The traveling salesman problem in constraint programming

Nicolas Isoart

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Le problème du voyageur de commerce en programmation par contraintes

Nicolas ISOART

Laboratoire d'Informatique, de Signaux et Systèmes de Sophia Antipolis (I3S)
UMR7271 UCA CNRS

Présentée en vue de l'obtention du grade de docteur en INFORMATIQUE **d'**Université Côte d'Azur

Dirigée par : Jean-Charles RÉGIN, Professeur, Université Côte d'Azur

Soutenue le : 19 novembre 2021

Devant le jury, composé de :
David COUDERT, Directeur de recherche, INRIA Sophia Antipolis
Claude-Guy QUIMPER, Professeur, Université Laval
Willem-Jan VAN HOEVE, Professeur, Carnegie Mellon University
Xavier LORCA, Professeur, IMT Mines Albi-Carmaux
Hadrien CAMBAZARD, Maître de conférences, Grenoble INP

LE PROBLÈME DU VOYAGEUR DE COMMERCE EN PROGRAMMATION PAR CONTRAINTES

The Traveling Salesman Problem in Constraint Programming

Nicolas ISOART

Jury :

Président du jury
David COUDERT, Directeur de recherche, INRIA Sophia Antipolis

Rapporteurs
Claude-Guy QUIMPER, Professeur, Université Laval
Willem-Jan VAN HOEVE, Professeur, Carnegie Mellon University
Xavier LORCA, Professeur, IMT Mines Albi-Carmaux

Examinateurs
Hadrien CAMBAZARD, Maître de conférences, Grenoble INP

Directeur de thèse
Jean-Charles RÉGIN, Professeur, Université Côte d'Azur

Université Côte d'Azur

Nicolas ISOART

Le problème du voyageur de commerce en programmation par contraintes

xiii+173 p.

Ce document a été préparé avec LaTeX2e et la classe `these-ISSS` version v. 2.10.

Impression : sommaire.tex – 1/12/2021 – 19:34

Révision pour la classe : these-ISSS.cls,v 2.10 2020/06/24 14:16:37 mpelleau

If it disagrees with experiment, it's wrong. In that simple statement is the key to science. It doesn't make any difference how beautiful your guess is, it doesn't matter how smart you are who made the guess, or what his name is . . . If it disagrees with experiment, it's wrong. That's all there is to it.

— Richard Feynman, Lecture.

Le problème du voyageur de commerce en programmation par contraintes

Résumé

Plusieurs modèles de programmation par contraintes, basés sur la méthode de relaxation lagrangienne (LR), ont été introduits pour résoudre le problème du voyageur de commerce (TSP). Dans cette thèse, nous définissons trois nouvelles contraintes et algorithmes de filtrage considérant la structure du graphe. La contrainte k-cutset impose que toute solution contienne un nombre strictement positif et pair d'éléments dans chaque cutset. La contrainte mandatory Hamiltonian path est basée sur l'algorithme de recherche locale k-opt. Si un chemin composé d'arêtes obligatoires n'est pas lui-même optimal (c.-à-d., il existe un k-opt), alors ce chemin n'appartient à aucune solution optimale. Enfin, la contrainte du 1-tree est basée sur l'idée que si le problème peut être décomposé en deux sous-problèmes indépendants, alors une partie du 1-tree peut être optimale dans un des sous-problèmes. De plus, nous introduisons l'algorithme SSSA afin d'améliorer les temps de résolution. SSSA évite les oscillations et les non-variations de la fonction objective de la LR. Ensuite, nous parallélisons la recherche de solutions avec Embarrassingly Parallel Search (EPS). Malheureusement, le mécanisme de décomposition d'EPS est un processus à profondeur borné, contrairement à la stratégie de recherche utilisée pour résoudre la TSP qui est en profondeur d'abord. Cela rend difficile l'obtention de bons résultats en appliquant directement EPS. Afin de diminuer ce défaut, nous introduisons un algorithme de précalcul. Cependant, des sous-problèmes avec des temps de résolution extrêmement différents peuvent apparaître. Pour remédier à cela, nous introduisons une méthode procédant à des redécompositions dans EPS. Finalement, nous expérimentons sur la TSPLib. Nous montrons que les contraintes structurelles permettent de réduire les temps de résolution d'un ordre de grandeur, et que la parallélisation permet d'obtenir de très bons résultats liés au nombre de coeurs.

Mots-clés : PPC, TSP, algorithmes, optimisation, parallélisme.

The Traveling Salesman Problem in Constraint Programming

Abstract

Several constraint programming (CP) models, based on Lagrangian relaxation (LR), have been introduced to solve the traveling salesman problem (TSP). In this thesis, we define three new constraints and filtering algorithms based on the structure of the graph. First, the k-cutset constraint imposes that any solution contains a strictly positive and even number of elements in each cutset. Then, the mandatory Hamiltonian path constraint is based on the local search k-opt algorithm. If a path of mandatory edges is not optimal (i.e. it exists a k-opt), then it cannot belong to any optimal solution. Finally, the 1-tree constraint is based on the idea that if the problem can be decomposed in two independent sub-problems, then a part of the 1-tree can be optimal in a sub-problem. In addition, to speed-up the practical performances, we introduce an algorithm named SSSA to avoid oscillations and non-variations of the objective function of LR, saving useless solving times. We also parallelize the search for solutions with Embarrassingly Parallel Search (EPS). Unfortunately, a direct application of EPS does not lead to good results for the TSP. Indeed, the decomposition mechanism of EPS is a depth-bounded process whereas the search strategy used to solve the TSP is depth-first. Therefore, we define a diving algorithm fixing this issue. However, sub-problems with extremely different solving times may appear. Thus, we introduce a re-decomposition policy in EPS. Finally, our experiments on the TSPLib showed that the structural constraints reduce the solving times by an order of magnitude. Moreover, we show that our version of EPS leads to a huge improvement related to the number of cores.

Keywords: CP, TSP, algorithms, optimization, parallelism.

Remerciements

Tout d'abord, je souhaiterais remercier très fortement mon directeur de thèse, Jean-Charles Régin, d'avoir cru en moi depuis le master jusqu'à la fin de ma thèse. Je pense très sincèrement que peu de doctorants ont la chance d'avoir un directeur de thèse aussi bienveillant, intelligent et présent que Jean-Charles.

Je remercie tout particulièrement Claude-Guy Quimper, Willem-Jan Van Hoeve et Xavier Lorca d'avoir accepté de rapporter ma thèse. Par ailleurs, je remercie David Coudert et Hadrien Cambazard d'avoir participé à mon jury de thèse.

Ensuite, je tiens à remercier les doctorants du laboratoire ayant rendu mes années de thèse plus qu'agréable. En commençant par les plus vieux, Ophélie Guinaudeau et Ingrid Grenet, merci pour toutes les discussions que nous avons pu avoir avant et pendant la thèse. Merci à Jonathan Behaegel et Heytem Zitoun. Merci à Adrien Gausseran et Laetitia Laversa pour tous ces bons (et moins bons) moments passés en master, puis en thèse. Merci tout particulièrement à Rémy Garcia pour toute l'aide qu'il a pu m'apporter au quotidien durant ma thèse. Enfin, merci à tous les doctorants et stagiaires du laboratoire Samvel Dersarkissian, Arthur Finkelstein, Sara Riva, Laetitia Gibart, Diana Resmerita, Loïc Germerie, François Doré, Amélie Gruel, Giulia Rocco, Bastien Rousseau, Aymeric Picard, Steve Malalel, Alexandre Bonlarron, Florian Régin et à ceux que je pourrais oublier.

Merci à tous les permanents de l'équipe pour l'accueil durant ces 3 années et pour les nombreuses discussions autour de la machine à café. Je tiens particulièrement à remercier Arnaud Malapert, Marie Pelleau, Cinzia Di Giusto pour l'aide et les conseils qu'ils m'ont apportés au cours de ma thèse. Aussi, je remercie Enrico Formenti, Bruno Martin et Sandrine Julia.

Je tiens aussi à remercier mes meilleurs amis Etienne Borgella, Laurine Chapelle, Emmanuel Hebrard et Fanny Spies pour notre amitié. Merci à Alexandre Authier de cultiver mon esprit de compétition. Je remercie aussi mes amis et compagnons de course à pied du midi, Julien Depagneux et Michaël Brunengo.

Je remercie ma famille, en particulier mes parents et mes frères pour tout l'amour et le support qu'ils m'apportent depuis toujours. Enfin, je remercie Carla Vallauri pour son amour et son soutien lors des moments difficiles de la thèse.

En bref, merci à toutes les personnes qui ont pu croiser mon chemin lors de se long et enrichissant voyage qu'est le doctorat !

Contents

CHAPTER 1

Introduction

The Traveling Salesman Problem (TSP) is a fundamental graph theory problem. It consists in finding a minimum cost cycle in a graph visiting all nodes (see Figure 1.1 and Figure 1.2). The TSP appeared in numerous domains such as biology with genome sequencing, industry with scan chains and electronic component drilling problems, positioning of very large telescopes, data clustering, scheduling problems and many others. The applications of the TSP make it as fundamental as interesting: it is often an underlying problem of real-world problems.

Figure 1.1: A set of nodes.

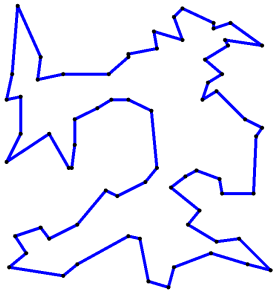


Figure 1.2: A TSP solution for Figure 1.1.

Like many real-world problems, the search for the existence of a TSP solution is NP-Complete and finding the optimal solution is NP-Hard. Thus, all classical methods designed for solving NP-Hard problems have been tried such as Mixed Integer Programming (MIP), Constraint Programming (CP), ...

In order to solve the optimization version of the TSP without side constraints, the most efficient method is based on MIP: the so-called specialized solver Concorde [Applegate et al., 2006]. It is mainly based on an LP relaxation of the TSP obtained by relaxing the integrity and the subtour constraints in combination with structural cutting planes correcting the structural defects of the LP relaxation lower bound. Simple ones such as imposing the 2-connectivity of a graph, and more complex ones such as the so-called Comb inequalities. Nevertheless, no polynomial time algorithm is known at this time to detect whether a graph does not violate a Comb inequality. Thus, a large number of polynomial algorithms have been developed in order to consider only particular cases of Comb inequalities [Edmonds, 1965, Grötschel and Padberg, 1979, Chvátal, 1973, Letchford and Lodi, 2002] leading to strong computational results. Unfortunately, it cannot deal with additional constraints that are very common in real-world problems such as Pickup & Delivery, Dial-a-Ride, automatic harvesting, *etc*.

Solving the TSP is difficult since it involves finding a single cycle going through all the vertices of a graph such that the sum of the edges costs of the cycle is minimal. It is quite easy to model

the fact that each vertex belongs to a cycle. Indeed, it is sufficient that each vertex has at least two distinct neighbors. In other words, each vertex must be the end of at least two edges. Such a result can be obtained by modeling the problem as an assignment problem, which is solved in polynomial time. However, this model is not sufficient to obtain a single cycle in the graph, because the assignment corresponds to the coverage of the vertices by a set of disjoint cycles. From this model, we obtain solutions where each vertex belongs to a cycle, but not to a unique cycle. The covering by a unique cycle can be achieved by imposing that the sub-graph generated by the selected edges is connected. The combination of these two aspects is what makes the TSP so difficult.

Unlike the previous approach, a model can be built based on the notion of a connected sub-graph. It was exactly the idea of Held and Karp [Held and Karp, 1970, Held and Karp, 1971] who represented this notion by a 1-tree that is a node x, two adjacent edges of x and a spanning tree of the graph without x. A 1-tree such that each node has exactly two neighbors is a Hamiltonian cycle, and a minimum 1-tree with these constraints is an optimal solution of the TSP. The use of a 1-tree is interesting because the minimum 1-tree is a strong lower bound of the TSP. In addition, its computation is strongly related to the computation of a minimum spanning tree, that is a polynomial time algorithm.

Held and Karp suggested the relaxation of the degree constraints (*i.e.* the constraints imposing that each node must have exactly two neighbors) with a Lagrangian relaxation (LR). A LR is a relaxation method which approximates P, a difficult problem of constrained optimization, by a simpler problem [Beasley, 1993]. It consists in removing difficult constraints by integrating them into the objective function. It is therefore appropriate for solving problems where the constraints can be partitioned into two parts: a set of constraints that can be easily solved and a set that contains the other constraints. The constraints of the second group are moved to the objective, so only constraints that are easy to solve remain. The satisfaction of difficult constraints is achieved by penalizing them in the objective. A cost for each constraint that measures the distance to satisfaction multiplied by a multiplier is then introduced. For each set of multipliers, the optimal solution of the LR of P is a lower bound of the optimal solution of P. These lower bounds are often used in conjunction with a branch-and-bound algorithm to speed up the search for an optimal solution of P.

Therefore, the Held and Karp Lagrangian relaxation of the degree constraints for the TSP can be defined as follows. The set of the hard constraints is the degree constraint of every node. Then, relaxing these constraints leads in a search for a 1-tree. In short, we have a polynomial problem to solve and some constraints to penalize (the degree constraints). Thus, for each node v, the expression $\mu_i(degree(v) - 2)$ with $\mu \geq 0$ is added to the objective function, where the degree of v is expressed as the number of the edges in the 1-tree with v as an endpoint.

Since for any set of multipliers μ the optimal value of the LR of P is a lower bound of the optimal value of P, it can be used to remove some values of the variables. Consider UB, an upper bound of the optimal solution of P (for example any solution of P, therefore not necessarily optimal), and $x = a$ an assignment. If for $x = a$ the optimal value of the LR of P is greater than UB, then we can remove a from $D(x)$ since we know that $x = a$ does not belong to the optimal solution. From this idea, the CP-based Lagrangian relaxation has been introduced [Sellmann and Fahle, 2003] and successfully used to solve many problems [Khemmoudj et al., 2005, Menana, 2011, Fontaine et al., 2014, Bergman et al., 2015, Cambazard and Fages, 2015, Demassey, 2017]. It consists in modeling the problem so that one or more cost-based filtering algorithms can be used on the easy part of the problem. Difficult con-

straints are moved to the objective function and the cost-based filtering algorithms are used when looking for good multipliers.

The *weighted circuit constraint* (WCC) [Benchimol et al., 2012] implements this approach with the Held and Karp Lagrangian relaxation in constraint programming. This constraint can be considered as the state-of-the-art in CP as mentioned by Ducomman et al. [Ducomman et al., 2016]: "The best approach regarding the number of instances solved and quality of the bound is the Held and Karp's filtering". In addition to the WCC, the state-of-the-art search strategy is a graph interpretation of Last Conflict heuristics [Haralick and Elliott, 1979, Lecoutre et al., 2009], named LCFirst [Fages et al., 2016]. This search strategy selects one node from the graph according to a heuristic. Then, it keeps branching on the edges adjacent to this node until there are no more edges to branch on around the node, no matter if a backtrack occurs or not. Hence, this search strategy learns from previous branching choices and tends to branch in areas which previously lead to a failure. The experiments of Fages et al. [Fages et al., 2016] show that LCFirst clearly outperforms all other search strategies. So far, LCFirst beats all other search strategies by one or more orders of magnitude.

In this thesis, we introduce three constraints into the CP model: the k-cutset constraint, the mandatory Hamiltonian path constraint and the one-tree constraint.

The idea of the k-cutset constraint [Isoart and Régin, 2019, Isoart and Régin, 2021b] is that each cutset of a graph must contain at least two edges and any solution contains an even number of elements from this cutset. Actually, the CP model is based on a single graph variable with mandatory and optional edges. Thus, the k-cutset constraint tries to detect inconsistency in mandatory edges and to filter optional edges. We developed two algorithms for this constraint. A first one with a quadratic complexity dealing with the cutsets of size lower or equal to 3. Then, a linear algorithm dealing with the cutsets of any size. With a static search strategy, the k-cutset constraint allows obtaining an improvement of an order of magnitude over the search nodes and a significant reduction of the solving times. With the state-of-the-art search strategy, it allows a mean reduction of the solving times by at least a factor of 3.

