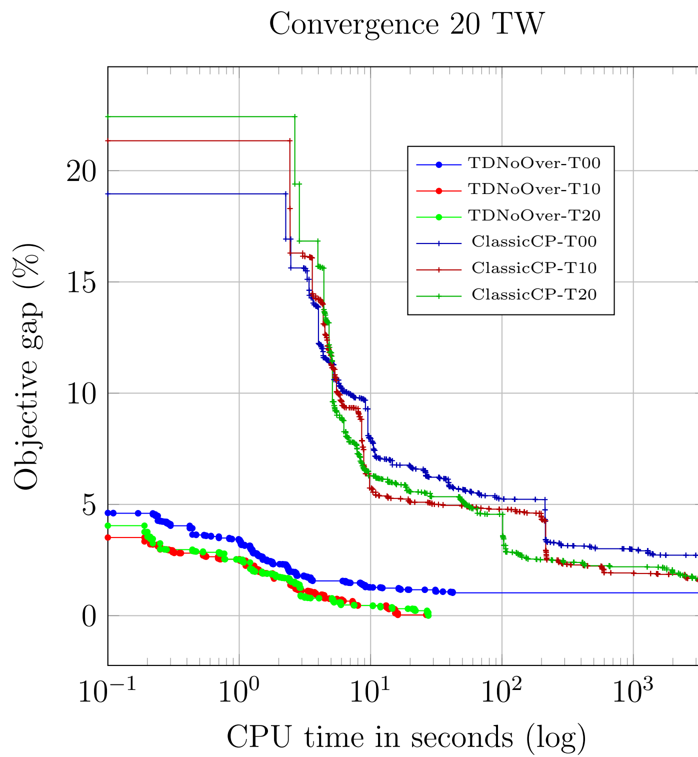
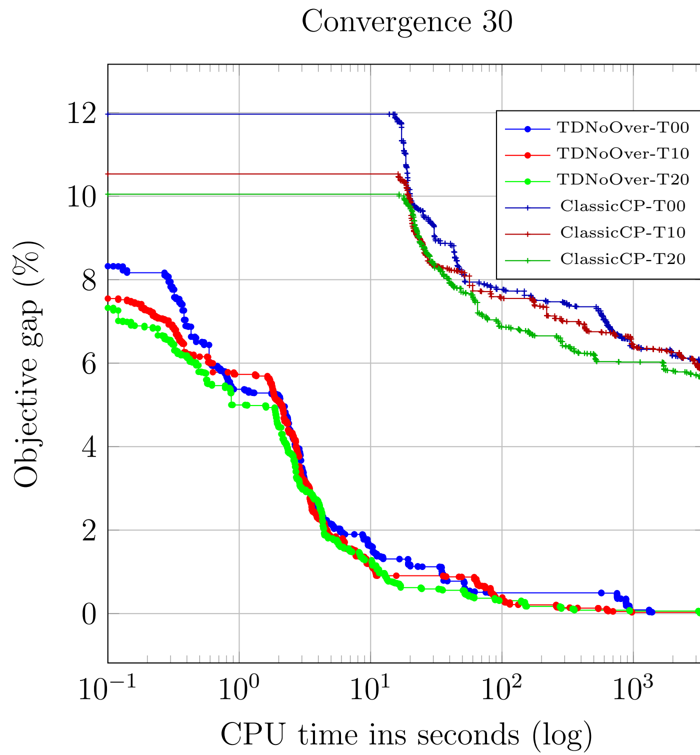


Figure 8.7: Average convergence of `Classic-TDTSP` and `TDNoOverlap-TDTSP` for size 20 with and without TW



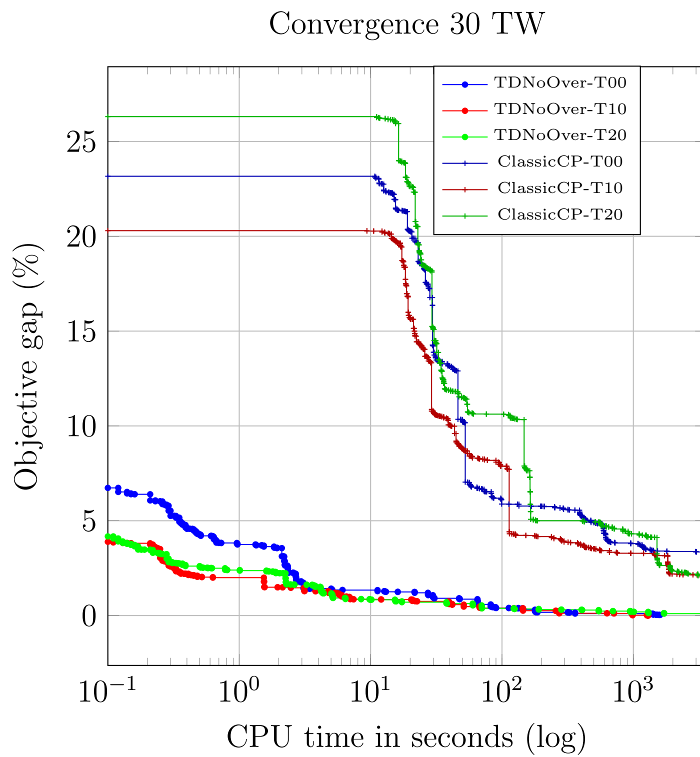


Figure 8.8: Average convergence of `Classic-TDTSP` and `TDNoOverlap-TDTSP` for size 30 with and without TW

In an effort to visualize all solutions instead of averages, as in the convergence curves shown above, we took snapshots of the values of the best solutions found by both models for a given search time t. In Fig. 8.9 we see one such snapshot took after 300 seconds of search for instances of size 30 with and without time-windows. In both cases `TDNoOverlap-TDTSP` is superior to `Classic-TDTSP`, as all values are plotted on the lower diagonal of the graph showing that values obtained by `TDNoOverlap-TDTSP` are smaller than those obtained by `Classic-TDTSP`. There seems to be no difference between the three matrices in terms of easiness of resolution, for one method or the other, since the different colored groups representing the matrices are all mingled together. We judged unnecessary to present other snapshots here since they do not bring any more information than this one.