Next, the mandatory Hamiltonian path constraint [Isoart and Régin, 2021b] is based on a local search algorithm. More precisely, it is based on the local search tour improvement k-opt algorithm [Lin, 1965, Lin and Kernighan, 1973]. The idea of this algorithm is to search for k edges in a given tour (not necessarily optimal) such that swapping them creates a cheaper tour. In contrast to heuristic methods, the CP method does not improve a tour but builds an optimal tour. Indeed, the CP model imposes and removes some edges through a branch and bound. The imposed edges, named mandatory edges, can form paths. Therefore, the purpose of the CP method is to find a tour going through these paths. However, it can occur that such a path is not itself optimal (*i.e.* there exists a k-opt). Thus, any solution going through this path will be sub-optimal. Then, the mandatory Hamiltonian path constraint uses the k-opt algorithm on the paths of mandatory edges. More precisely, let us define p, a path composed of mandatory edges going from s to t through a set of nodes X'. If p can be improved by another path p' going from s to t through X', then p cannot belong to any optimal solution and the current branch of the search tree can be pruned. In addition, we define a filtering algorithm removing edges: if a path can be improved when an edge is appended, then it cannot exist an optimal solution simultaneously containing that path and that edge. Experimentally, we show that using a k-opt algorithm with $k > 3$ is impracticable for this constraint. In addition, almost all the improvements are found with $k \leq 3$. Then, integrating this constraint to the WCC combined with the k-cutset constraint allows obtaining an improvement of at least a factor of 3 in solving times.

The one-tree constraint is based on the following property: if all nodes of the 1-tree have exactly two neighbors, then the 1-tree is a solution of the TSP. In some cases, the TSP can be decomposed into independent sub-problems. For instance, let us take v_1 and v_2 two nodes such that removing v_1 and v_2 from the graph (a 2-vertex-connected graph) disconnects it and creates the cut (S, T). Then, we note that there are exactly two nodes connecting S and T and any TSP's solution contains all the nodes of both S and T in a unique minimum cost cycle. Thus, any TSP's solution is a minimum cost Hamiltonian path going from v_1 to v_2 in S combined with a minimum cost Hamiltonian path going from v_1 to v_2 in T. We thus search for 2-cutsets in the graph such that the 1-tree is a minimum cost Hamiltonian path in S (resp. T) in order to assign the part of the 1-tree belonging to S (resp. T).

On the other hand, we engineered the CP-based Lagrangian relaxation in order to improve the solving times [Isoart and Régin, 2020a]. Sellmann made two important observations about the relationship between the LR and the filtering algorithms (FAs) [Sellmann, 2004]:

- Sub-optimal multipliers can be more efficient for filtering than the optimal multipliers for the original problem.
- It is not clear whether FAs should actually take place during the optimization of the Lagrangian multipliers, because the standard approach for the optimization of the multipliers are not guaranteed to be robust enough to enable a change (*i.e.* the removal of a value) of the underlying sub-problem during the optimization.

These observations show the complexity of the interactions between FAs and multipliers, which have important consequences, such as losing the monotonicity so dear to CP. Most of the articles in the literature using CP-based LR do not address this issue and it is only by looking in the source code of the programs that we discover precisely when FAs are called. We then introduced Subgradient Scope Sized algorithm (SSSA) in order to dynamically trigger the FAs. It studies the variation of the value of the LR objective function. Experimentally, we observed that this value often stagnates or oscillates. Then, we measured that these variations do not bring anything in terms of filtering. Therefore, we suggest detecting stagnation and oscillations and immediately stop iterations when they occur. Stopping multipliers computations is not a problem, because convergence towards optimality is done using a search procedure and not only with the LR. However, it is important to note that stopping multipliers computations prematurely can lead to a weaker bound leading to a larger search tree. Thus, a good trade-off is mandatory.

Over the last few years, machines with multiple cores have been commercialized. When the solving time is important, making the best use of your machine is also important. As a result, parallelism has been widely used to improve the solving of some problems. When the solving method uses a branch-and-bound, Embarrassingly Parallel Search (EPS)[Régin et al., 2013, Malapert et al., 2016, Régin and Malapert., 2017] allows parallelism in problem solving in a non-intrusive way. The idea is to statically decompose the initial problem into a large number of sub-problems that are consistent with propagation (*i.e.* running the propagation mechanism on them does not detect any inconsistency). These sub-problems are added to a queue, which is managed by a master. Then, each waiting worker takes a sub-problem from the queue and solves it. The process is repeated until all the sub-problems have been solved. More precisely, EPS decomposes step by step. At each step, sub-problems are decomposed into other sub-problems. The decomposition of a sub-problem is done by calculating a set of variables to be assigned. Then, it generates all sub-problems consistent with the propagation assigning the set of variables to be affected. The

decomposition is done when a specific number of sub-problems is obtained, therefore EPS repeats this step in a breadth-first way. Thus, we define an assignment limit in the enumeration mechanism, which we will increase iteratively until we obtain the required number of sub-problems.

Unfortunately, good results are not obtained from a direct application of EPS for the TSP. Indeed, the state-of-the-art search strategy solving the TSP in CP (LCFirst) is depth-first whereas the decomposition mechanism of EPS is breadth-first. Therefore, the combination of the search strategy and the decomposition of EPS is not straightforward. Experiments show that the use of LCFirst during the decomposition gives results that are not robust. Sometimes the decomposition will be deep enough for LCFirst, but most of the time this is not the case. This issue leads to a bad load balancing and harder sub-problems to solve.

The solving of the TSP cannot be performed without LCFirst, since the results would be deteriorated by several orders of magnitude. Thus, we introduce *Bound-Backtrack-and-Dive*, a method used to approximate the information learned by LCFirst [Isoart and Régin, 2020b]. It consists of running a sequential execution with a low number of backtrack allowed in order to build an ordered set containing the nodes selected by LCFirst in the search tree. The order is defined according to the number of times the nodes are backtracked and how deep they are. Then, the parallel execution of the TSP is started (decomposition step) such that the LCNodes are chosen accordingly to the previous defined order. Finally, the solving step is performed with the classical LCFirst algorithm. The idea comes from our analysis of LCFirst behavior, we notice that it quickly distinguishes some nodes and then refines its knowledge. We then suggest studying the information learned by LCFirst at the beginning of the search tree and to use it to simulate the main trends of LCFirst during the decomposition step of EPS.

EPS is based on the idea that if there are many sub-problems to solve, then the solving times of the workers will be balanced even if the solving times of the sub-problems are not. Unfortunately, sub-problems with extremely different solving times may appear with the TSP. For example, one requiring a huge part of the total solving time. In this case, the load balancing is poor. We show that a general increase of the number of sub-problems does not solve this imbalance. We thus introduce a method that identifies the presence of difficult sub-problems during the solving process and decompose them again. This method keeps the advantages of EPS: the communication is very reduced (the workers do not communicate with each other) and it is independent of the solver.

Finally, we tried to use a "brutal" approach in order to solve the TSP: the shaving. For each non-mandatory edge of the graph, it assigns the edge and runs a propagation step. If a failure occurs, then the non-mandatory edge cannot belong to any optimal solution of the TSP. This process is quite heavy since a propagation step is run for every edge for each search node. Nevertheless, it reduces the search space by several orders of magnitude. Then, we implemented different variations of the shaving in order to find a good trade-off. Unfortunately, this method did not also lead to a reduction of the solving times. However, we believe that a reduction of several orders of magnitude of the search space is not quite common and should be mentioned.

The thesis is organized as follows. First, we introduce and formally define the TSP in Section 2.1. Then, in Section 2.2 and Section 2.3 we present the state-of-the-art for both exact and heuristic solving methods. Next, in Chapter 3 we introduce the TSP model in CP. To do so, we first define the CP background and the WCC with the state-of-the-art search strategy. In Section 3.4, Section 3.5 and Section 3.6 we define respectively the k-cutset constraint, the mandatory Hamiltonian path constraint and the one-tree constraint. In addition, we introduce our works on the CP-based Lagrangian relaxation with SSSA in Section 3.7. Then, in Section 3.8 we show some general results in order to position our works in the world of TSP solving. Next, we show how to

parallelize the search for solutions for the TSP with EPS. In Section 4.2, we formally define EPS and we give some modifications of EPS in order to deal with the TSP model. Then, we introduce *Bound-Backtrack-and-Dive* in Section 4.3 for more robustness and improved results when solving the TSP with EPS. In order to deal with the extremely heterogeneous problems when solving with a large number of cores, we give an algorithm in Section 4.4 allowing re-decomposition of the hardest sub-problems. Then, in Chapter 5 we discuss the shaving method. In Chapter 6, we give some details on our implementation of the TSP model. Finally, we conclude and give some perspectives.

Note that most of the works of this thesis come from publications in international conferences [Isoart and Régin, 2019], [Isoart and Régin, 2020a], [Isoart and Régin, 2020b], [Isoart and Régin, 2020], [Isoart and Régin, 2021a], [Isoart and Régin, 2021b]. In addition, Section 4.4 is still under submission.

1.1 Experimental design

First, all the code of this thesis has been implemented in Java 11. It is integrated in a locally developed constraint programming solver. We consider the instances of the TSPLib [Reinelt, 1991], a library of reference graphs for the TSP. In this library, the name of each instance is suffixed by its number of nodes. For instance, st70 from the TSPLib is a complete graph of 70 nodes. In this thesis, we only consider symmetric graphs. The experiments were performed on Clear Linux with two Intel Xeon E5-2696v2 (12 cores and 24 threads per CPU) and 64 GB of RAM. We use such a machine in order to have stable results. For instance, let us take a laptop on Windows 10 with Intel Core i7-7820HQ CPU @ 2.90 GHz and 32 Go of RAM. Solving three times st70 from the TSPLib give the following solving times: 670ms, 722ms and 619ms. On the Clear Linux with the Xeon, we obtain: 530ms, 524ms, 529ms. Therefore, the machine and the operating system can have an impact on the reproducibility. Note that the laptop we take for the example is a high-performance laptop, *i.e.* a mobile workstation. Later in this thesis we will show results obtained by parallel computing. For the laptops, loading the CPU at 100% causes the heat to increase, which, due to the miniaturization of things, is poorly evacuated. It leads to the following issue: CPUs do not stay at their maximum frequency. Therefore, we observe a decrease of the performance increasing with the solving time. Conversely, the Intel Xeon E5-2696v2 is designed to be loaded at 100%: it achieves better intrinsic performance when fully loaded. Indeed, we must carefully interpret the results when comparing a result on 1 and 100 cores. In addition, Clear Linux allows us to have a machine minimizing the number of threads running for the OS and the internet. Thus, since our goal is to improve the solving times of the TSP in CP, we tried to be as rigorous about the way we experimented as we were about the way we do research.

We run our algorithms on 40 instances from the TSPLib. We do not consider the instances that are solved in less than 100ms and the instances remaining unsolved in 24h with our best model. Therefore, we set a timeout *t.o.* of 24h. In the experimental sections of each chapter, we will express the solving times in seconds and we will denote by #sn the number of search nodes.

CHAPTER 2

The Traveling Salesman Problem (TSP)

In this chapter, we introduce the problem that we consider in this thesis: the Traveling Salesman Problem (TSP). It consists, in a graph, to find the shortest tour going through all the nodes. Then, we introduce the exact solving methods consisting in finding the optimal solution. Finally, we introduce the heuristic solving methods consisting in quickly finding a good solution.

2.1 The problem

The origin of TSP is unclear. We do not really know who first introduced it and the origin of the name is quite obscure. A simple way to define the TSP is the following: for a given set of cities, find the shortest tour going through each city once. Its statement may be simple, but its solving is far from simple. In the 1800s a large number of salesmen were on the road and had to travel to many cities, sometimes very far away, especially when traveling on horseback. At this time, the interest of visiting a number of cities as quickly as possible was very pronounced. For example, a handbook from 1832 [ein alter Commis-Voyageur, 1832] gives various hand-calculated heuristic tours through regions of Germany and Switzerland. It appears that many such handbooks were published in order to guide salesmen through their regions. Today, even if this profession is not as popular as it used to be, the interest in solving a TSP is still fundamental. In the context of travel, whether it is to visit monuments of a city or to go to professional appointment, the TSP is the problem we try to solve. In the 1950s, Flood was one of the first to be interested in the TSP as a mathematical problem with the application of school bus routing [Flood, 1956]. Afterwards, TSP appeared in many other domains such as biology with genome sequencing [Karp and L. Ruzzo, 1996], industry with scan chains [Barbagallo et al., 1996] and electronic component drilling problems [Grötschel et al., 1991], positioning of very large telescopes [Carlson, 1997], data clustering [Lenstra, 1974], scheduling problems, and many others.

For instance, scheduling problems can be modeled as a TSP such that cities are tasks that should be performed and arcs are transition times between tasks. An additional node corresponding to the initial state is often added and each node can be reached from the initial node with no cost. Since each task must be performed, the TSP in this graph is a solution of the scheduling problem performing each task exactly once such that the transition times are minimized. Figure 2.1 shows an example of such a transformation.

i	j		
	0	1	2
0	0	8	9
1	3	0	2
2	4	1	0

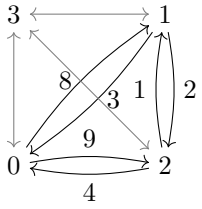


Figure 2.1: A transition table giving the transition times between the task i and j and the corresponding graph such that the initial state is the node 3 connected to each other state with gray arcs and no cost.

The applications of TSP make it as fundamental as interesting: it is often an underlying problem of real life.

2.1.1 Definitions

2.1.1.1 Graph Theory

The definitions of graph theory are taken from Tarjan's book [Tarjan, 1983].

A **directed graph** or **digraph** $G = (X, U)$ consists of a **node set** X and an **arc set** U, where every arc (x_i, x_j) is an ordered pair of distinct nodes. A **multigraph** is a digraph such that there

can exist arcs that are not unique. We let $X(G)$ denote the set of nodes of G such that $n = |X(G)|$ and $U(G)$ the set of arcs of G such that $m = |U(G)|$. In addition, $U(i)$ is the set of adjacent edges of i. The **cost** of an arc is a value associated with the arc. An **undirected graph** is a digraph such that for each arc $(x_i, x_j) \in U$, $(x_i, x_j) = (x_j, x_i)$. If $G_1 = (X_1, U_1)$ and $G_2 = (X_2, U_2)$ are graphs, both undirected or both directed, G_1 is a **subgraph** of G_2 if $X_1 \subseteq X_2$ and $U_1 \subseteq U_2$. A **path** from node x_1 to node x_k in G is a list of nodes $[x_1, \ldots, x_k]$ such that (x_i, x_{i+1}) is an arc for $i \in [1..k-1]$. The path **contains** node x_i for $i \in [1..k]$ and arc (x_i, x_{i+1}) for $i \in [1..k-1]$. The path is **simple** if all its nodes are distinct. The path is a **cycle** if $k > 1$ and $x_1 = x_k$. A cycle is **Hamiltonian** if $[x_1, \ldots, x_{k-1}]$ is a simple path and contains every node of X. The **cost** of a path p, denoted by $w(p)$, is the sum of the costs of the arcs contained in p. An undirected graph G is **connected** if there is a path between each pair of nodes, otherwise it is **disconnected**. A **tree** is a connected graph without a cycle. A tree $T = (X', U')$ is a **spanning tree** of G if $X' = X$ and $U' \subseteq U$. The U' edges are the **tree edges** T and the $U - U'$ edges are the **non-tree edges** T. A **minimum spanning tree** $T = (X', U')$ is a spanning tree minimizing the cost of the tree edges. The maximum connected subgraphs of G are its **connected components**. A partition (S, T) of the nodes of G such that $S \subseteq X$ and $T = X - S$ is a **cut**. The set of edges $(x_i, x_j) \in U$ having $x_i \in S$ and $x_j \in T$ is the **cutset** of the (S, T) cut. A k**-cutset** is a cutset of cardinality k.

Definition 2.1.1 (TSP)**.** Given a graph G, a solution to the **Traveling Salesman Problem (TSP)** in G is a Hamiltonian cycle $HC \in G$ minimizing $w(HC)$.

Without loss of generality, we will assume that each arc appears at most once and that there is no self-loop, *i.e.* an arc (u, v) such that $u = v$. Unless specified, we will only consider the search for TSP in undirected graphs.

2.1.1.2 Model

In the context of a graph, a solution of the TSP is a unique minimum-cost cycle going through all the nodes. A classical formulation of the TSP with integer variables is given in the following model:

$$min \sum_{e \in U} w(e) x_e \tag{2.1}$$

s.t.

$$\sum_{e \in U(i)} x_e = 2 \qquad \forall i \in X \tag{2.2}$$

$$\sum_{i \in S, j \in S, i<j} x_{(i,j)} \leq |S| - 1 \qquad \forall S \subset X, |S| > 2 \tag{2.3}$$

$$x_e \in \{0, 1\} \qquad \forall e \in U \tag{2.4}$$

If e belongs to the solution, then $x_e = 1$. Otherwise, $x_e = 0$. The degree constraint (2.2) requires that each node $i \in X$ has exactly two adjacent arcs in the solution: it means that each node must belong to a cycle. Next, the subtour constraint (2.3) requires that there is no cycle formed by a subset of the nodes, except for the subset containing all the nodes of the graph: that enforces the unicity of the cycle. Finally, the TSP is a minimization problem and the objective function (2.1) minimize the cost of the solution. Thus, by solving this problem we obtain an integer solution containing a unique cycle going through all the nodes of the graph.

2.2 Exact solving

A straightforward method for solving a problem is to try all possible combinations. For the TSP, it means $O(n!)$ combinations: it quickly turns to be impractical. Indeed, the time complexity for the optimization version of the TSP is NP-hard while the decision version of the TSP is NP-Complete.

Several algorithms using dynamic programming theory [Bellman et al., 1957] have been developed to solve the TSP [Bellman, 1958, Gonzales, 1962, Held and Karp, 1962]. In an optimal solution of the TSP, each sub-path is an optimal path itself. Starting from this remark, the idea is to compute all sub-paths starting with the smallest sub-path: the one visiting a single node. Recursively, for each path already computed, we add a node not yet visited at its end. When all the nodes are visited, we add to the path the cost of the edge returning to the starting node. Finally, the optimal solution is obtained by taking the smallest computed paths. An analysis led by Held and Karp shows that this algorithm has a time complexity in $O(n^2 2^n)$. Even though the time complexity is huge for practical applications, it is much better than the factorial obtained by enumerating all combinations: it saves an exponential in the time complexity. Moreover, it is a very nice theoretical result: even today, no known algorithm has a better time complexity for the exact solving of the TSP. However, it remains impractical even for small instances. We will now describe some exact solving methods giving better practical results.

2.2.1 Relaxations

Another way to solve NP-Hard problems is to relax hard constraints (*i.e.* the ones making the problem NP-Hard) in order to exploit substructures of the problem. Looking more closely at the model given in Section 2.1.1.2, we can relax some constraints in order to obtain a polynomial sub-problem to solve. Indeed, if we relax the constraint (2.2) by replacing it by 2.5, then the solution is a minimum-cost connected subgraph without subtours: it is a minimum spanning tree and a large number of polynomial algorithms have been developed for this problem [Borůvka, 1926, Kruskal, 1956, Prim, 1957, Karger et al., 1995, Chazelle, 2000].

$$\sum_{e \in U(i)} x_e > 0 \; \forall i \in X \tag{2.5}$$

If we relax the (2.3) constraint by removing it, then we try to solve a min-cost assignment problem solved in a polynomial time by a minimum cost flow problem [Edmonds and Karp, 1972] or the Hungarian algorithm [Kuhn, 1955]. Due to the exponential number of constraints of (2.3), solving the linear programming relaxation by relaxing (2.4), *i.e.* replacing (2.4) by $0 \leq x_e \leq 1$, may take an exponential time. Thus, the combination of (2.2) and (2.3) make the TSP problem very hard to solve.

2.2.1.1 Lagrangian Relaxation

The Lagrangian Relaxation (LR) procedure uses the idea of relaxing some difficult constraints by bringing them into the objective function with associated Lagrangian multipliers $\mu \geq 0$. The application of LR to a mixed integer program can be defined as follows.

$$\begin{array}{l} Z = \min c \cdot x \\ \text{s.t.} \left\{ \begin{array}{l} A_1 \cdot x \leq b_1 \\ A_2 \cdot x \leq b_2 \\ x \in X \end{array} \right. \end{array} \longrightarrow \begin{array}{l} Z_{LR}(\mu) = \min c \cdot x + \mu(A_1 \cdot x - b_1) \\ \text{s.t.} \left\{ \begin{array}{l} A_2 \cdot x \leq b_2 \\ x \in X \end{array} \right. \end{array}$$

Assume that the constraint $A_1 \cdot x \leq b_1$ is difficult to solve whereas constraint $A_2 \cdot x \leq b_2$ is easy. Then, the LR moves the first one into the objective. If $A_1 \cdot x \leq b_1$ is violated, then $A_1 \cdot x > b_1$ and so $d = A_1 \cdot x - b_1 > 0$. This value d measures the distance to the satisfaction of this constraint. Intuitively, the larger d is, the more the constraint should be penalized and the smaller d is, the less the constraint should be penalized. This result is obtained by adding the value $(A_1 \cdot x - b_1)$ in the objective function. The Lagrangian relaxation proposes to use a non-negative multiplier μ for each constraint introduced in the objective.

The interest of the multipliers is shown by the following property:

Property 2.2.1. For any vector μ, the value of $Z_{LR}(\mu)$ is a lower bound of Z.

The Lagrangian multiplier problem consists of searching for the best multipliers. The two most popular types of methods for solving it are the subgradient and the bundle methods [Frangioni, 2002]. This second type of method converges faster than the first one. Since we only need to use suboptimal multipliers for our filtering algorithms, we will focus our attention on the subgradient method.

2.2.1.2 Subgradient Algorithm

Subgradient algorithms work in steps and locally re-optimize the multipliers according to a certain precision, called agility. We will denote by $LR(P, \mu)$ the LR of P associated with the multiplier set μ. Beasley's algorithm [Beasley, 1993] is one of the most widely used. Its structure is depicted in Algorithm 1. It calls Function SOLVELR which computes the optimal value of the LR for a given set of multipliers and define new multipliers. The number of agility values is defined by $\#agility$, which is close to 6 most of the time for the TSP solving. Usually, and as mentioned in Algorithm 1, the agility starts at 2 and is divided by 2 at each iteration. For a given agility value, $\#scope$ is the maximal number of internal iterations of the LR. For each value of scope, the optimal value of the LR is computed and multipliers are updated accordingly. We have also added Function STOPCONDITION(...), which takes some parameters and tests if some stopping conditions of the current loop are met. For instance, the program can be stopped when there is no more progression of the objective function value.

2.2.1.3 Held-Karp Relaxation

Held and Karp introduced a strong lower bound obtained through a Lagrangian relaxation of the degree constraint (2.2) [Held and Karp, 1970, Held and Karp, 1971]. It consists in solving a succession of MST. Nevertheless, they noticed that a TSP is not an MST since deleting an edge of the TSP does not disconnect the graph. However, deleting an edge of the MST disconnects it: the MST does not have the notion of a cycle. To fix this issue, they suggested the following process for a graph $G = (X, U)$: choose a node $u \in X$, named the *1-node*, search for an MST in $G' = (X - \{u\}, U)$ and connect u to the MST by its two lowest cost edges. Thus, we obtain a subgraph $G_{1T} = (X, U_{1T})$ named a *1-tree* that contains a unique cycle and cover G with minimum cost. An example from [Benchimol et al., 2012] is depicted in Figure 2.2.

The last issue is that all the nodes do not necessarily belong to this cycle unless the degree constraint (2.2) is satisfied. Thus, they suggested to associate for each node i a Lagrangian multiplier $\mu_i \geq 0$. If $U_{1T}(i) > 2$, then μ_i is increased (there must be fewer edges in the neighborhood of i). Otherwise, if $U_{1T}(i) < 2$, then μ_i is decreased (there must be more edges in the neighborhood of i). At each iteration of the Lagrangian relaxation, the edge cost is updated as follows:

Algorithm 1: Beasley's algorithm.

subgradientBeasley *(P,Z_{ub},μ)*

Input: A problem P, an upper bound Z_{ub} of P and a set of multipliers μ.
Output: A set of multipliers μ^{k+1} and x^k the optimal solution of LR(P,μ^k).

$\pi \leftarrow 2$; // Subgradient agility
$k \leftarrow 0$;
$\mu^0 \leftarrow \mu$; // Start with the current values of multipliers
foreach $iterAgility = 1 \dots \#agility$ **do**
  $scope \leftarrow 0$;
  while $scope < \#scope$ **do**
    $(\mu^{k+1}, x^k, Z^k) \leftarrow$ SOLVELR(P,Z_{ub}, π, μ^k, k) ;
    // Return the optimal solution of P if reached
    if $Z^k = Z_{ub}$ **then return** (μ^{k+1}, x^k);
    $k \leftarrow k + 1$;
    $scope \leftarrow scope + 1$;
    if STOPCONDITION(...) **then** break;
  $\pi \leftarrow \pi/2$;
return (μ^{k+1}, x^k)

solveLR *(P,Z_{ub}, μ^k, k)*

Input: A problem P, an upper bound Z_{ub} of P, the set of multipliers μ^k obtained in the k^{th} iteration and k the number of performed iterations.
Output: A set of multipliers μ^{k+1}, the optimal solution x^k of LR(P,μ^k) and the objective value Z^k of LR(P,μ^k).

$x^k \leftarrow$ solve $LR(P, \mu^k)$ to optimality ;
$R \leftarrow |\mu|$; // Number of relaxed constraints
$Z^k \leftarrow obj(x^k) + \sum_{1 \leq r \leq R} \mu_r^k obj_r(x^k)$;
$\Delta^k \leftarrow \frac{\pi(Z_{ub} - Z^k)}{\sum_{1 \leq r \leq R} (obj_r(x^k))^2}$; // Step
// Update of the multipliers
$\forall 1 \leq r \leq R : \mu_r^{k+1} \leftarrow \max(0, \mu_r^k + \Delta^k obj_r(x^k))$;
return (μ^{k+1}, x^k, Z^k)

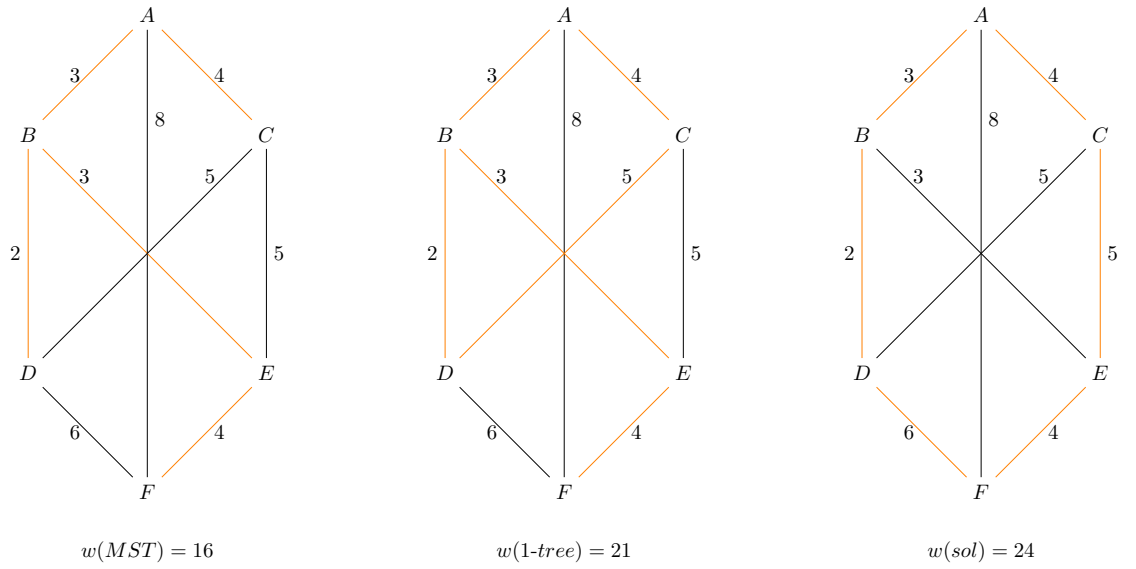


Figure 2.2: A comparison between an MST, a 1-tree and the optimal tour in a graph.

$\forall (i,j) \in U, w'((i,j)) = w(e) + \mu_i + \mu_j$ such that the function $w(e)$ returns the initial cost of an edge e and the function $w'(e)$ is the updated cost of an edge e. Here we look for the set of μ such that the degree constraint is satisfied. In summary, we compute a succession of minimum 1-tree with a Lagrangian relaxation. An example [Benchimol et al., 2012] is shown in Figure 2.3.

Experimentally, [Applegate et al., 2006] showed in an example of a problem of 42 cities that the linear relaxation gives a bound at 9% of the optimal and the computation of a minimum 1-tree gives a bound at 14% of the optimal. However, the Lagrangian relaxation allows obtaining, with a modest number of iterations, a bound at 0.4% of the optimal solution. Then, given the quality of the Held-Karp bound, it is often used to evaluate the quality of heuristic methods. Since the convergence of a Lagrangian relaxation process can be very slow and the non-monotonicity of subgradient algorithms, then Held and Karp also suggested to combine this method with a branch-and-bound [Held and Karp, 1971].

2.2.1.4 Assignment problem relaxation

Another bound is the one obtained by relaxing the constraint (2.3), it allows the graph to be covered by multiple cycles. As we explained above, solving this relaxed problem is equivalent to solving the minimum cost assignment problem that can be solved with the Hungarian algorithm [Kuhn, 1955] by searching for each node of the graph a minimum cost assignment with its successor. Another way to proceed is to create a graph G' such that for each node x of the initial graph, we create two nodes x_1 and x_2. Then, all the nodes of the initial graph are duplicated. For each edge (a, b) of the initial graph, we create the edges (a_1, b_2) and (a_2, b_1). Therefore, G' is a bipartite graph such that the nodes with 1 in index belong to a partition and the ones with 2 in index belong to the other partition. Then, using a flow algorithm [Ford, 1956, Bellman, 1958], we can compute a minimum cost assignment in by setting the capacity of the edges to 1 and sending

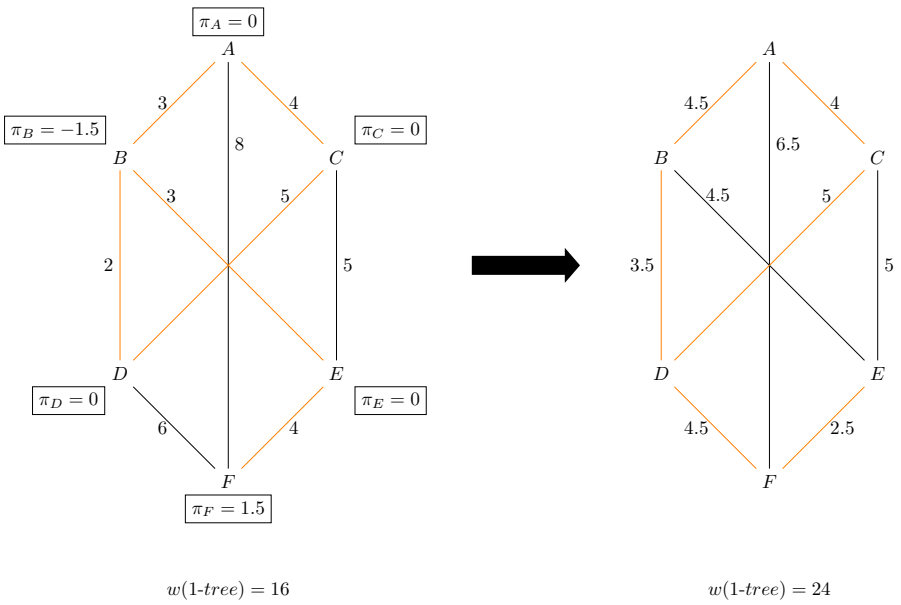


Figure 2.3: An iteration of the Held and Karp Lagrangian relaxation in a graph. In the left graph, we give a 1-tree and for each node the associated multiplier π_i. In the right graph, the edges cost have been updated and a new 1-tree has been computed leading to a better lower bound.

1 flow unit per node in a part of the bipartite. The issue with these two approaches is that they allow coverage by cycles of size two, and thus do not give sufficiently good bounds. One way to fix this issue is to use 2-flow, *i.e.* to send 2 flow units on each node of a part of the bipartite graph. Then, it looks for a predecessor and a successor. The solution has $2 * n$ edges, so it creates two solutions: one in one direction and one in the other. Thus, the value of the obtained bound must be divided by two. In practice, the 2-flow gives much better bounds than the 1-flow (sometimes better than a simple 1-tree). However, when we have a cycle covering, we want to forbid these cycles in order to obtain only one cycle covering all the nodes. With the Lagrangian relaxation method, we will add a Lagrangian multiplier per cycle found (potentially an exponential number) with an inequality constraint. For example, if we the 2-flow generates the cycle $a - b - c - a$, the constraint $x_a + x_b + x_c < 3$ is generated and added into the objective function. Conversely to the 1-tree with the equality degree constraints, we have no longer a link with the objective function of the initial problem and the relaxed one. It can be annoying for the convergence and for some applications such that filtering in Constraint Programming. Some methods suggest to dynamically purging the constraints when they are trivially solved in order to minimize this issue.