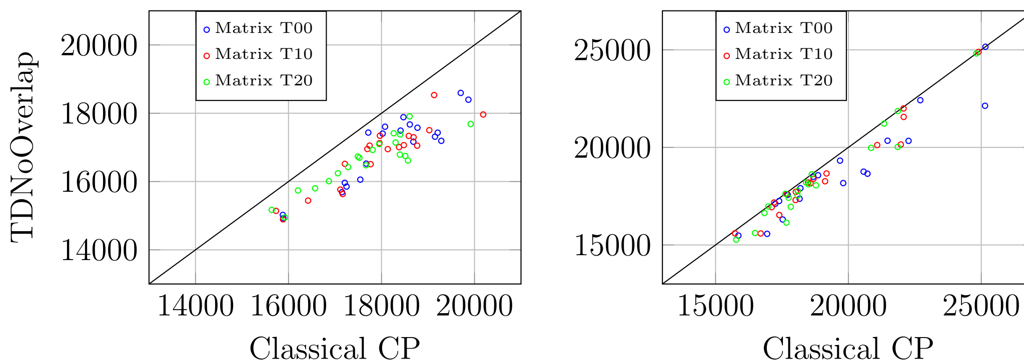


Figure 8.9: A colored circle on (x, y) represents one instance, where x is the cost of the best solution found by `Classic-TDTSP` and y is the cost found by `TDNoOverlap-TDTSP`, for problems of size 30 without time-windows (on the left) and with (on the right) after 300s

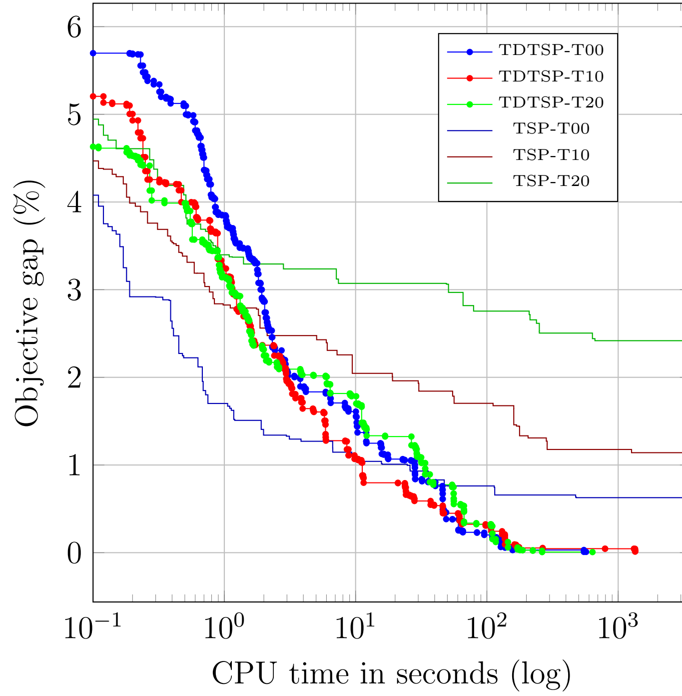
8.4.3 Comparison of convergence of `NoOverlap-TSP` and `TDNoOverlap-TDTSP`

When studying the effects of time-dependency on TSP solutions (section 8.3) we noticed that the model using median travel times `NoOverlap-TSP` had a "quick" convergence towards good TDTSP solutions. It could be interesting to make use of this fact to accelerate the search, if `NoOverlap-TSP` was able to find better solutions faster than `TDNoOverlap-TDTSP`. In this subsection we compare the convergence of those methods for instances without time-windows only since, as presented in section 8.3, the model `NoOverlap-TSP` most frequently produces infeasible solutions for the TDTSPTW.

In figures 8.10 and 8.11, the average convergence curves of both models are given for sizes 20, 30, 50 and 100. For the convergence curves of `NoOverlap-TSP` (represented in the graphs below by the dark colored curves, labeled with "TSP-"), the values plotted at each time correspond to the value of the best solution in terms of time-dependent travel times found so far. This means that TSP solutions found during the search are plotted only if they improve the current time-dependent objective obj_T, i.e., if $sol_{TSP}*$ was the last solution plotted, a new solution sol_{TSP} only appears in the convergence curve if $obj_T(sol_{TSP}) < obj_T(sol_{TSP}*)$.

As a general rule, `NoOverlap-TSP` produces solutions with smaller gaps for matrix $T00$ and then $T10$ and finally $T20$. This is normal since the larger the variations of travel times during the day, the larger the errors of estimation made with constant travel times. For sizes 20 and 30, the `TDNoOverlap-TDTSP` model tends to find better solutions than `NoOverlap-TSP` after at most 10 seconds (except for size 20 with $T00$, where `TDNoOverlap-TDTSP` takes a little longer to surpass `NoOverlap-TSP`). The larger the instances grow, the later this crossing point of convergences curves arrives, at around 30 seconds for size 50 and 400 seconds for size 100. This indicates that providing `NoOverlap-TSP` solutions (obtained after a few seconds) as a starting point for the `TDNoOverlap-TDTSP` model could accelerate the search but we did not have time to test that. One can also notice certain plateaus - for example, for the TDTSP curves of size 30 at around 1 and 2 seconds and for curves of size 100 between 5 and 10 seconds for the TSP and between 30 and 100 seconds for the TDTSP - that can be explained by the failure directed search of CP Optimizer.

When analyzing those graphs it is important to keep in mind that time is in log scale so at a first glance it might seem like it takes a very long time for `TDNoOverlap-TDTSP` to catch up, which is not the case (up to size 50). The main reason why `NoOverlap-TSP` improves solutions faster than `TDNoOverlap-TDTSP` is that the model is a lot lighter without the time-dependent travel time constraints. Another important factor to take into account in the comparison is that `NoOverlap-TSP` is not able of providing good solutions for the TDTSP with time windows, as a matter of fact, the solutions produced by this model are infeasible most of the time. So the only thing that can come out of it is an idea about how to speed up `TDNoOverlap-TDTSP`'s search.