A stronger algorithm than the 2-flow is the 2-matching [Edmonds, 1965]. It consists in finding a matching in the graph, that is a subset of edges, such that each node of the graph has exactly two adjacent edges in the matching. Jünger et al. [Jünger et al., 1995] wrote: “implementing an algorithm for the minimum cost 2-matching problem efficiently is not as simple as for the minimum cost spanning tree problem. The lower bound obtained by finding the minimum cost 2-matching is in general poor”. For these reasons, we did not investigate this idea that much.

2.2.2 LP relaxation

2.2.2.1 Cutting planes

In any relaxation of a problem P into P', the optimal solution x^*_P of P is feasible in P' but the optimal solution $x^*_{P'}$ of P' is not necessarily feasible in P. Specifically, $x^*_{P'}$ is feasible in P if and only if $x^*_{P'}$ is the optimal solution of P. With this in mind, for an integer programming problem P and its linear relaxation P', the idea of the cutting planes [Dantzig et al., 1954] is to refine P' by solving it iteratively until $x^*_{P'}$ become a feasible of P. To do so, we create at each iteration some new constraints such that x^*_P remains valid in P' but $x^*_{P'}$ violates one of the created constraints. In the TSP case, the simple linear relaxation of the problem is not really simpler because there can be an exponential number of subtours, *i.e.* an exponential number of constraints. Thus, Dantzig et al. suggested applying the cutting plane method to the TSP relaxation such that the integrity (2.4) and subtour (2.3) constraints are relaxed [Dantzig et al., 1954].

2.2.2.2 Constraint generation

Several approaches have been developed in order to generate constraints for the cutting plane method. In a general context, it can be done for mixed-integer linear programming problems [Gomory, 1958]. For the TSP solving, Dantzig et al. were interested in the structural defects of the lower bound obtained by the relaxation introduced in Section 2.2.2.1 where the integrity (2.4) and subtour (2.3) constraints are relaxed [Dantzig et al., 1954]. For example, if $x^*_{P'}$ contains subtours then constraints that forbid them are generated. We note that constraints are generated as they needed. Moreover, if $x^*_{P'}$ is not a 2-connected graph (deleting 1 node disconnects the graph), then a constraint enforcing that the graph must be 2-connected is generated. However, this may not be sufficient to optimally solve the TSP. Thus, more sophisticated constraints named Comb inequalities are generated.

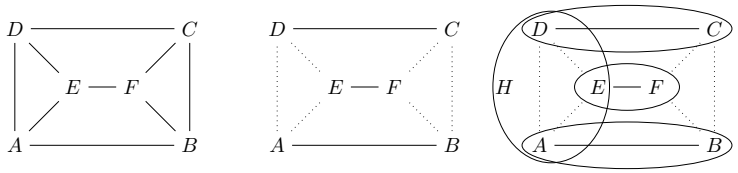
As stated in Cook's book [Cook et al., 2011]: "A comb is defined by giving several subsets of nodes of the graph: We need one nonempty handle $H \subseteq V$, $H \neq V$ and $2k+1$ pairwise disjoint, nonempty teeth $T_1, T_2, ..., T_{2k+1} \subseteq V$. for k at least 1. (So the number of teeth is odd and at least 3.) We also require each tooth to have at least one node in common with the handle and at least one node that is not in the handle."

Formally, Comb inequalities can be defined by Equation 2.6.

$$\sum_{e \in U(H)} x_e + \sum_{i=1}^{2k+1} x(T_i)) \leq |H| + k \tag{2.6}$$

An example is given in Figure 2.4. Given $H = \{(A, D, E)\}, T = \{\{D, C\}, \{E, F\}, \{A, B\}\}$ and $k = 1$, the Comb inequality is violated since replacing in Equation 2.6 we obtain $3 + 3 \leq 3 + 1$, which is wrong. Then, the cutting plane $\sum_{e \in U(H)} x_e + \sum_{i=1}^{2k+1} x(T_i)) \leq 4$ is generated.

Nevertheless, no polynomial algorithm is known at this time to detect whether a subgraph $x^*_{P'}$ violates a Comb inequality. However, many polynomial time algorithms have been developed to detect particular cases of Comb [Edmonds, 1965, Grötschel and Padberg, 1979, Chvátal, 1973, Letchford and Lodi, 2002]. For instance, the blossom inequality [Edmonds, 1965] is a Comb where each tooth of a Comb has exactly two nodes. Finally, all these techniques are efficiently implemented in the Concorde [Applegate et al., 2006] code, which is to date the best pure TSP solver.



1. Graph G 2. A feasible solution of G 3. Handle and teeth of G

Figure 2.4: An example of comb inequalities. The left graph is the input graph such that each edge has a cost equal to one. The middle graph is a solution of the relaxation of Dantzig et al. such that a solid line is equal to one and a dot line is equal to 1/2. The right graph is an example of a violated comb inequality for the solution of the middle graph.

2.3 Heuristic solving

In contrast to exact algorithms, heuristics compute a TSP tour without a guarantee that the tour is optimal. However, they are usually much faster. Thus, if a good solution can be accepted, heuristics can be more suitable. For the TSP, there are three different kinds of heuristics: the *tour construction algorithms* iteratively add nodes in the current tour in order to find a solution, the *tour improvement algorithms* improve a tour with cities order exchange and *composite algorithms*, that combine the two approaches. In this section, we show some heuristics for the three approaches and we compare them with the Held-Karp bound. More heuristics are detailed in [Gutin et al., 2001, Rosenkrantz et al., 1997].

2.3.1 Tour construction algorithms

2.3.1.1 Nearest neighbor algorithm

The Nearest Neighbor (NN) algorithm is one of the first, and surely the most natural, heuristic used to solve the TSP. As described by Flood [Flood, 1956], it consists in building a tour starting from a node and iteratively visit the nearest neighbor not yet visited until all nodes are visited, then returning to the starting node. It leads to a time complexity in $O(n^2)$. This algorithm is quite greedy and often gives solutions such that the returning edge is costly. However, Rosenkrantz et al. showed that if the triangular inequality is satisfied, then the tour found is at most $O(log(V))$ longer than the optimal tour [Rosenkrantz et al., 1974]. This heuristic may be too greedy, but Johnson and McGeoch showed that the bound is less than 20% above Held-Karp bound [Johnson and McGeoch, 2008].

2.3.1.2 Christofides

Christofides' algorithm [Christofides, 1976] was one of the very first approximation algorithms. It gives a TSP tour which is at most $3/2$ longer than the optimal one if the graph is metric and the triangular inequality holds. For a graph $G = (X, U)$, the algorithm starts by computing $T = (X, U')$, an MST. Then, it computes a minimum weight perfect matching $M = (X', U'')$ for the subgraph induced by the nodes with an odd degree in T. Next, it computes H an Eulerian cycle in $G' = (X, U' \cup U'')$. Note that several nodes can be visited more than once in

H. Thus, we delete in the order H the "already visited" nodes, that gives us an approximated TSP tour. Because of the minimum weight perfect matching, the time complexity of the algorithm is in $O(n^3)$. Finally, Christofides' algorithm may be slower than the nearest neighbor algorithm but Johnson and McGeoch showed that the bound is less than 10% above Held-Karp bound [Johnson and McGeoch, 2008].

2.3.2 Tour improvement algorithms

In the previous section, we have presented some TSP tour heuristics. What if we want a better tour? That is exactly the purpose of tour improvement algorithms. It takes as input a tour and iteratively tries to improve it. The most popular form of tour improvement is the local search methods of 2-opt and 3-opt.

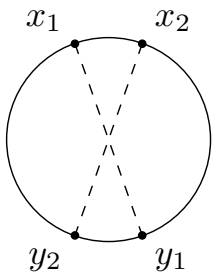


Figure 2.5: An example of 2-opt. The circle represents a tour where dashed lines are the suggested move for the pair of edges $((x_1, x_2), (y_1, y_2))$.

The 2-opt heuristic proceeds as follows. Given a tour T, for each pair of edges (e_1, e_2) in T, if replacing (e_1, e_2) by another pair of edges (e_3, e_4) of T leads to a connected and shorter tour, then we can replace (e_1, e_2) with (e_3, e_4) in T. We name such a replacing procedure a move. Note that some heuristics search for the best improving move and then does not perform the replacement of a move when found. In addition, for each pair of edges, there exists only one move reconnecting the graph which is not the null move. The iteration on pairs leads to a time complexity in $O(n^2)$. Figure 2.5 shows an example where $e_1 = (x_1, x_2)$, $e_2 = (y_1, y_2)$ and the move is $e_3 = (x_1, y_1)$, $e_4 = (x_2, y_2)$. Johnson and McGeoch reported that 2-opt algorithm usually improve the tour such that the bound is less than 5% above Held-Karp bound [Johnson and McGeoch, 2008].

2.3.2.1 3-opt algorithm

For the 3-opt algorithm, instead of choosing pair of edges, we choose a triplet of edges and, as for 2-opt, we search for moves reducing the overall cost of the tour. In that case, there are seven ways to reconnect the graph. Note that three of them are simple 2-opt (that is a combination with one edge of the triplet not moved). Thus, 3-opt allows checking more sophisticated combination than 2-opt and then can potentially find better moves. However, it leads to an algorithm with a time complexity in $O(n^3)$. Figure 2.6 shows an example of all 3-opt moves that are not 2-opt. Johnson and McGeoch showed that 3-opt algorithm usually improves the tour such that the bound is less than 3% above Held-Karp bound [Johnson and McGeoch, 2008]. Thus, we notice that increasing the complexity of checked combination can lead to a slight improvement of the tour quality.

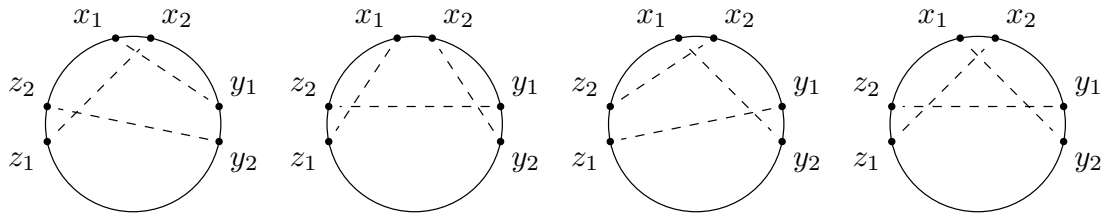


Figure 2.6: 3-opt moves such that the circle represents a tour where dashed lines are the suggested move for the triplet of edges $((x_1, x_2), (y_1, y_2), (z_1, z_2))$.

2.3.2.2 k-opt

Naturally, the 2-opt and the 3-opt algorithms can be generalized to the k-opt algorithm with a time complexity in $O(n^k)$. Experiments have shown that increasing the value of k improves the quality of the tours but slows down solving times. Thus, some methods [Or, 1977, Bentley, 1992] consider some 3-opt and/or 4-opt, but not all, in order to reduce the time complexity and speed up the solving times.

2.3.3 Composite algorithm: Lin-Kernighan

Lin and Kernighan suggested using a variable k while solving [Lin, 1965, Lin and Kernighan, 1973] in order to include larger moves. This makes the algorithm more complex, but greatly improves the results (tour quality and solving times). To do so, they suggested several rules. They restrict the considered exchanges by looking at the most promising permutations only. In addition, they allow improving k-opt moves that can be built from a sequence of 2-opt moves such that some 2-opt moves does not improve the tour. These moves are much more complex and provide better moves than a simple run of the 2-opt algorithm. Next, Helsgaun reported that the heuristic of Lin-Kernighan allows obtaining tours very close to optimal with a single run for 50 cities [Helsgaun, 2000]. When 100 cities are involved, a single run allows obtaining tours above 20-30% of the Held-Karp bound. However, multiple runs allow obtaining a tour very close to optimal. In order to make this algorithm extremely efficient, Helsgaun [Helsgaun, 2000] has remarkably refined most of the rules given by Lin and Kernighan [Lin and Kernighan, 1973]. Today, the Lin-Kernighan-Helsgaun (LKH) algorithm is considered as one of the most efficient heuristic solving the TSP and therefore it is embedded in most of the exact methods.

CHAPTER 3

The TSP in CP

The TSP in Constraint Programming (CP) is mainly based on the Weighted Circuit Constraint (WCC) itself based on the 1-tree Lagrangian relaxation of Held and Karp. In combination with this model, a branch and bound is used. Thus, at any moment we search for a TSP with a lower bound. Then, we introduced three new constraints. First, the k-cutset constraint looks for the cutsets of the graph and enforces that any solution has a strictly positive and even number of edges in each cutset. Next, we defined the Mandatory Hamiltonian path constraint. It uses the well-known local search algorithms 2-opt and 3-opt on the lower bound imposed by the branch and bound and the filtering algorithms. Then, we defined another structural constraint based on the minimality of some cut in the graph and the lower bound obtained by the 1-tree. Finally, we engineered the Lagrangian relaxation in order to speed up the solving times. All of this, leads to reducing the number of search nodes by several orders of magnitude and therefore improving the solving times.

3.1 Constraint Programming

In this thesis, we use Constraint Programming (CP) [Rossi et al., 2006] in order to solve the TSP. The CP is a programmation paradigm designed to solve combinatorial problems. This consists in modeling the problem under constraints, and using each constraint to reduce the search space of solutions.

3.1.1 Modeling

A finite **constraint network** $\mathcal{N}$ is defined as a set of n **variables** $X = \{x_1, \ldots, x_n\}$, a set of current **domains** $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible **values** for variable x_i, and a set $\mathcal{C}$ of **constraints** between variables. We introduce the particular notation $\mathcal{D}_0 = \{D_0(x_1), \ldots, D_0(x_n)\}$ to represent the set of initial domains of $\mathcal{N}$ on which constraint definitions were stated.

A **constraint** C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ that specifies the **allowed** combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ is named a **tuple on** $X(C)$. A value a for a variable x is often denoted by (x, a). Let C be a constraint. A tuple τ on $X(C)$ is **valid** if $\forall (x, a) \in \tau, a \in D(x)$. C is **consistent** iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is **consistent with** C iff $x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ with $(x, a) \in \tau$. A constraint is **arc consistent** iff $\forall x_i \in X(C), D(x_i) \neq \emptyset$ and $\forall a \in D(x_i)$, a is consistent with C. A **set-variable** x_i is a particular variable such that its domain is a set of set defined by $D(x_i) = \{S \mid lb(x_i) \subseteq S \subseteq ub(x_i)\}$. The domain can also be represented by the sets $lb(x_i)$ and $ub(x_i)$ such that $v \in lb(x_i)$ iff $\forall S \in D(x_i), v \in S$ and $v \in ub(x_i)$ iff $\exists S \in D(x_i), v \in S$. We say that $lb(x_i)$ is the set of mandatory elements, and $ub(x_i) \setminus lb(x_i)$ is the set of optional elements. Usually a set-variable is also associated with an interval representing its cardinality. A **graph variable** is a variable associated to a graph composed of two set-variables: one for the nodes, one for the edges.

3.1.2 Filtering

For each constraint, a filtering algorithm is associated. It allows removing the values from the domain of the variables of the constraint such that the constraint cannot be satisfied with these values. For instance, if there are two variables x and y such that $D(x) = [5, 10]$ and $D(y) = [0, 7]$ constrained by $x < y$, then a filtering algorithm may remove the values from 7 to 10 from $D(x)$ and the values from 0 to 5 from $D(y)$. Therefore, the problem to solve becomes $x < y$ such that $D(x) = [5, 6]$ and $D(y) = [6, 7]$. Thus, there are two valid solutions: $S_1 = \{D(x) = \{5\}, D(y) = \{6\}\}$ and $\{D(x) = \{6\}, D(y) = \{7\}\}$. In addition, solving optimization problems such as the TSP introduce an objective function. For instance, if the objective function minimizes the sum of the values of the domains, S_1 would have been the only optimal solution since 5 + 6 < 6 + 7. In the TSP, the objective function minimizes the sum of the mandatory edges cost.