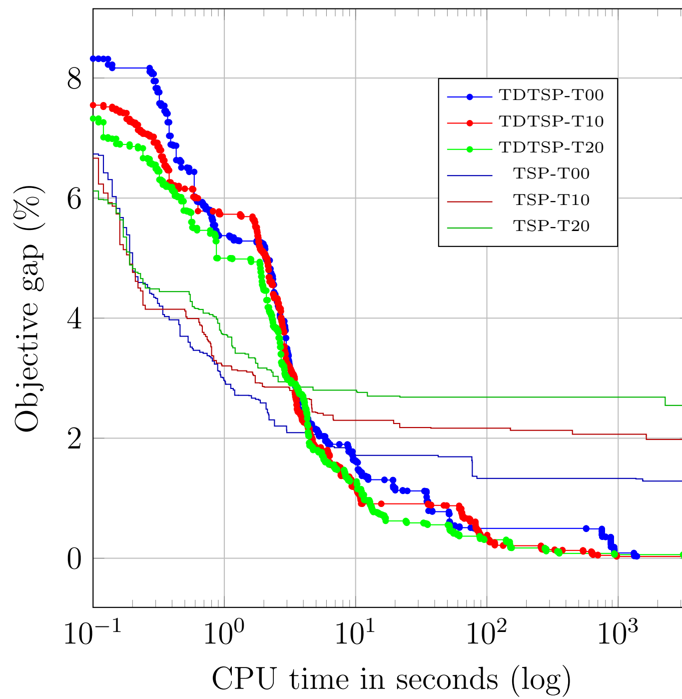
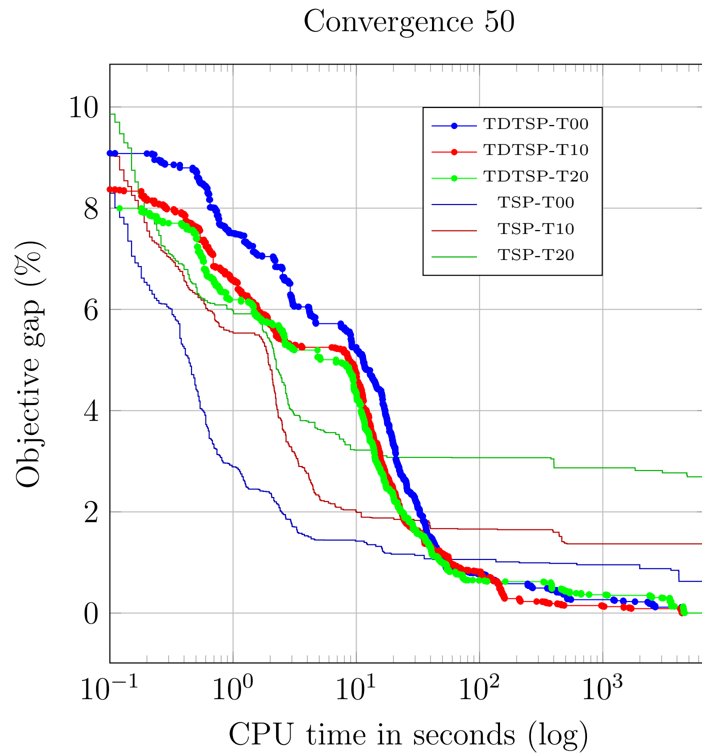


Figure 8.10: Average convergence curves of instances of size 20 (top) and 30 (bottom)



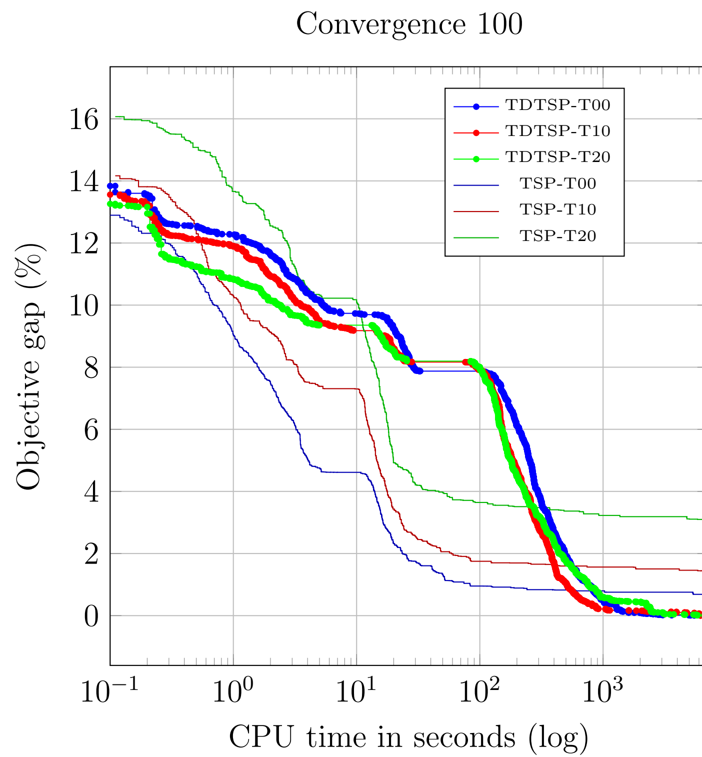


Figure 8.11: Average convergence curves of instances of size 50 (top) and 100 (bottom)

8.5 Tests on other instances

As mentioned in section 3.3.3, the only instances for the TDTSP available online (other than ours) are the ones from Cordeau et al. [26]. Time-window constraints are not provided online but in a subsequent paper [9], the authors randomly generate and test their approach on instances with time-windows. Here we briefly describe their instances (without time-windows) and the work done to adapt them to our format.