Frequently, cost-based filtering algorithms are developed from the objective function. Given P a minimization problem, $LB(P)$ and $UB(P)$ a function computing respectively a lower bound and an upper bound of P and (x, a) a pair (variable,value). If $cost(LB(P)) \leq cost(UB(P))$ and $cost(LB(P_{x=a}) > cost(UB(P))$, then $x = a$ cannot belong to an optimal solution because there is no lower bound with a cost lower or equal than the upper bound and $x = a$. Then, a

can be safely removed from the domain of x. Conversely, if $cost(LB(P)) \leq cost(UB(P))$ and $cost(LB(P_{x \neq a}) > cost(UB(P))$, then $x = a$ is necessary in order to verify $cost(LB(P)) \leq cost(UB(P))$. Then, $x = a$. In the case of a graph variable represented with a set variable (such as for the TSP), we want to find values to remove from the optional set and to find elements of the optional set to add in the mandatory set.

3.1.3 Propagation

A variable can belong to several constraints. Therefore, if the filtering algorithm of a constraint removes some values from the domain of a variable, then the filtering algorithm of other constraints implying the variable should study the modifications. This mechanism is named the propagation. For instance, let us take the three variables x, y and z such that $D(x) = [5, 10]$, $D(y) = [0, 7]$ and $D(z) = [5, 7]$ with the two constraints $x < y$ and $y < z$. The filtering algorithm of $x < y$ will deduce that $D(x) = [5, 6]$ and $D(y) = [6, 7]$ and the one of $y < z$ will deduce that $D(y) = \{6\}$ and $D(z) = \{7\}$. Then, y and z will be assigned. Then, the propagation mechanism allows studying y again for $x < y$ since $D(y)$ has been modified by $y < z$. Therefore, $x < y$ will deduce that $D(x) = \{5\}$.

3.1.4 Search

For the Constraint Satisfaction Problem (CSP), a solution is found when all the variables are assigned. Moreover, the CP can solve optimization problems, which is the case of the TSP. Then, the solutions solve the CSP while minimizing an objective function. When the propagation step is over (there is no more deduction to do), a search strategy is used in order to find all the solutions. To do so, the most common way is to select with some heuristics a pair (variable,value). Then, split the current problem in two sub-problems: the one with the variable assigned to the value and the one with the value removed from the domain of the variable. We then perform a complete search of the solutions set. For instance, the search strategies for the TSP consider edges of the graph. In the case of the TSP and many other optimization problems, an upper bound of the objective function is computed. Therefore, if the current values of the domains could not lead to a solution lower or equal to this upper bound, then the search of solutions in that part of the search tree is aborted. The process of stopping the search in the search tree is named a failure, it happens when the current sub-problem is inconsistent.

3.2 The Weighted Circuit Constraint

Nowadays, the Weighted Circuit Constraint (WCC) is the best way to model the TSP in CP. It is mainly based on the 1-tree Lagrangian Relaxation (LR) of Held and Karp [Held and Karp, 1970, Held and Karp, 1971] introduced in Section 2.2.1.3. However, experiments show a very slow convergence toward the optimal solution. Thus, the WCC integrates the following filtering algorithms based on the edges costs:

- Marginal cost: if an edge e does not belong to any 1-tree with cost smaller or equal than a given upper bound, then e can be safely deleted.
- Replacement cost: if an edge e belongs to all 1-trees with cost smaller or equal than a given upper bound, then e is mandatory.

Note that the upper bound can be obtained very efficiently before solving with an LKH run [Lin and Kernighan, 1973, Helsgaun, 2000].

The WCC also integrates the degree constraint that is a structural constraint imposing that each node has exactly two neighbors in any solution of the TSP.

In addition, the WCC is modeled with a single undirected graph variable such that all nodes are mandatory. Without loss of generality, we note $G_{init} = (X, M_{init}, O_{init})$ the input graph such that M_{init} is the set of mandatory edges and O_{init} is the set of optional edges. While solving, some edges will be removed or become mandatory. Therefore, we introduce D the set of removed edges. An edge can become mandatory and therefore it is removed from O_{init} and added to M_{init}. In addition, a removed edge is removed from O_{init} and added to D.

For the sake of clarity, we define $G = (X, M, O)$ as the graph G_{init} such that some modifications have been made by the search tree and/or the filtering algorithms. Therefore, we have $M_{init} \subseteq M \subseteq (O_{init} \cup M_{init})$ and $O \subseteq O_{init}$. Note that if $G' = (X, U)$ is the graph such that G_{init} is its initial graph variable, then $O \cup M \cup D = U$, $O \cap M = \emptyset$, $O \cap D = \emptyset$ and $M \cap D = \emptyset$. When a solution is found, $|M| = n$ and $O = \emptyset$. In addition, we define $G_{mand} = (X, M, \emptyset)$ the graph of mandatory edges and $G_{opt} = (X, \emptyset, O)$ the graph of optional edges. We note $M(i)$ (resp. $O(i)$) the set of mandatory (resp. optional) edges having i for extremity in M (resp. O) and $\gamma(i)$ the set formed by the union of $O(i)$ and $M(i)$. Unless specified, we will use these notations and data structures in the next sections.

3.2.1 Filtering

We recall that a 1-tree is a lower bound of the optimal solution of the TSP. Therefore, the general ideas presented above of the filtering algorithms can be applied. Without loss of generality, we define UB an upper bound of TSP(G), T(G) a minimum 1-tree of G such that $w(T(G)) \leq w(UB)$ and the node 1 is the 1-node.

3.2.1.1 Marginal cost

The idea of this filtering algorithm is to remove, with the marginal costs, the edges that cannot belong to any solution of the TSP. We can define a first naive filtering algorithm: for each edge $e \in O$, add e to M, compute the minimum 1-tree $T'(G)$ and check if $w(T'(G)) < w(UB)$. However, this algorithm requires recomputing a 1-tree for each edge of O. With Kruskal's algorithm having a time complexity in $O(m\alpha(m, n))$ such that $\alpha(m, n)$ is the inverse function of Ackermann [Chazelle, 2000]), we obtain a time complexity in $O(m^2\alpha(m, n))$. Another way is to use the marginal costs. For an edge e, we define its marginal cost $\overline{w}(e)$ as the extra cost due to the integration of e in $T(G)$. Thus, Benchimol et al. [Benchimol et al., 2012] defined the following property in order to filter edges:

Property 3.2.1. Given $e \in O$. If $w(T(G)) + \overline{w}(e) > w(UB)$, then e cannot belong to any TSP solution.

In order to compute $\overline{w}(e)$ for each edge $e \in O$, Benchimol et al. [Benchimol et al., 2012] showed the following property for the edges of $O(X - \{1\})$:

Property 3.2.2. Given $e = (i, j) \in O$ such that $e \notin T(G)$, $i \neq j \neq 1$ and $P_{i,j}$ the unique $i - j$ path in $T(G)\backslash\{1\}$. Then, $\overline{w}(e) = w(e) - max\{w(a) \mid a \in P_{i,j}\}$.

Thus, for each edge $e = (i, j) \in O$, we have to find the maximum-cost edge in the path $P_{i,j}$. It can be done with a Depth-First Search (DFS) on $T(G)$ in a time complexity $O(n)$ (DFS time complexity is $O(n + m)$ but $m = n - 1$ in a tree). Thus, for each edge e we can run a DFS in order to compute $\overline{w}(e)$, it leads to a time complexity in $O(nm)$.

Then, Régin introduced a much more complex algorithm based on the Range Minimum Query [Régin, 2008]. It has a time complexity in $O(n + m + nlog(n))$ and is very efficient in practice. It has also defined an incremental version allowing maintaining $\overline{w}(e)$ when edges are deleted or become mandatory. However, the incremental method cannot be used systematically in the TSP framework because of the Lagrangian relaxation which modifies the cost of the edges at each iteration and thus changes the problem.

Finally, in order to filter the neighbors of node 1, Benchimol et al. [Benchimol et al., 2012] have shown the following property:

Property 3.2.3. Given $e \in \gamma(1)$ such that $e \notin T(G)$. Then, $\overline{w}(e) = w(e) - max\{w(a) \mid a \in \gamma(1), a \in T(G)\}$.

In practice, it is sufficient to simply take the maximum cost non-mandatory edge connecting the 1-node to the MST.

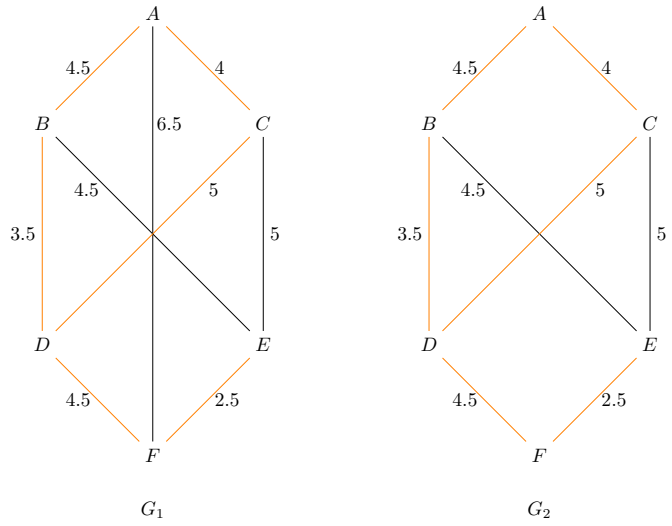


Figure 3.1: An example from Benchimol et al. [Benchimol et al., 2012] of the marginal costs filtering algorithm such that $w(T(G_1)) = 24$, $w(UB) = 25$ and A is the 1-node. G_2 represents the resulting graph from the application of the filtering on G_1 with Property 3.2.1 such that $T(G_1)$ is represented by orange edges.

In G_1 of Figure 3.1 there are 3 edges outside the 1-tree: one around node 1, and two connecting two nodes of the MST. Computing all the marginal costs, we get:

- $\overline{w}((A, F)) = 6.5 - w((A, B)) = 6.5 - 4.5 = 2$
- $\overline{w}((B, E)) = 4.5 - w((D, F)) = 4.5 - 4.5 = 0$
- $\overline{w}((C, E)) = 5 - w((C, D)) = 5 - 5 = 0$

Given $w(T(G_1)) = 24$ and $w(UB) = 25$, Property 3.2.1 implies that (A, F) cannot belong to any solution of the TSP because $24 + 2 > 25$.

In addition, we proved the following property:

Property 3.2.4. Let T be an MST and $I = \{e_1, e_2, ...e_n\}$ a set of non-tree edges of T such that T_I is the MST containing all the edges of I. Then, $w(T_I) \geq \sum_{e \in I} \overline{w}(e) + cost(T)$.

Proof.

See Section B. □

Note that this property could be used in order to define a stronger filtering algorithm based on a combination of edges. For instance, we used this idea in order to generate constraints on edges set. More precisely, given $d = w(UB) - w(LB)$, for each subset O' of edges such that the sum of the marginal costs is greater than d, then integrating O' in LB leads to an inconsistency. Unfortunately, we did not obtain great results with this idea. A similar property has been proved in Benchimol et al. [Benchimol et al., 2012].

3.2.1.2 Replacement cost

The idea of this filtering algorithm is to find the edges that are mandatory for a solution of cost less than or equal to UB. Thus, for each edge $e \in O$, if $w(T(G\backslash\{e\})) > w(UB)$, then e is mandatory (*i.e.* e can be added to M). Thus, a first naive algorithm is the following: for each edge $e \in O$, compute $T(G\backslash\{e\})$ and check if $w(T(G\backslash\{e\})) > w(UB)$. This algorithm has a time complexity in $O(m^2\alpha(m, n))$. It can be improved by noticing that the only way to increase the cost of $T(G)$ is to remove some edges from $T(G)$. Indeed, for each edge $e \notin T(G)$ we have $T(G) = T(G\backslash\{e\})$. Then, we only need to consider the edges in $T(G)$, that is n edges. For each edge e, we then define its replacement cost $w'(e)$ as the extra cost due to the removal of e in $T(G)$. Note that for each $e \in O$ we have $w'(e) \geq 0$ since $T(G)$ is a lower bound of UB. Thus, if $e \in O$ and $e \notin T(G)$, then $w'(e) = 0$. We then define Property 3.2.5 leading to an algorithm with a time complexity in $O(nm * \alpha(m, n))$ because it only requires n calls to Kruskal's algorithm.

Property 3.2.5. Given $e \in T(G)$. If $w(T(G)) + w'(e) > w(UB)$, then e belong to any TSP solution.

Next, Benchimol et al. [Benchimol et al., 2012] have shown the following property:

Property 3.2.6. Given $e \in T(G)\backslash\{1\}$, T^1 and T^2 the two subtrees of $T(G)\backslash\{e\}$. Then $w'(e) = w(e) - min\{w(i, j) \mid (i, j) \in U, (i, j) \neq e, i \in T^1, j \in T^2\}$

This allows us to set up a more efficient algorithm: first, we sort the non-tree edges of $T(G)$ by increasing cost. Then, for each non-tree edge $e = (i, j)$, we mark the path going from i to j in $T(G)$ as visited. For each edge e' on this path that was not yet visited, we can then set $w'(e') = w(e') - w(i, j)$. For each non-tree edge, we perform a DFS in $T(G)$ ($O(n)$) in order to find the unvisited edges in the path going from i to j in $T(G)$ Therefore, this algorithm has a time complexity in $O(nm)$. A last and more efficient approach is to use the algorithm introduced in Régin et al. [Régin et al., 2010]. The main idea is to compress the visited paths for each performed DFS, it allows us to traverse each edge at most once. This yields to a more efficient algorithm with a time complexity in $O(m\alpha(m, n))$. However, we must add the time complexity of sorting

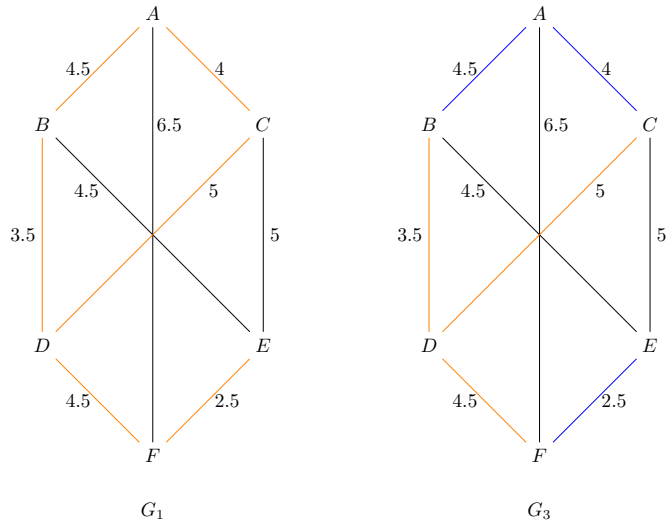


Figure 3.2: An example from Benchimol et al. [Benchimol et al., 2012] of the replacement cost filtering algorithm. G_3 represents the resulting graph from the application of the filtering on G_1 with Property 3.2.5 such that $T(G_1)$ is represented by orange edges and the blue edges in G_2 are the mandatory edges.

the edges to $O(m\alpha(m,n))$. Note that it can be reused from the $T(G)$ computation if Kruskal's algorithm is used.

Finally, Benchimol et al. [Benchimol et al., 2012] defined the following property for the neighbors of the 1-node:

Property 3.2.7. Given $e \in \gamma(1)$ such that $e \in T(G)$. Then, $w'(e) = w(e) - min\{w(a) \mid a \in \gamma(1), a \notin T(G)\}$.

Figure 3.2 represents an application of the replacement cost filtering algorithm. Computing all the replacement costs, we get:

- $w'((A,B)) = w((A,F)) - w((A,B)) = 6.5 - 4.5 = 2$
- $w'((A,C)) = w((A,F)) - w((A,C)) = 6.5 - 4 = 2.5$
- $w'((B,D)) = w((B,E)) - w((B,D)) = 4.5 - 3.5 = 1$
- $w'((C,D)) = w((C,E)) - w((C,D)) = 5 - 5 = 0$
- $w'((D,F)) = w((B,E)) - w((D,F)) = 4.5 - 4.5 = 0$
- $w'((E,F)) = w((B,E)) - w((E,F)) = 4.5 - 2.5 = 2$

Finally, given $w(T(G)) = 24$ and $w(UB) = 25$, Property 3.2.5 implies that (A,B), (A,C) and (E,F) belong to any solution of the TSP. Thus, they are added to M.

3.2.1.3 Degree constraint

The filtering algorithm of the degree constraint Equation 2.2 is as follows.

$$\sum_{e \in \gamma(i)} x_e = 2 \qquad \forall i \in X \qquad \text{(Equation 2.2)}$$

The degree constraint does not hold if $|M(i)| > 2$ or $|\gamma(i)| < 2$. If $|\gamma(i)| = 2$, then the edges of $O(i)$ become mandatory. Finally, if $|M(i)| = 2$, then the edges of $O(i)$ are removed. Initially, we can apply these filtering rules for all the nodes of the graph. Then, we can incrementally check the nodes. Indeed, the cardinality of a node only changes if an edge adjacent to this node is deleted or becomes mandatory. Thus, each time we call this filtering algorithm, we can check only the nodes with a cardinality that has changed since the last call.

3.3 The search strategy

In order to find the solutions of the TSP, the search strategy considers only the edges of the graph variable. A solution is found when the number of mandatory edges is equal to the number of nodes such that all the constraints hold. The search tree is a binary tree such that a left branch is an edge assignment and a right branch is an edge removal. Several search strategies have been designed for the TSP solving. In CP, the most efficient is the search strategy LCFirst of Fages et al. [Fages et al., 2016] which is an interpretation of Last Conflict heuristics [Haralick and Elliott, 1979, Lecoutre et al., 2009] for graph variables. It selects one edge in the graph according to a heuristic and keeps branching on all the edges adjacent to one extremity (the LCNode) of this edge until the LCNode is exhausted. Note that LCFirst keeps the LCNode even if a backtrack occurs. Thus, it is a highly dynamic search strategy that learns from previous choices. Algorithm 2 is a possible implementation of the LCFirst search strategy. In addition, an example is depicted in Figure 3.3. It should be noted that the use of LCFirst makes most of the search strategies much more efficient (up to an order of magnitude).

Algorithm 2: $LCFirst(G)$

```
LCFirst (G = (X, M, O))
    Input: The current graph to solve G.
    Output: An edge (u, v).
    global LCNode;
    if LCNode ≠ nil then
        // Select an optional edge adjacent to LCNode in G
        if O(LCNode) ≠ ∅ then return select(O(LCNode)) ;
    // Select an optional edge in G
    (u, v) ← select(O);
    LCNode ← u;
    return (u, v)
```

In practice, we observed that the use of LCFirst strongly interferes with Lagrangian relaxation and filtering algorithms. Indeed, we have shown that an additional constraint leading to a huge

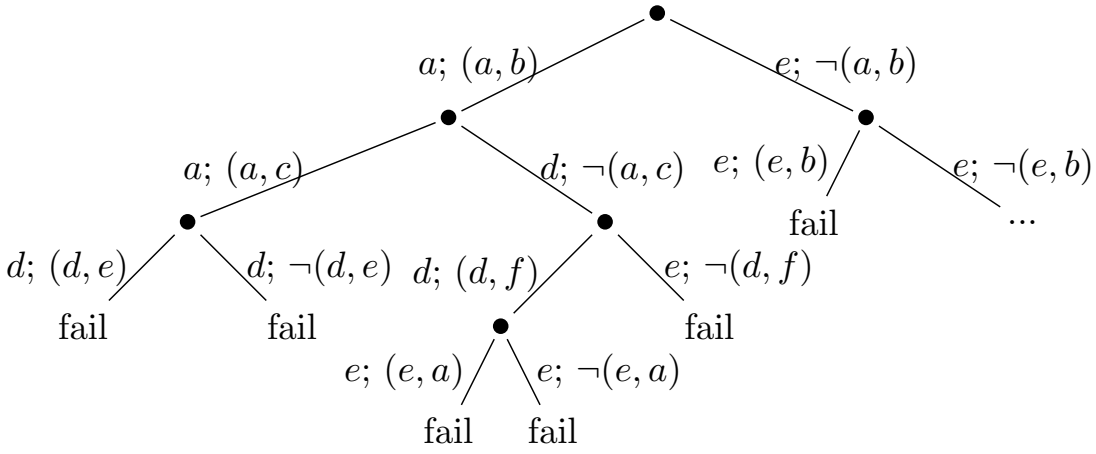


Figure 3.3: An example of a search strategy with LCFirst. We note "LCNode; (u, v)" such that (u, v) is an assigned edge and $\neg(u, v)$ is a removed edge. We observe that when (d, e) and $\neg(d, e)$ have caused a fail, then d is backtracked as the LCNode choice. We also observe that the choice of LCNode=e is backtracked from the fail of (e, a) to the first refutation because no choice could so far exhaust the neighborhood of e.

improvement with a static search strategy (*i.e.* without LCFirst) such as maxCost (selecting the edges by decreasing cost) can lead to a degradation of the results with LCFirst maxCost and that additional constraint [Isoart and Régin, 2019]. Note that LCFirst maxCost is considered as one of the best search strategies by Fages et al. [Fages et al., 2016]. Nevertheless, we also showed that the search strategy LCFirst minDeltaDegree (selecting the edges for which the sum of the endpoint degrees in the upper bound minus the sum of the endpoint degrees in the set-variable lower bound is minimal) combined with WCC and that additional constraint works well. Therefore, unless specified, we will use the search strategy LCFirst minDeltaDegree.

3.4 k-cutset constraint

3.4.1 Introduction

In any TSP solution, each cutset of a graph must contain at least two edges. In addition, any solution contains a positive even number of elements from this cutset. Therefore, we define a structural based constraint, the k-cutset constraint, based on this idea. For each cutset of size k, it checks if it can contain an even and strictly positive number of mandatory edges.

For instance, we show two k-cutsets in the graph of Figure 3.4. The 2-cutset K_1 contains 2 edges and the 4-cutset K_2 contains 4 edges.

First, we introduced a quadratic algorithm for the k-cutset constraint checking the consistency and performing some filtering operations [Isoart and Régin, 2019]. It is based on a 2-edge-connected subgraph and Tsin's algorithm [Tsin, 2009]. However, this algorithm only handles the k-cutset of size lower or equal than 3. Since the number of k-cutsets in a graph is exponential, we

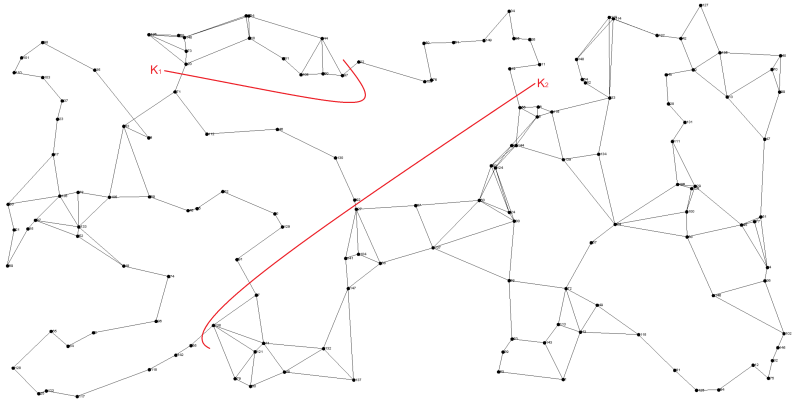


Figure 3.4: Graph kroA150 from TSPLib [Reinelt, 1991] while solving in a CP solver. K_1 is a 2-cutset and K_2 is a 4-cutset.

set this limitation. Note that computing all cutsets is equivalent to compute all possible partitions of the graph nodes, *i.e.* 2^n partitions.

However, in order to deduce something, we are not interested in all of them. The ones we are interested in are the k-cutsets containing k or $k-1$ mandatory edges in the graph. In addition, there are at most n mandatory edges in any TSP solution. Therefore, we introduced an algorithm based on Tarjan's bridges algorithm [Tarjan, 1974] performing the consistency check and the pruning in linear time [Isoart and Régin, 2021b].

In this section, we first introduce the quadratic algorithm with $k \leq 3$. Then, we show that it is sufficient to study a set of k-cutsets lower than or equal to n in order to obtain a complete filtering. Moreover, we introduce a linear time algorithm for the k-cutset constraint for any k.

3.4.2 The k-cutset constraint and its filtering rules

We previously said that the purpose of the k-cutset constraint is to ensure that for each cutset in G that a strictly positive and even number of edges are mandatory in any solution.

More formally, we introduce the following proposition:

Proposition 3.4.1. *Given K a k-cutset. Then, any Hamiltonian cycle C contains an even and strictly positive number of edges from K.*

Proof.

Consider a k-cutset of a graph G and C a Hamiltonian cycle in G. The k-cutset partition G into two sets of vertices X_1 and X_2. Let u be our starting vertex in X_1, by definition C visits all the vertices of G and ends up visiting u (its starting vertex). Thus, visiting the vertices of X_2 involves taking one edge of the k-cutset and taking a different one to come back into X_1, at that moment: either all the vertices of X_2 have been visited and we end up joining u without using other edges of the k-cutset, or we have to visit X_2 again and return to X_1, every time we visit X_2 from X_1 we need an edge to go in, and another to go back: this means an even and strictly positive number of edges and the proposition holds. □

Since G contains mandatory and optional edges, a k-cutset of G can be partitioned into two disjoint subsets of O and M. Therefore, we note a k-cutset $K = (M', O')$ of G such that $M' \subset M$ and $O' \subset O$.

Definition 3.4.1 (k-cutset constraint)**.** For each k-cutset $K = (M', O')$, the k-cutset constraint ensure that $|M'| + |O'| \geq 2$ and $|M'|$ is even if $O' = \emptyset$.

From Definition 3.4.1, we can therefore define the following consistency checks:

Corollary 3.4.2. *If there is a k-cutset in G such that $k < 2$, then there is no solution for TSP(G).*

Corollary 3.4.3. *If there is a k-cutset in G containing only mandatory edges such that k is odd, then there is no solution for TSP(G).*

In addition, we can define from Definition 3.4.1 the following filtering rules:

Corollary 3.4.4. *Given a 2-cutset $K = (M', O')$ in G. Then, the edges of O' must become mandatory ($M \leftarrow M + O'$).*

Corollary 3.4.5. *If there is a k-cutset K in G containing $k - 1$ mandatory edges such that k is odd, then the non-mandatory edge e of K can be safely deleted ($O \leftarrow O - \{e\}$).*

Corollary 3.4.6. *If there is a k-cutset K in G containing $k - 1$ mandatory edges such that k is even, then the non-mandatory edge e of K must become mandatory ($M \leftarrow M + \{e\}$ and $O \leftarrow O - \{e\}$).*

Therefore, Corollary 3.4.2 and Corollary 3.4.3 allow checking the consistency, Corollary 3.4.4 and Corollary 3.4.6 allow finding mandatory edges, Corollary 3.4.5 allows removing edges.

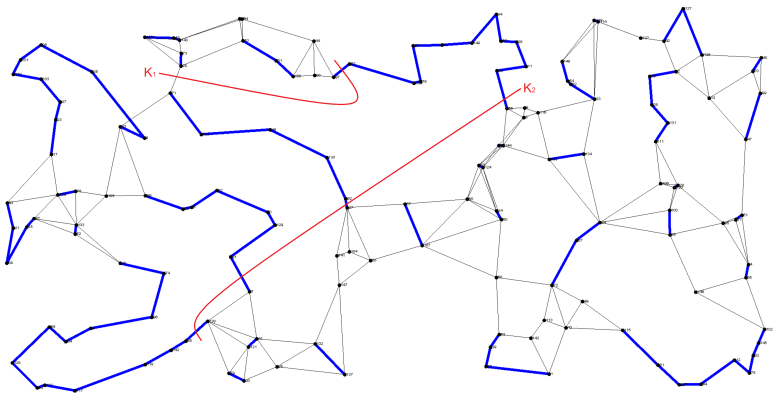


Figure 3.5: Representation of the graph from Figure 3.4 with mandatory edges (blue) and optional edges (dark). The two k-cutsets K_1 and K_2 are displayed in red.

For instance, we can deduce in Figure 3.5 that for the 2-cutset K_1, all the K_1 edges become mandatory by Corollary 3.4.4 or Corollary 3.4.6. Moreover, the 4-cutset K_2 is valid because the k-cutset constraint is consistent (K_2 contains only mandatory edges and $|K_2| = 4$ is even).

Given k, a first approach is to enumerate all the k-cutsets of size lower or equal to k and apply Corollary 3.4.2, Corollary 3.4.3, Corollary 3.4.4, Corollary 3.4.5 and Corollary 3.4.6 to the k-cutsets. However, it does not seem reasonable since the complexity is related to k for enumeration algorithms [Yeh et al., 2010]. Therefore, we introduce in the next section a quadratic algorithm limiting the study to $k \leq 3$.

3.4.3 A non-complete quadratic time algorithm

For the k-cutsets such that $k \leq 3$, we can split the problem in many cases. First, the 1-cutsets are bridges in a graph. A bridge is an edge such that its removal increases the number of connected components. In order to find them, there is Tarjan's bridges algorithm [Tarjan, 1974] running in $O(n + m)$ based on a DFS. Then, the 2-cutsets can be found with Tsin's algorithm [Tsin, 2009] in $O(n + m)$ which is also based on a DFS. The strength of Tsin's algorithm is that it allows us to find bridges and 2-cutsets. Thus, we can manage $k = 1, 2$ at the same time with a linear time algorithm. Now, we must manage $k = 3$. The only 3-cutsets we are interested in are the cutsets with at least two mandatory edges.

By the definition of a cut in a graph, if we remove an edge of a k-cutset K, then K becomes a $(k-1)$-cutset. With this idea, we then can introduce a first simple algorithm: for each mandatory edge $e \in M$, we look for the 2-cutsets of $G - \{e\}$. In this way, each 2-cutset K' found forms a 3-cutset with at least one mandatory edge if K' is not a 2-cutset in G. There are at most n mandatory edges in the graph, therefore this algorithm run in $O(n * (n + m))$.

Next, we will use Definition 3.4.2 in order to name the mandatory edges that we consider for our algorithm.

Definition 3.4.2 (Identification edges)**.** The **identification edges** are the mandatory edges for which a 2-cutset algorithm is run.

Definition 3.4.3 (k-edge-connected graph)**.** A **k-edge-connected graph** is a graph in which there is no edges set of cardinality strictly less than k disconnecting the graph.

In the next section, we introduce a special data structure named a CST. It is a 2-edge-connected subgraph of G. Note that the CST is not a required data structure for the proper functioning of the k-cutset propagator, just an improvement.

3.4.3.1 CST : Cycled Spanning Tree

A CST is a 2-edge-connected subgraph of G. Moreover, for each edge e of G there is a cycle in G formed only by edges of the CST and e. One way to build a CST is to compute a spanning tree T. Then, add some edges to T until all the edges, those of T and those outside of T, belong to a CST cycle. Since T is a spanning tree, any edge $e \notin T$ belongs to a cycle composed of e and only T edges. For the edges of T, the CST is built by adding edges to the spanning tree such that each tree edge belongs to a cycle of the CST. This can be done by marking the tree edges each time a cycle is found. More precisely, we consider three graphs at the same time: G the graph, T the spanning tree, and CST the CST. Initially, it is T and all the tree edges are unmarked. We parse the non-tree edges of G until we find $e_{\overline{T}} = (i, j) \notin T$ such that there is a cycle formed with at least one unmarked edge of T. Then, we add $e_{\overline{T}}$ to CST and we mark all the tree edges of the cycle. We repeat this operation until there is no more unmarked edge in T. Clearly, at the end, each tree edge which has been marked belongs to a cycle. In addition, there is at least one tree

edge in each cycle, so the number of added edges is bound by n. An example of a construction is shown in Fig. Figure 3.6. This algorithm can be implemented in linear time, by using a union-find data structure in order to avoid traversing each edge of each cycle. If we consider first the non-mandatory edges for the construction of the spanning tree and for the construction of the CST, then we can expect to reduce the number of mandatory edges in the CST.

Without loss of generality, we assume that G is a connected bridgeless graph. Thus, there is a CST in G. Note that both Tarjan's bridges algorithm [Tarjan, 1974] and Tsin's algorithm [Tsin, 2009] ensure that the graph is connected and bridgeless.

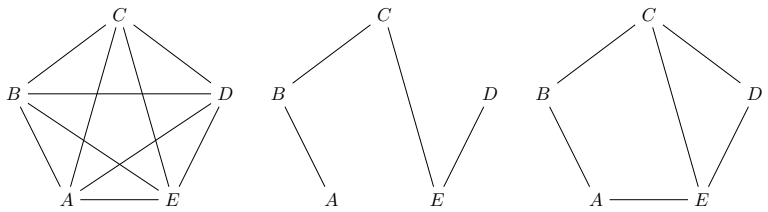


Figure 3.6: Example of building a CST.