Travel speeds functions model Their instances are based on stepwise travel speeds functions $v_{i,j}$ for every couple of visits (i, j), the horizon $[0, T]$ is discretized into H time steps $[T_h, T_{h+1}]$. Given a couple of visits (i, j) and a time step h, the travel speed function can be written in terms of three factors: $v_{i,j,h} = \delta_{i,j,h} * b_h * u_{i,j}$, where $u_{i,j}$ is the maximum travel speed across arc (i, j) during the whole horizon $[0, T]$, $b_h = max_{i,j}\{v_{i,j,h}/u_{i,j}\}$ is the lightest congestion factor for the whole graph (during time step h) and $\delta_{i,j,h}$ represents the degradation of the congestion factor of arc (i, j) with respect to the least congested arc during time step h. Both b_h and $\delta_{i,j,h}$ belong to $[0, 1]$. This decomposition of speeds into those three factors is important for their method as they calculate different travel time bounds with them.

Instances structure Cordeau et al. generated 30 instances for each possible value of visits n in the set {15, 20, 30, 40}, that can be combined with each of two distinct traffic density patterns (A or B), giving a total of 60 instances per size.

- The number of customer locations n, which are randomly chosen within a $[0, 100]^2$ square. The depot is located at the center of this square and traffic density varies according to three concentric circular zones C1, C2 and C3 (Fig. 8.12).
- A time steps matrix $H \times 2$ gives the beginning and end of each time period, where the H is the number of times periods (always 3 for these instances), and the first and the last periods correspond to morning and evening rush hours respectively. In [26], they seem to indicate that this matrix is slightly modified according to the departure node, meaning that the boundaries of time periods vary for different nodes of the graph though they do not mention it in the sequel paper [9]. Therefore we used the matrix exactly as given in the input data files (the same for every node of the graph).

- A cluster matrix C $(n \times n)$, giving the clusters $c \in Clusters = \{C1, C2, C3\}$, for each couple of visits i, j. It is not clear how the authors attributed clusters for each arc as the instances description seem to suggest that nodes belong to clusters (as in Fig. 8.12) but in the input data, a matrix defines a unique cluster for each arc (i,j).
- A distance matrix D $(n \times n)$, giving the distances for each couple of visits i, j.
- A speed matrix S, which has to be multiplied by traffic pattern values to obtain the final speeds, corresponding to $b_h * u_c$ for each time period h and each cluster c

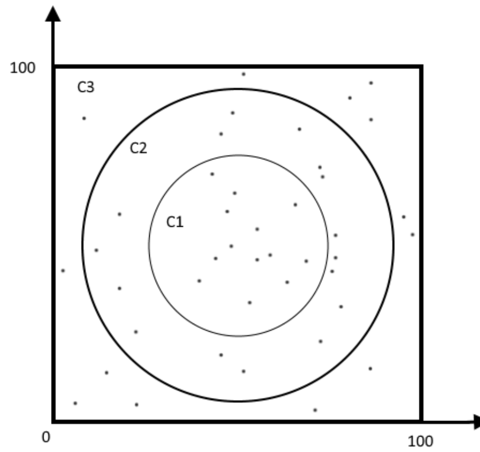


Figure 8.12: Example of customer locations, with concentric zones C1, C2 and C3 (from [9])

Traffic patterns ($\delta_{c,h}$ values) A and B are given in table 8.3, for each time step h and each cluster h. Traffic pattern A represents situations where the old city center is located in C1 with lower congestion, higher in C2 and decreasing towards C3 while traffic pattern B represent cities that have highest congestion in the center C1, that "continuously" decreases the further one gets from the center.

A	h=1	h=2	h=3
C1	1	1	1
C2	[0.70,0.98]	1	[0.70,0.98]
C3	[0.95,0.99]	1	[0.95,0.99]

B	h=1	h=2	h=3
C1	[0.70,0.98]	1	[0.70,0.98]
C2	[0.80,0.99]	1	[0.80,0.99]
C3	1	1	1

Table 8.3: Traffic patterns A and B

Adaptation to our input format In order to test with their instances, they have to be "transformed" to fit our input format, as described in the following steps:

- Obtain stepwise speed functions f^s for each cluster c: for every time step h, $f^s_{c,h} = S_{c,h} * \delta_{c,h}$.
- Calculate piecewise linear travel time function f^t from f^s and distance matrix D, using algorithm 2. It is important to use IGP's algorithm, as this is the one used by the authors on their method.

Our model `TDNoOverlap-TDTSP` can take general piecewise linear functions as input but it can only handle integer values for travel times (as interval variables have to have integer bounds) so, if a certain travel time is not integer it has to be rounded up for propagation. This process might introduce rounding errors that add up and produce a final objective value that is slightly different than the values obtained by Cordeau et al. In an effort to reduce this errors we added a conversion factor to increase the "precision" of travel times. The conversion factor is introduced by multiplying all travel times by it and then dividing the final solution by it again to retrieve the normal scale.

Using a conversion factor of 10, the worse gap between `TDNoOverlap-TDTSP` optimal solutions (proved up to size 20) and Cordeau's solutions is of 0.25% with an average gap of 0.06%. When `TDNoOverlap-TDTSP` no longer manages to prove optimality, from sizes 25 to 40, the maximum gap obtained when running `TDNoOverlap-TDTSP` for a time limit of 10 minutes was 2.6%, while the average gap for those instances was only 0.69%. It is possible that differences in optimal values are explained by the boundaries of time steps that might have been different in their case, as mentioned earlier. Their method does not prove optimality for all instances, and it ran with a time limit of one hour. We also observed that the computation times of their method seems to vary quite a lot with the instance, varying from 0.56 seconds to 38.29 for instances of size 20, while ours had a very similar running time for all instances of the same size, running in around 20 seconds for size 20.

Experimentation on their instances was not thorough as we judged that proper comparison was hard, so here we only gave an idea of why it was not simple to compare with their results (float versus integer comparison and ambiguity on their instances' definition) and did not present results for all instances and different conversion factors. Even if solutions produced were not exactly the same, it was interesting to validate our model on other instances as well. For a more complete and fair comparison between both methods it would be interesting to test their approach on our instances (with

a larger number of time steps, travel time functions for every arc instead of clusters and also with time-window constraints), even though the same difficulty of comparing float and integer solutions would remain.