Corollary 3.4.7. *Given $k > 1$. If there is a k-cutset in G, then at least two edges of the cutset are in the CST.*

Proof.

By construction, the CST is connected and covers the graph with cycles. So each cutset has a cardinality greater than or equal to two. □

Corollary 3.4.8. *If there is a 3-cutset containing at least two mandatory edges, then at least one mandatory edge belongs to the CST.*

Proof.

Immediate from Corollary Corollary 3.4.7. □

From Corollary Corollary 3.4.8, the simple algorithm can be improved by reducing the number of mandatory edges that are considered. Considering the identification edges as each mandatory edge e of CST, the algorithm becomes: for each identification edges e, search for the 2-cutsets of $G - \{e\}$. For each 3-cutset found, we obtain either a 3-cutset with three mandatory edges, a 3-cutset with two mandatory edges, or a 3-cutset with one mandatory edge. Then, we apply Corollary 3.4.3, Corollary 3.4.5 and Corollary 3.4.6 on these 3-cutsets.

Since mandatory edges outside the CST are not considered as identification edges and the edges in the CST are chosen during construction, it is a good idea to minimize the number of mandatory edges in the CST in order to reduce the number of iterations of the algorithm.

3.4.3.2 Additional improvement

The proposed algorithm is highly dependent on the number of identification edges. From Corollary 3.4.7, if two edges belong to the same 2-cutset and are mandatory, then they are identification edges. However, when searching for the 3-cutsets with an identification edge, it is not necessary to repeat the search for all the edges forming a 2-cutset with it. More precisely, the problem of searching for 3-cutsets with the identification edge e has the same set of solutions as the problem of searching for 3-cutsets with each edge forming a 2-cutset with e. Figure 3.7 illustrates it well since the 2-cutset is a path.

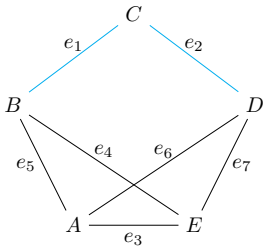


Figure 3.7: $\{e_1,e_2\}$ is a 2-cutset. $\{e_1,e_4,e_5\}$ and $\{e_1,e_6,e_7\}$ are 3-cutsets including e_1. We can deduce that $\{e_2,e_4,e_5\}$ and $\{e_2,e_6,e_7\}$ are 3-cutsets including e_2.

Property 3.4.1. Let S_1 be a k-cutset and S_2 be a 2-cutset such that $k > 1$ and $S_2 \not\subseteq S_1$. If $\exists a \in S_1$ such that $a \in S_2$ then $(S_1 \cup S_2) - \{a\}$ forms a k-cutset.

Proof.

Given $S_2 = \{e_1, e_2\}$ a 2-cutset and $e_1 \in S_1$. Removing S_1 from the graph disconnects it into two connected components. In the modified graph, $S_2 - \{e_1\} = \{e_2\}$ is a bridge. Removing e_2 further increases the number of connected components: there are now three. If we put back e_1, G is disconnected into two connected components, its cutset is $(S_1 - \{e_1\}) \cup \{e_2\} = (S_1 - \{e_1\}) \cup (S_2 - \{e_1\}) = (S_1 \cup S_2) - \{e_1\}$. Since S_1 is a k-cutset, there is no subset of it that disconnects the graph other than the k-cutset itself. If $(S_1 \cup S_2) - \{e_1\}$ disconnects the graph then it is a k-cutset because we delete and add an edge in a set of initial cardinality k. □

Consider S_1 a 3-cutset, $S_2 = \{e_1, e_2\}$ and S_3 two distinct 2-cutsets. From Property 3.4.1 the number of identification edges is reduced:

- If $e_1 \in S_1$, then $(S_1 - \{e_1\}) \cup \{e_2\}$ is a 3-cutset.
- If $e_1 \in S_3$, then $(S_3 - \{e_1\}) \cup \{e_2\}$ is a 2-cutset.

Thus, the set of identification edges is defined by the mandatory edges of the CST that do not belong to any 2-cutset and the subset of edges belonging to all 2-cutsets of G that maximizes its cardinality such that there is no combination of it forming a 2-cutset.

To avoid any inconsistency, all 2-cutsets must be searched before performing the 3-cutset search. Otherwise, there may be a 2-cutset containing at least one non-mandatory edge. This may result in an edge being marked as removable when searching for 3-cutsets while it is necessary

for the existence of a Hamiltonian cycle. In addition, deleting an edge in a 3-cutset may create a 2-cutset and so either we perform a 2-cutset search immediately or we wait until the end of the search of all 3-cutsets to make the deletions effective. The first possibility is too time-consuming, a better solution is to postpone the deletions.

With this method we consider a subset of the identification edges. The higher the mandatory number of edges required, the more likely it is that the number of edges considered will be reduced.

Finally, CST has another advantage: it is incremental. Indeed, as long as no CST edges are removed, all edges outside the CST belong to a cycle composed of CST edges, so there is no need to rebuild it.

3.4.3.3 Implementation

A possible implementation of the k-cutset constraint such that $k \leq 3$ is given in Algorithm 3 and Algorithm 4. The main function is propagKCutset(G). The function propag2Cutset defines a 2-cutset filtering and the function propag3Cutset defines a 3-cutset filtering. Both filtering functions use find2Cutset($G, bridgeFunction, 2 - cutsetFunction$) which finds all the 2-cutsets in G. In addition, bridgeFunction and 2-cutsetFunction are functions describing the wanted behavior when either a bridge or a 2-cutset is found. Note that this function is used as a black box since it uses Tsin's algorithm [Tsin, 2009] running in $O(n + m)$ in order to find the 2-cutsets. Next, the function mergeCutpairs allows the use of the improvement introduced in Section 3.4.3.2. We will now describe the overall behavior of the algorithm. In propagKCutset(G), we define set as a set of pairs of edges forming 2-cutsets in G. Then, we use the filtering algorithm propag2Cutset in order to find and make mandatory all the edges belonging to a 2-cutset in G. Note that the 2-cutsets are stored in set. The id array represents for each edge its 2-cutset identifier. In order to create sets of edges forming 2-cutsets between them Function mergeCutpairs is called. Each disjoint set will finally have a different identifier and each edge belonging to the same set will have the same identifier. Then, we define an array $visited$ to allow us to consider only one edge per set described above. The $identificationEdges$ set contains the mandatory edges which are in the CST. Then, we consider one edge per set calculated by mergeCutpairs and all the edges of the CST not being in any set. For each of its edges, the filtering propag3Cutset is performed, *i.e.* the Corollary 3.4.5, Corollary 3.4.6 and Corollary 3.4.3 are used. As recommended in Section 3.4.3.2, deletions are postponed. The final complexity of Algorithm 3 is $O(k * (n + m))$ where $k <= |M| <= n$: Tsin's algorithm in $O(n + m)$ is called k times.

Algorithm 3: k-cutset($G = (X, M, O)$)

propagKCutset *($G = (X, M, O)$)*
- **Input:** A graph $G = (X, M, O)$
- **Output:** A boolean specifying if the constraint is consistent
- $set \leftarrow \emptyset$; `// set of 2-cutsets`
- **if *not*** *propag2Cutset(G, M, set)* **then return** *False* ;
- $\forall e \in O \cup M : id[e] \leftarrow nil$; `// 2-cutset identifier of each edge`
- $\forall e \in O \cup M : visited[e] \leftarrow False$;
- $G' = (X', M', O') \leftarrow G$;
- $identificationEdges \leftarrow$ CST(G).getMandatoryEdges();
- mergeCutpairs($identificationEdges, set, id$);
- **foreach** $e \in identificationEdges$ **do**
  - **if** *id[e] = nil* ***or*** *$\neg$visited[id[e]]* **then**
    - **if *not*** *propag3Cutset(G, M, O, O', e)* **then return** *False* ;
    - **if** *id[e] $\neq$ nil* **then** $visited[id[e]] \leftarrow True$;
- $G \leftarrow G'$; `// Update of G since deletions are postponed`
- **return** *True*;

mergeCutpairs *(S,set,id)*
- **Input:** The set of identification edges S, the set of 2-cutsets set and the 2-cutset identifier of each edge id
- $cpt \leftarrow 0$;
- **foreach** $(e_1, e_2) \in set$ **do**
  - `// if both` e_1 `and` e_2 `do have an identifier`
  - **if** *id[e_1] $\neq$ nil* ***and*** *id[e_2] $\neq$ nil* **then**
    - `// (`e_1, e_2`) is a 2-cutset: id[`e_1`] must be equals to id[`e_2`] : id are merges`
    - **foreach** $s' \in S$ **do**
      - **if** *id[s'] = id[e_1]* **then**
        - $id[s'] \leftarrow id[e_2]$;
  - `// if both` e_1 `and` e_2 `do not have an identifier`
  - **if** *id[e_1] = nil* ***and*** *id[e_2] = nil* **then**
    - $id[e_1] \leftarrow id[e_2] \leftarrow cpt$;
    - $cpt \leftarrow cpt + 1$;
  - `// if` e_2 `does not have an identifier and` e_1 `have one`
  - **if** *id[e_1] $\neq$ nil* ***and*** *id[e_2] = nil* **then**
    - $id[e_2] \leftarrow id[e_1]$;
  - `// if` e_1 `does not have an identifier and` e_2 `have one`
  - **if** *id[e_1] = nil* ***and*** *id[e_2] $\neq$ nil* **then**
    - $id[e_1] \leftarrow id[e_2]$;

Algorithm 4: k-cutset($G = (X, M, O)$)

propag2Cutset *(G, M, set)*
- **Input:** A graph G, its set of mandatory edges M and the set of 2-cutsets set
- **Output:** A boolean specifying whether the graph is 3-edge-connected
- `// A fail is triggered if a bridge is found and the 2-cutsets become mandatory`
- **return** *find2Cutset(G,fail(),makeCutsetMandatory($M, set, ..., ...$)*;

propag3Cutset *(G, M, O, O', e)*
- **Input:** A graph G, its set of mandatory edges M, its set of optional edges, O, a copy of O to modify O' and an identification edge e
- **Output:** A boolean specifying whether the graph contains a 3-cutset with 3 mandatory edges
- $G'' \leftarrow (X(G), M - \{e\}, O)$;
- `// If a 2-cutset with two mandatory edges are found, then a failure is triggered`
- `// If a 2-cutset with one mandatory edge is found, then the other one is removed`
- **return** *find2Cutset(G'',pass(),consistencyAndPruningOf3Cutset($M, O', set, ..., ...$))*;

pass *()*
- Do nothing;

fail *()*
- Trigger a failure;

makeCutsetMandatory *(M, set, e_1, e_2)*
- **Input:** The set of mandatory edges M, the set of 2-cutsets set, two edges e_1 and e_2
- **if** $e_1 \notin M$ **then** $M \leftarrow M \cup \{e_1\}$;
- **if** $e_2 \notin M$ **then** $M \leftarrow M \cup \{e_2\}$;
- $set \leftarrow set \cup (e_1, e_2)$;

consistencyAndPruningOf3Cutset *($M, O' set, e_1, e_2$)*
- **Input:** The set of mandatory edges M, the set of 2-cutsets set, two edges e_1 and e_2
- **if** $e_1 \in M$ *and* $e_2 \in M$ **then** fail();
- **else if** $e_1 \in M$ **then** $O' \leftarrow O' - \{e_2\}$;
- **else if** $e_2 \in M$ **then** $O' \leftarrow O' - \{e_1\}$;

3.4.4 A linear time algorithm

In order to improve the previous algorithm, we introduce Definition 3.4.4 and Definition 3.4.5.

Definition 3.4.4 (Failing k-cutset)**.** A k-cutset $K = (M', O')$ is failing if $|M'| = k$ and k is odd.

Definition 3.4.5 (Prunable k-cutset)**.** A k-cutset $K = (M', O')$ is prunable if $k > 1$ and $|M'| = k - 1$ and $|O'| = 1$.

The main issue of the previous algorithm is that it cannot check Corollary 3.4.3, Corollary 3.4.5 and Corollary 3.4.6 for any k in linear time. Looking closely, we notice that there is no need to enumerate all the k-cutsets. We only need to find the k-cutsets with k or $k-1$ mandatory edges. Therefore, instead of searching the k-cutsets and applying the corollaries to each of them, we will directly search for the k-cutsets that are in the scope of the corollaries, *i.e.* the failing and the prunable k-cutsets.

Therefore, the consistency check will search for failing k-cutsets and the filtering algorithm will search for both the prunable k-cutsets and 2-cutsets.

Definition 3.4.6 (Mandatory path)**.** A **mandatory path** of G is a path $p = [x_1, ..., x_k]$ in G such that for each $i \in [1, k-1]$, the edge (x_i, x_{i+1}) is mandatory.

Definition 3.4.7 (Path-merged graph)**.** A **path-merged graph** G_{p-m} of G is the graph G such that for each mandatory path $p = [x_1, ..., x_k]$ of G and $k > 2$, the nodes from x_2 to x_{k-1} are removed and a mandatory edge (x_1, x_k) is added.

Definition 3.4.8 (Merge of nodes)**.** Given X' a set of nodes. The **merge** of X' is a mapping from all nodes of X' to a single node.

Definition 3.4.9 (Merged graph)**.** Given $G_{p-m} = (X_{p-m}, M_{p-m}, O)$ a path-merged graph. A **merged graph** G_m of G_{p-m} is the multigraph G_{p-m} such that each connected component of the subgraph $G_{opt} = (X_{p-m}, \emptyset, O)$ of G_{p-m} are merged.

Definition 3.4.10 (Optional graph)**.** Given $G_{p-m} = (X_{p-m}, M_{p-m}, O)$ a path-merged graph. We note $G_{opt} = (X_{p-m}, \emptyset, O)$ the graph of optional edges.

Definition 3.4.11 (2-merged graph)**.** Given $G_{p-m} = (X_{p-m}, M_{p-m}, O)$ a path-merged graph. A **2-merged graph** $G_2 - m$ of G_{p-m} is the multigraph G_{p-m} such that each 2-edge-connected component of the subgraph $G_{opt} = (X_{p-m}, \emptyset, O)$ of G_{p-m} are merged.

For instance, Figure 3.8 shows an example of Definition 3.4.7, Definition 3.4.9 and Definition 3.4.11.

Without loss of generality, we will consider that G is connected. In addition, we will use $G_{p-m} = (X_{p-m}, M_{p-m}, O)$ the path merged graph of G, $G_m = (X_m, M_m, \emptyset)$ the merged graph of G_m and $G_{opt} = (X_{p-m}, \emptyset, O)$ the subgraph G_{p-m} containing only optional edges. Each removed node of G in G_{p-m} is connected with exactly two mandatory edges (thanks to the degree constraint of the WCC) and each path is replaced by a single mandatory edge. Therefore, we will often consider G_{p-m} instead of G.

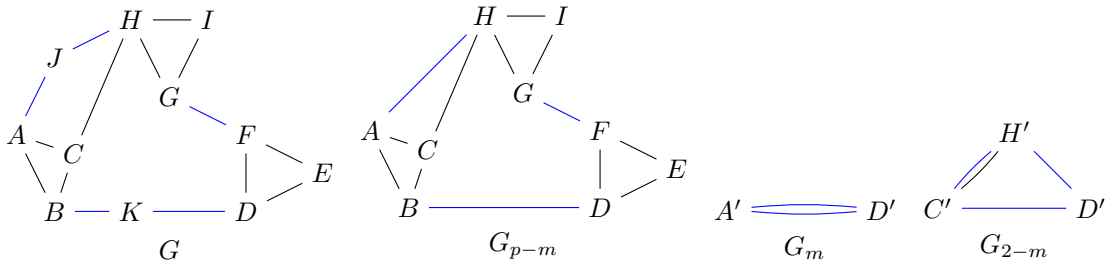


Figure 3.8: Given G a 2-edge-connected graph. G_{p-m} is the path-merged graph of G such that the mandatory paths are $p_1 = [A, J, H]$ and $p_2 = [B, K, D]$. G_m is the merged graph of G_{p-m}. G_{2-m} is the 2-merged graph of G_{p-m}.

3.4.4.1 Consistency Check

In order to determine whether G is consistent with the k-cutset constraint, we must check Corollary 3.4.2 and Corollary 3.4.3. We can check Corollary 3.4.2 with Tarjan's bridges algorithm [Tarjan, 1974]. Using a DFS, it finds all the bridges of a graph (*i.e.* the 1-cutsets) in $O(n + m)$. Checking Corollary 3.4.3 requires to check for the existence of a failing k-cutset. We will show that it can be done in a linear time with Proposition 3.4.9.