Chapter 9

Conclusion

In this thesis we presented the practical and theoretical aspects of scheduling a vehicle delivery tour in a urban center. In the initial chapters (from 1 to 3) the context, the necessary background, as well as a formal definition and a literature review of the theoretical problem concerned, the TDTSP(TW), were given.

In a second part, questions concerning the modeling of road-networks, using real-world time-dependent data obtained in the context of the Optimod'Lyon project, were studied. In chapter 4, the procedure used to generate a set of benchmark TDTSP(TW) instances was described in details and more specific questions concerning the time-dependent travel time functions modeling were discussed in chapter 5. The contributions of chapter 4 and 5 were the benchmark and algorithms: Alg. 2, the extension of the IGP algorithm that calculates a travel time function instead of a single travel time; Alg. 5, that calculates a piecewise linear FIFO travel time function as close as possible to the input travel time function (stepwise or piecewise linear); and Alg. 6, which transforms a TDTSP instance graph to make it satisfy the time-dependent triangular inequality.

In a third part, our solving approach was introduced. In chapter 6, a CP model for the TDTSP was presented, based on a CP model for the TSP. The limitations of this model were pointed out - namely, lack of reasoning with indirect successors and no global reasoning with time-dependent transition times - and a scheduling model tackling the first of the drawbacks of the first model was given. This scheduling model served as basis for the development of a new global constraint, aiming to solve the second limitation. The internal functioning of the new constraint `TDNoOverlap-TDTSP` was presented in details in chapter 7. From an application perspective, one of the interests of a constraint-based scheduling model is that it is very easy to integrate additional constraints like precedence between visits or disjunctive time-windows,

these constraints are in fact directly available in CP Optimizer.

An experimental evaluation was performed in chapter 8, in three steps: (1) an evaluation of the effects of taking time-dependency into account confirmed previous results of the literature showing that gains of up to 20% in terms of total travel time can be obtained with time-dependent models and that time-window constraints are almost never satisfied when time-dependency is ignored (2) a comparison of the scaling of the different models proposed in this thesis showed that the new constraint has better performance than the models presented in chapter 6 but we also see that there is still room for improvement (3) the instances proposed by [9] and the work done in order to take those instances as input were presented though a detailed comparison was not possible.

We believe that CP can be a good platform for vehicle scheduling problems but a lot of work still has to be done in order to compare it with other approaches. Part of the work is to improve the performance of `TDNoOverlap-TDTSP` by optimizing the implementation as proposed in 7.3. The lack of common benchmarks for the TDTSP or TDVRP also makes comparisons difficult, in that sense, we expect to contribute to the community with the benchmark instances proposed in this thesis.

Future research One research direction to explore would be to improve the propagation of `TDNoOverlap-TDTSP` by calculating tighter bounds for the TDTSP, using Minimum Spanning Trees or Assignment Problem relaxations on the precedence graph, extending the approaches described in [40, 17, 36]. It would also be interesting to see if the successor relations stored in the precedence graph could be exploited in this context. Still on the solving part, it would be interesting to test with different branching heuristics and to take a closer look into the search in general, as we have focused on the propagation.

In what concerns the final application, a possible future research direction would be the adaptation to the dynamic case - taking into account changes occurring in the road-network while the tour is being executed. Detecting that a change of speeds in a given arc can potentially have an impact on the tour (even if the tour does not currently, or in the future, travel through that arc) is a not an easy job. A re-optimization of the sequence of deliveries left can be done once time-dependent shortest paths are updated accordingly. An alternative, in order to produce more robust solutions that would not suffer so much from changes in travel times, would be to consider the stochastic version of the time-dependent problem.

Another possibility would be to generalize to other constraints the work

of extension done for the *NoOverlap* constraint, in order to be able to efficiently reason on time-dependent data and solve other problems where data varies with time. For example, in the case of the shortest path problem with resource constraints, for which [69] proposed the *Multicost-Regular* global constraint. In an urban context, costs (and specially durations) depend on time and it would be interesting to extend this global constraint as we have done for *NoOverlap* in this thesis.

Appendix A

Backward IGP algorithm

Input: v, t_f

1: $t \leftarrow t_f$
2: $k \leftarrow k_f : T_{k_f} \leq t_f < T_{k_f+1}$
3: $d \leftarrow L_{ij}$
4: $t' \leftarrow t - d/v(i,j,k)$
5: **while** $t' < T_k$ **do**
6: $\quad d \leftarrow d - v(i,j,k) * (t - T_k)$
7: $\quad t \leftarrow T_k$
8: $\quad t' \leftarrow t - d/v(i,j,k+1)$
9: $\quad k \leftarrow k - 1$
10: **end while**
11: **return** t'

Bibliography

[1] Cplex optimization studio 12.6.3. "http://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.3/ilog.odms.studio.help/Optimization_Studio/topics/COS_home.html".

[2] Openstreetmap database. "https://www.openstreetmap.org/".

[3] Tomtom traffic index. "https://www.tomtom.com/en_gb/trafficindex/".

[4] Hernán Abeledo, Ricardo Fukasawa, Artur Pessoa, and Eduardo Uchoa. The time dependent traveling salesman problem: polyhedra and algorithm. *Mathematical Programming Computation*, 5(1):27–55, 2013.

[5] BH Ahn and JY Shin. Vehicle-Routeing with Time Windows and Time-Varying Congestion. *Journal of the Operational Research Society*, 42(5):393–400, 1991.

[6] J Albiach, J Sanchis, and D Soler. An asymmetric TSP with time windows and with time-dependent travel times and costs: An exact solution through a graph transformation. *European Journal Of Operational Research*, 189:789–802, 2008.

[7] L.S.Kang A.M.Zhou and Z.Y.Yan. Solving dynamic tsp with evolutionary approach in real time. In *Proceedings of the Congress on Evolutionary Computation, Canberra, Australia*. IEEE Press, 2003.

[8] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[9] a Arigliano, G Ghiani, and E Guerriero. Time Dependent Traveling Salesman Problem With Time Windows: Properties and an Exact Algorithm. *EJOR*, 2014.

[10] P. Baptiste, P. Laborie, C. Le Pape, and W. Nuijten. Constraint-based scheduling and planning. In T. Walsh F. Rossi, P. van Beek, editor, *Handbook of Constraint Programming*, chapter 22, pages 759–798. Elsevier, 2006.

[11] P. Baptiste, P. Laborie, C. Le Pape, and W. Nuijten. *Handbook of Constraint Programming*, chapter 22: Constraint-Based Scheduling and Planning, pages 759–798. Elsevier, 2006.

[12] Ph. Baptiste and C. Le Pape. Disjunctive constraints for manufacturing scheduling: Principles and extensions. In *Proc. 3rd International Conference on Computer Integrated Manufacturing*, 1995.

[13] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer Science & Business Media, 2012.

[14] Roman Bartak. Guide to constraint programming. `http://kti.ms.mff.cuni.cz/~bartak/constraints/propagation.html`. [Online; accessed 20-October-2012].

[15] Thomas Baudel, Laetitia Dablanc, Penelope Aguiar-Melgarejo, and Jean Ashton. Optimizing urban freight deliveries: From designing and testing a prototype system to addressing real life challenges. *Transportation Research Procedia, Tenth International Conference on City Logistics*, 12(2016):170–180, 2015.

[16] JE Beasley. Adapting the savings algorithm for varying inter-customer travel times. *Omega*, 9(6):658–659, 1981.

[17] Pascal Benchimol, Willem Jan van Hoeve, Jean-Charles Regin, Louis-Martin Rousseau, and Michel Rueher. Improved filtering for weighted circuit constraints. *Constraints*, pages 205–233, 2012.

[18] Johannes Bentner, Günter Bauer, Gustav M Obermair, Ingo Morgenstern, and Johannes Schneider. Optimization of the time-dependent traveling salesman problem with Monte Carlo methods. *Physical Review E*, 64(3), August 2001.

[19] W. C. Benton and Arthur V. Hill. Modelling intra-city time-dependent travel speeds for vehicle scheduling problems. *Journal of the Operational Research Society*, 1992.

[20] Ashish Bhaskar. A methodology (cuprite) for urban network travel time estimation by integrating multisource data. 2009.

[21] JJM Bront. *Integer Programming approaches to the Time Dependent Travelling Salesman Problem*. PhD thesis, Universidad de Buenos Aires, 2012.

[22] Hadrien Cambazard and Bernard Penz. A constraint programming approach for the traveling purchaser problem. In *Principles and Practice of Constraint Programming*, pages 735–749. Springer, 2012.

[23] Yves Caseau and Francois Laburthe. Solving small tsps with constraints. In *ICLP*, volume 97, page 104, 1997.

[24] C.J.Eyckelhof and M.Snoek. Ant systems for a dynamic tsp - ants caught in a traffic jam. *3rd International Workshop on Ant Algorithms*, 2002.

[25] William Cook. Concorde tsp solver. `http://www.tsp.gatech.edu/concorde.html`. [Online; accessed 27-August-2016].

[26] Jean-François Cordeau, Gianpaolo Ghiani, and Emanuela Guerriero. Properties and Branch-and-Cut Algorithm for the Time-Dependent Traveling Salesman Problem, 2012.

[27] T. H. Cormen, C. E. Leiserson, and Rivest R. L. *Introduction to Algorithms*. MIT Press, McGraw-Hill edition, 1990.

[28] Said Dabia and Stefan Ropke. Branch and Price for the Time-Dependent Vehicle Routing Problem with Time Windows. 47(3):380–396, 2013.

[29] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks. Lecture Notes in Computer Science*. Springer, 2009.

[30] Emrah Demir, Tolga Bektaş, and Gilbert Laporte. A comparative analysis of several vehicle emission models for road freight transportation. *Transportation Research Part D: Transport and Environment*, 16(5):347–357, 2011.

[31] Académie des technologies. Le transport de marchandises, 2009.

[32] 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. *European Journal of Operational Research*, 185(3):1174–1191, March 2008.

[33] Stuart E Dreyfus. An appraisal of some shortest-path algorithms. *Operations research*, 17(3):395–412, 1969.

[34] Jan Fabian Ehmke, Ann Melissa Campbell, and Barrett W. Thomas. Vehicle Routing to Minimize Time-Dependent Emissions in Urban Areas. *European Journal of Operational Research*, 2015.

[35] Jan Fabian Ehmke, André Steinert, and Dirk Christian Mattfeld. Advanced routing for city logistics service providers based on time-dependent travel times. *Journal of Computational Science*, 3(4):193–205, 2012.

[36] J. G. Fages and X Lorca. Improving the asymmetric tsp by considering graph structure. arxiv preprint arxiv:1206.3437, 2012.

[37] Miguel Andres Figliozzi. The time dependent vehicle routing problem with time windows: Benchmark problems, an efficient solution algorithm, and solution characteristics. *Transportation Research Part E Logistics and Transportation Review*, 48:616–636, 2012.

[38] Bernhard Fleischmann, Martin Gietz, and Stefan Gnutzmann. Time-Varying Travel Times in Vehicle Routing. *Transportation Science*, 38(2):160–173, May 2004.

[39] F. Focacci, P. Laborie, and W. Nuijten. Solving scheduling problems with setup times and alternative resources. In *Proc. Fifth International Conference on Artificial Intelligence Planning and Scheduling, AIPS'00*, pages 92–101. AAAI Press, 2000.

[40] F Focacci, P Laborie, and W Nuijten. Solving Scheduling Problems with Setup Times and Alternative Resources. *AIPS Proceedings*, 2000.

[41] Kenneth R Fox, Bezalel Gavish, and Stephen C Graves. An n-Constraint Formulation of the (Time- Dependent) Traveling Salesman Problem. *Operations Research*, 28(4):1018–1021, 1980.

[42] Fabio Furini, Carlo Alfredo Persiani, and Paolo Toth. The Time Dependent Traveling Salesman Planning Problem in Controlled Airspace. *Transportation Research Part E*, 2015.

[43] Michel Gendreau, Gianpaolo Ghiani, and Emanuela Guerriero. Time-dependent routing problems: A review. *Computers & Operations Research*, 64:189–197, 2015.

[44] Gianpaolo Ghiani and Emanuela Guerriero. A Note on the Ichoua et al (2003) Travel Time Model. *Transportation Science*, 2013.

[45] Daniel Godard, Philippe Laborie, and Wim Nuijten. Randomized large neighborhood search for cumulative scheduling. In *ICAPS*, volume 5, pages 81–89, 2005.

[46] Luis Gouveia and Stefan Voss. A classification of formulations for the (time-dependent) traveling salesman problem. *European Journal of Operational Research*, 83(1):69 – 82, 1995.

[47] Hideki Hashimoto, Mutsunori Yagiura, and Toshihide Ibaraki. An iterated local search algorithm for the time-dependent vehicle routing problem with time windows. *Discrete Optimization*, 5(2):434–456, May 2008.

[48] H.N.Psaraftis. Dynamic vehicle routing problems. In B.L.Golden and A.A.Assad, editors, *Vehicles Routing: Methods and Studies*. Elsevier Science Publishers, 1988.

[49] Soumia Ichoua, Michel Gendreau, and Jean-Yves Potvin. Vehicle Dispatching With Time-Dependent Travel Times. *European Journal of Operations Research*, 2003.

[50] D. E. Kaufman and R. L. Smith. Fastest paths in time-dependent networks for intelligent vehicle-highway systems application. *Journal of Intelligent Transportation Systems*, 1993.

[51] Elena Kelareva, Kevin Tierney, and Philip Kilby. CP Methods for Scheduling and Routing with Time-Dependent Task Costs. *EURO Journal on Computational Optimization*, 2:147–194, 2014.

[52] Astrid S Kenyon and David P Morton. Stochastic vehicle routing with random travel times. *Transportation Science*, 37(1):69–82, 2003.

[53] a. L. Kok, E. W. Hans, and J. M J Schutten. Vehicle routing under time-dependent travel times: The impact of congestion avoidance. *Computers and Operations Research*, 39:910–918, 2012.

[54] Ratnesh Kumar and Zhonghui Luo. Optimizing the operation sequence of a chip placement machine using tsp model. *IEEE Transactions on Electronics Packaging Manufacturing*, 26(1):14–21, 2003.

[55] P. Laborie. Algorithms for Propagating Resource Constraints in AI Planning and Scheduling: Existing Approaches and New Results. *Artificial Intelligence*, 143(2):151–188, 2003.

[56] P. Laborie and D. Godard. Self-adapting large neighborhood search: Application to single-mode scheduling problems. In *Proceedings of the 3rd Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA)*, pages 276–284, 2007.

[57] Philippe Laborie and Jerome Rogerie. Reasoning with Conditional Time-Intervals. In *FLAIRS Conference*, pages 555–560, 2008.

[58] Philippe Laborie, Jerome Rogerie, Paul Shaw, and Petr Vilím. Reasoning with Conditional Time-Intervals. Part II: an Algebraical Model for Resources. In *FLAIRS Conference*, 2009.

[59] Gilbert Laporte, Francois Louveaux, and Hélène Mercure. The vehicle routing problem with stochastic travel times. *Transportation science*, 26(3):161–170, 1992.

[60] Allan Larsen and Oli BG Madsen. *The dynamic vehicle routing problem*. PhD thesis, Technical University of DenmarkDanmarks Tekniske Universitet, Department of TransportInstitut for Transport, Logistics & ITSLogistik & ITS, 2000.

[61] C. Lecoutre. *Constraint Networks: Techniques and Algorithms*. ISTE/Wiley, 2009.

[62] Ji Youn Lee, Soo-Yong Shin, Tai Hyun Park, and Byoung-Tak Zhang. Solving traveling salesman problems with dna molecules encoding numerical values. *BioSystems*, 78(1):39–47, 2004.

[63] Feiyue Li, Bruce Golden, and Edward Wasil. Solving the time dependent traveling salesman problem. In Bruce Golden, S. Raghavan, and Edward Wasil, editors, *The Next Wave in Computing, Optimization, and Decision Technologies*, volume 29 of *Operations Research/Computer Science Interfaces Series*, pages 163–182. Springer US, 2005.

[64] W Maden, RW Eglese, and DP Black. Vehicle routing and scheduling with time varying data: A case study. *Journal of the Operational Research Society*, 61:515–522, 2009.

[65] Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. Optimal and efficient filtering algorithms for table constraints. *Constraints*, 19(1):77–120, 2014.

[66] C. Malandraki and M. S. Daskin. Time Dependent Vehicle Routing Problems: Formulations, Properties and Heuristic Algorithms. *Transportation Science*, 26:185–200, 1992.

[67] Chryssi Malandraki and Robert B Dial. A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem. *European Journal Of Operational Research*, 90:45–55, 1996.

[68] Aguiar Penélope Melgarejo, Philippe Laborie, and Christine Solnon. A Time-Dependent No-Overlap Constraint : Application to Urban Delivery Problems. *CPAIOR*, pages 1–17, 2015.

[69] Julien Menana and Sophie Demassey. Sequencing and counting with the multicost-regular constraint. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 178–192. Springer, 2009.

[70] Junichi Ochiai and Hitoshi Kanoh. Solving Real-World Delivery Problem Using Improved Max-Min Ant System With Local Optimal Solutions In Wide Area Road. *International Journal of Artificial Intelligence & Applications (IJAIA)*, 5(3):21–36, 2014.

[71] A. Orda and R. Rom. Shortest-path and minimum delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, pages 607 – 625, 1990.

[72] J.-C. Picard and M. Queyranne. The Time-Dependent Traveling Salesman Problem and Its Application to the Tardiness Problem in One-Machine Scheduling, 1978.

[73] Nicola Policella, Stephen F Smith, Amedeo Cesta, and Angelo Oddi. Generating robust schedules through temporal flexibility. In *ICAPS*, volume 4, pages 209–218, 2004.

[74] Philippe Refalo. Impact-based search strategies for constraint programming. In *CP*, pages 557–571, 2004.

[75] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, volume 94, pages 362–367, 1994.

[76] G. Reinelt. Tsplib95. "`http://www.iwr.uniheidelberg.de/iwr/comopt/software/TSPLIB95`". [Online; accessed 27-August-2016].

[77] Johannes Schneider. The time-dependent traveling salesman problem. *Physica A: Statistical Mechanics and its Applications*, 314(1-4):151–155, November 2002.

[78] D Soler, J Albiach, and E Martinez. A way to optimally solve a time-dependent Vehicle Routing Problem with Time Windows. *Operations Research Letters*, 37:37–42, 2009.

[79] Marius M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, 1987.

[80] Gabriella Stecco, Jean-François Cordeau, and Elena Moretti. A branch-and-cut algorithm for a production scheduling problem with sequence-dependent and time-dependent setup times. *Computers & Operations Research*, 35(8):2635– 2655, August 2008.

[81] Gabriella Stecco, Jean-François Cordeau, and Elena Moretti. A tabu search heuristic for a sequence-dependent and time-dependent scheduling problem on a single machine. *Journal of Scheduling*, 12:3–16, 2009.

[82] Willem-Jan van Hoeve and Irit Katriel. Global constraints, 2006.

[83] T. Van Woensel, L. Kerbache, H. Peremans, and N. Vandaele. Vehicle routing with dynamic travel times: A queueing approach. *European Journal of Operational Research*, 186(3):990–1007, may 2008.

[84] Petr Vilím, Philippe Laborie, and Paul Shaw. Failure-directed search for constraint-based scheduling. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 437–453. Springer International Publishing, 2015.

[85] D. Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, pages 19–91. P. H. Winston, McGraw–Hill, New York, 1975.

[86] Tom Van Woensel, Laoucine Kerbache, Herbert Peremans, and Nico Vandaele. A Queueing Framework for Routing Problems with Time-dependent Travel Times. *Journal of Mathematical Modelling and Algorithms*, 6(1):151–173, November 2006.

[87] Laura Wynter, Barry M Trager, Yichong Yu, Yiannis Kararianakis, Saif Jabari, Jean Coldefy, Dimitri Marquois, and Thomas Baudel. Traffic estimation and prediction for urban road networks, application to grandlyon. In *ITS World Congress, Bordeaux*. ERTICO, 2015.

[88] Kamran Zaidi, Milos Milojevic, Veselin Rakocevic, and Muttukrishnan Rajarajan. Data-centric rogue node detection in vanets. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 398–405. IEEE, 2014.

[89] X.L.Hu Z.C.Huang and S.D.Chen. Dynamic traveling salesman problem based on evolutionary computation. In *Congress on Evolutionary Computation(CEC'01)*. IEEE Press, 2001.

FOLIO ADMINISTRATIF

THESE DE L'UNIVERSITE DE LYON OPEREE AU SEIN DE L'INSA LYON

NOM : AGUIAR MELGAREJO DATE de SOUTENANCE : 16/12/2016

Prénom : Penélope

TITRE : A Constraint Programming Approach for the Time Dependent Traveling Salesman Problem

NATURE : Doctorat Numéro d'ordre : 2016LYSEI142

Ecole doctorale : Informatique et Mathématiques de Lyon

Spécialité : Informatique

RESUME : L'optimisation des tournées de livraison est souvent modélisée par un problème de voyageur de commerce (Traveling Salesman Problem / TSP). Pour ce problème, il est fréquent d'avoir des contraintes additionnelles telles que, par exemple, des fenêtres horaires limitant les heures de livraison chez le client ou des pauses obligatoires pour les conducteurs des camions. Le temps est une dimension importante à prendre en compte pour respecter ces contraintes. Cependant, les durées des trajets ne sont généralement pas constantes mais varient en fonction des congestions, et cette variabilité doit être intégrée au moment de l'optimisation des tournées. Ainsi, le problème du voyageur de commerce dépendant du temps (Time Dependent TSP / TD-TSP) est la version étendue du TSP où le coût d'un arc dépend de l'heure à laquelle cet arc est emprunté.

Dans cette thèse nous proposons un nouveau benchmark pour le TDTSP basé sur des données réelles de trafic (fournies par la Métropole de Lyon) et nous montrons l'intérêt de prendre en compte la variabilité des durées dans ce problème. Nous étudions comment mieux modéliser les fonctions de durée de trajet dépendantes du temps. Nous introduisons et comparons différents modèles pour résoudre le TDTSP avec la programmation par contraintes (Constraint Programming / CP). Un premier modèle est directement dérivé du modèle CP classique pour le TSP. Nous montrons que ce modèle ne permet pas de raisonner avec des relations de précédence indirectes, ce qui pénalise sa performance sur notre benchmark. Nous introduisons une nouvelle contrainte globale qui est capable d'exploiter des relations de précédence indirectes sur des données dépendantes du temps et nous introduisons un nouveau modèle CP basé sur notre nouvelle contrainte. Nous comparons expérimentalement les deux modèles sur notre benchmark, et nous montrons que notre nouvelle contrainte permet de résoudre le TDTSP plus efficacement.

MOTS-CLÉS : Programmation par Contraintes ; Time Dependent Traveling Salesman Problem; Vehicle Scheduling ; Constraint Programming ; Problème du Voyageur de Commerce

Laboratoire (s) de recherche : LIRIS

Directrice de thèse : Christine SOLNON

Président de jury :
EL-FAOUZI, Nour-Eddin Directeur de Recherche IFSTTAR

Composition du jury :
EL-FAOUZI, Nour-Eddin Directeur de Recherche IFSTTAR Président
DEVILLE, Yves Professeur des Universités Université Catholique de Louvain Rapporteur
HUGUET, Marie-José Professeur des Universités INSA de Toulouse Rapporteur
CAMBAZARD, Hadrien Maître de Conférences INP Grenoble Examinateur

SOLNON, Christine Professeur INSA Lyon Directrice de thèse
LABORIE, Philippe Docteur IBM Co-directeur de thèse