Definition 3.4.12 (Outgoing mandatory edges of a nodes set)**.** Given X' a set of nodes. We note $M^+(X') = \{(i, j) \mid (i, j) \in M, i \in X', j \notin X'\}$ the set of outgoing mandatory edges of X' in M.

Proposition 3.4.9. *There is no failing k-cutset in G_{p-m} if and only if there is no connected component X' in G_{opt} such that $|M^+(X')|$ is odd.*

Proof.

We note (i) there is no failing k-cutset in G_{p-m} and (ii) there is no connected component X' in G_{opt} such that $|M^+(X')|$ is odd.

$(i) \Rightarrow (ii)$ By definition, if there is no failing k-cutset in G, then there is no connected component X' in G_{opt} such that $|M^+(X')|$ is odd.

$(i) \Leftarrow (ii)$ If a non-empty k-cutset K cuts a connected component of G_{opt}, then K contains optional edges since G_{opt} is the graph of optional edges. Therefore, a k-cutset containing only mandatory edges cannot cut a connected component of G_{opt}. Then, the failing k-cutsets are obtained by partitioning the connected components of G_{opt} in G_{p-m}, *i.e.* all the k-cutsets of the merged graph.

For each $S \subset X_m$, the cutset size of the cut $(S, X_m - S)$ can be computed as follows: (1) make the sum of the number of adjacent edges of each node of S, then (2) subtract twice the number of edges (i, j) such that i and j belong to S (an edge connecting two nodes of S is counted twice in (1)). Since we consider that (ii) is true, the sum obtained by (1) is even. (2) subtract an even number to the sum obtained by (1). Therefore, the cutset size is even and there is no failing k-cutset in G_{p-m}. □

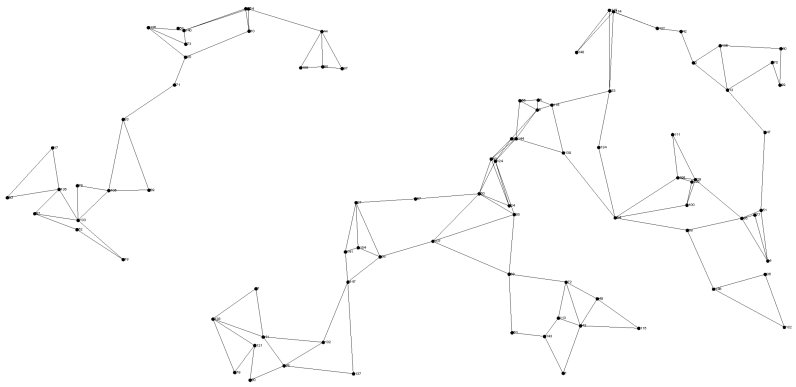


Figure 3.9: Representation of G_{opt} such that G is the graph of Figure 3.5.

For instance, in Figure 3.9, we notice that there are two connected components connected by 4 mandatory paths in Figure 3.5, so the k-cutset constraint is consistent with the graph of Figure 3.5. In addition, if we consider G_m of Figure 3.8, there is no failing k-cutset and each node has an even number of adjacent edges.

Then, we can describe an algorithm. First, compute the connected components of G_{opt}. Then, for each mandatory edge of M having its two endpoints in two different connected components of G_{opt}, we increase the number of mandatory outgoing edges for these connected components. Finally, we iterate on the connected components. If there is a connected component with an odd number of mandatory outgoing edges, then there is a failing k-cutset in G. The computation of the connected components of G_{opt} can be done in $O(n+m)$ with a DFS. The iteration over the mandatory edges of M can be done in $O(n)$. The check of the number mandatory outgoing edges for the connected components can be done in $O(n)$. Thus, we can test the consistency of the k-cutset constraint in $O(n+m)$.

3.4.4.2 Pruning

Corollary 3.4.4, Corollary 3.4.5 and Corollary 3.4.6 define filtering rules for the k-cutset constraint. First, Corollary 3.4.4 can be enforced with Tsin's algorithm [Tsin, 2009]. It performs a single DFS in order to find all the 2-cutsets in a given graph, so it has a time complexity in $O(n+m)$.

In order to enforce Corollary 3.4.5 and Corollary 3.4.6, we need a method finding all k-cutsets having exactly $k-1$ mandatory edges, *i.e.* the prunable k-cutsets. Given a prunable k-cutset $K=(M',O')$ of G. If M' is removed, then the edge of O' is a bridge of G since $|O'|=1$. Formally, we define it in Proposition 3.4.10. We will exploit those bridges in order to enforce Corollary 3.4.5 and Corollary 3.4.6.

Proposition 3.4.10. *If $K=(M',O')$ is a prunable k-cutset of G, then the edge of O' is a bridge in $G'=(X,M-M',O)$.*

Proof.

A prunable k-cutset $K = (M', O')$ contains exactly $k-1$ mandatory edges and 1 optional edge. Thus, removing the $k-1$ mandatory edges in G transform K in a 1-cutset, *i.e.* in a bridge. □

Corollary 3.4.11. *If $K = (M', O')$ is a prunable k-cutset of G, then the edge of O' is a bridge in G_{opt}.*

Proof.

For each prunable k-cutset $K = (M', O')$, $M' \subseteq M$. Thus, from Proposition 3.4.10, the edge of O' is a bridge in $G' = (X, \emptyset, O)$. Therefore, it is a bridge in $G_{opt} = (X_{p-m}, \emptyset, O)$. □

From Corollary 3.4.11, we find the edges belonging to some prunable k-cutsets in G by searching for bridges in G_{opt}. It can be done with Tarjan's bridges algorithm [Tarjan, 1974] in $O(n+m)$. Next, we must determine for each bridge whether it should be deleted or become mandatory. We therefore need to retrieve the set of prunable k-cutsets that contain each bridge.

Without loss of generality, we will consider that G is 2-edge-connected, *i.e.* G is connected and bridgeless. Note that it can be checked in $O(n+m)$ with Tarjan's bridges algorithm [Tarjan, 1974]. In addition, we note $X_i(G')$ the connected component of G' containing the node i.

Proposition 3.4.12. *Given $e \in O$ a bridge in the connected component X' of G_{opt} connecting $(X_1, X' - X_1)$. Then, the k-cutset $K = (M^+(X_1), \{e\})$ is a prunable k-cutset of G.*

Proof.

In order to disconnect X_1 in G, the edges having exactly one end in X_1 must be removed, *i.e.* the k-cutset $K = (M', O')$ of $(X_1, X - X_1)$. It means $M' = M^+(X_1)$ and $O' = O^+(X_1)$. Since $e \in O$ is the bridge in X' of G_{opt} connecting $(X_1, X' - X_1)$, $O' = \{e\}$. Otherwise, e is not a bridge in G_{opt}. Finally, G is 2-edge connected, then $|M'| > 0$. Thus, K is a prunable k-cutset of G. □

Proposition 3.4.13. *Given $e \in O$ a bridge in G_{opt} and a prunable k-cutset $K' = (M', \{e\})$. If there are no failing k-cutsets in G, then there are no prunable k-cutsets K'' such that the pruning of e with K' and K'' is different.*

Proof.

For any k-cutset $K'' = (M'', \{e\})$ such that $M' \neq M''$, we can build a k-cutset $K''' = (M' \cup M'', \emptyset)$ containing only mandatory edges. In addition, if K''' is not a cutset, then e is not a bridge for M' or M''. If K''' is not a failing k-cutset, then $M' \cup M''$ is even. Therefore, M' and M'' are either even or odd. Thus, if there are no failing k-cutsets in G, then there are no prunable k-cutsets K'' such that the pruning of e with K' and K'' is different. □

From Proposition 3.4.12 and Proposition 3.4.13, we can describe a first algorithm: for each bridge of the connected component X' of G_{opt} connecting $(X_1, X' - X_1)$, count the number of mandatory edges having one end in X_1 and the other in $X - X_1$. If there is an even number of mandatory edges, then delete e. Otherwise, add e to the mandatory edges. Thus, for each bridge, we parse at most all the mandatory edges. There are at most $n-1$ bridge in a graph and at most n mandatory edges. Therefore, this algorithm finds all the prunable k-cutsets in $O(n^2)$. Next, we

show how to improve this algorithm in order to obtain a linear time complexity. However, note that this algorithm is already much better than the one of Section 3.4.3 because we find the k-cutsets for all k with a better time complexity.

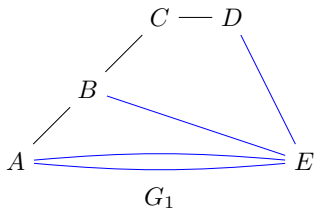


Figure 3.10: G_1 represents the 2-merged graph of the path-merged graph of Figure 3.5. Blue edges are mandatory paths and dark edges are bridges.

In Figure 3.10, the 2-merged graph of Figure 3.5, we notice that some prunable k-cutsets are much simpler than others to find. Indeed, for (A, B) there are two prunable k-cutsets: $K_1 = (M_1, \{(A, B)\})$ where $M_1 = \{(A, E), (A, E)\}$ and $K_2 = (M_2, \{(A, B)\})$ where $M_2 = \{(B, E), (D, E)\}$. In order to find M_1, we can simply search for the mandatory edges having one end in A and the other in $\{B, C, D, E\}$. In order to find M_2, we have to search for the mandatory edges having one end in $\{B, C, D\}$ and the other in $\{A, E\}$. The difference between M_1 and M_2 is that we can simply find M_1 by considering the mandatory edges with exactly one end in a single component. Thus, if for each bridge there is such a prunable k-cutset, then we simply must count the number of outgoing mandatory edges of each 2-edge-connected components of G_{opt} in G. It leads to an algorithm with a linear time complexity. Unfortunately, this algorithm may not handle all the prunable k-cutsets. For instance, there are two prunable k-cutsets containing (B, C): $K_3 = (M_3, \{(B, C)\})$ such that $M_3 = \{(D, E)\}$ and $K_4 = (M_4, \{(B, C)\})$ such that $M_4 = \{(B, E), (A, E), (A, E)\}$. In that case, we cannot simply look at the neighbors of B or C to find the prunable k-cutsets containing (B, C). Indeed, B and C have more than one optional neighbor. It means that the bridge (B, C) disconnect G_{opt} in $(\{A, B\}, \{C, D\})$. Therefore, both connected components are not 2-edge-connected. Thus, we show in Corollary 3.4.14 that for a bridge e connecting (X_1, X_2) in G_{opt}, if X_1 (resp. X_2) is 2-edge-connected, then there exists a prunable k-cutset containing e and all the mandatory edges having exactly one end in X_1 (resp. X_2).

Corollary 3.4.14. *Given X' a connected component of G_{opt}. If $e \in O$ is a bridge in X' of G_{opt} connecting $(X_1, X' - X_1)$ such that X_1 is 2-edge-connected, then there is a prunable k-cutset $K = (M^+(X_1), \{e\})$.*

Proof.

Immediate from Proposition 3.4.12. □

The advantage of Corollary 3.4.14 over Proposition 3.4.12 is that the 2-edge-connected component X_1 of Corollary 3.4.14 are disjoint nodes sets. Thus, parsing the neighbors of the 2-edge-connected components consider at most twice the total number of mandatory edges whereas Proposition 3.4.12 can reconsider for each component all the mandatory edges of the 2-merged graph. We will use this idea in order to obtain a linear time algorithm.

In Figure 3.10, all the k-cutsets formed by the neighborhood of a component are: $K_A = (\{(A,E),(A,E)\},\{(A,B)\})$, $K_B = (\{(B,E)\},\{(B,A),(B,C)\})$, $K_C = (\emptyset,\{(C,B),(C,D)\})$ and $K_D = (\{(D,E)\},\{(D,C)\})$. Among them, K_A and K_D are prunable k-cutsets, then we can immediately deduce for K_1 that (A,B) is deleted (by Corollary 3.4.5) and for K_D that (D,C) becomes mandatory (by Corollary 3.4.4 or Corollary 3.4.6). If we update the k-cutsets we have: $K_A = (\{(A,E),(A,E)\},\emptyset)$, $K_B = (\{(B,E)\},\{(B,C)\})$, $K_C = (\{(C,D)\},\{(C,B)\})$ and $K_D = (\{(D,E),(D,C)\},\emptyset)$. We notice that both K_B and K_C become prunable k-cutsets. Thus, (C,B) becomes mandatory for both K_B and K_C (by Corollary 3.4.4 or Corollary 3.4.6). Finally, all bridges of G_1 have been solved.

Note that the subgraph of optional edges of the 2-merged graph can be more sophisticated than a simple path of bridges edges: it can be a tree. However, it cannot exist a cycle in this subgraph since the 2-edges-connected components are merged in the 2-merged graph. For example, (1) of Figure 3.11 shows a 2-merged graph such that the subgraph of optional edges is not a single path. We note that (1) is rooted in A and each node of the set of nodes $S = \{B,F,G,H,I,J\}$ has exactly one optional neighbor. Thus, we can start by applying Corollary 3.4.14 on S, *i.e.* the leaves of the tree. Leaves are always valid candidates for Corollary 3.4.14 because they have no optional child and a single optional parent. Then, either there are no more leaves and therefore there are no more prunable k-cutsets or there are leaves and we can apply Corollary 3.4.14 to these leaves. Finally, we suggest to recursively apply this process until there are no more leaves in the tree. A sketch of the algorithm is:

- Find bridges in G_{opt} with Tarjan's bridges algorithm.
- Mark the 2-edge-connected components in postorder, *i.e.* the order of a node is set when it is backtracked in the DFS.
- For each mandatory edge (i,j), increase the number of outgoing mandatory edges of the 2-edge-connected component of i and j.
- Iterate over the 2-edge-connected components C_i of the 2-merged graph with the defined postorder and prune the bridge connected to C_i.

For instance, Figure 3.11 shows an execution of the algorithm in a 2-merged graph. Tarjan's bridges algorithm allows us to create the 2-merged graph where black edges are bridges and blue edges are mandatory edges. In this tree, the postorder traversal is $\{B,F,G,D,H,I,J,E,C,A\}$. Note that the postorder is used to guarantee that for each node considered in the execution of the algorithm, all its children have already been pruned. From (1) to (10), we show the iteration over the 2-edge-connected components C_i marked as red nodes. Finally, the bridges are found in $O(n+m)$, parsing the mandatory edges is performed in $O(n)$ and parsing the 2-edge-connected components is performed in $O(n)$. Thus, our algorithm finds the prunable k-cutsets for all k in $O(n+m)$.

We show a possible implementation in Algorithm 5. It takes as input $G = (X,M,O)$. We note CC the set of connected components in G_{opt} and $2CC$ the set of 2-edge-connected components in G_{opt} such that $2CC_i$ is the 2-edge-connected component containing the node i. First, we start by searching the bridges with the DFS-based Tarjan's bridges algorithm in order to build CC and $2CC$. Within the DFS, $2CC$ is constructed with respect to the postorder tree traversal. Thus, iterating on $2CC$, we obtain the postorder tree traversal of the 2-merged graph of G. In Tarjan's bridges algorithm, a 2-edge connected component C is found when a bridge e is found. Thus,

Algorithm 5: Perform the consistency check and the pruning of k-cutset constraint

```
k-cutset (G = (X, M, O))
    Input: A graph G=(X,M, O).
    Output: A boolean specifying whether G contains a failing k-cutset
    // CC : connected components of G − M (G_opt)
    // 2CC : postorder 2-edge-connected components of G − M
        (G_opt)
    computeBridgesDFS(G − M, CC, 2CC) ;
    foreach connected components C ∈ CC do
        if |C| > 1 then
            foreach node i ∈ C do
                if |M(i)| = 1 then
                    (i, j) ← M(i).firstEdge();
                    if not C.isIn(j) then
                        M+(C) ← M+(C) + 1;
                    if not 2CC_i.isIn(j) then
                        M+(2CC_i) ← M+(2CC_i) + 1;
    // Consistency check
    foreach connected components C ∈ CC do
        if M+(C) is odd then return False ;
    // Pruning
    foreach 2-edge-connected components C ∈ 2CC do
        if C.bridge ≠ nil then
            (i, j) ← C.bridge;
            if M+(C) is odd then
                M+(2CC_i) ← M+(2CC_i) + 1;
                M+(2CC_j) ← M+(2CC_j) + 1;
                M ← M + (i, j);
            O ← O − (i, j);
    return True;
```

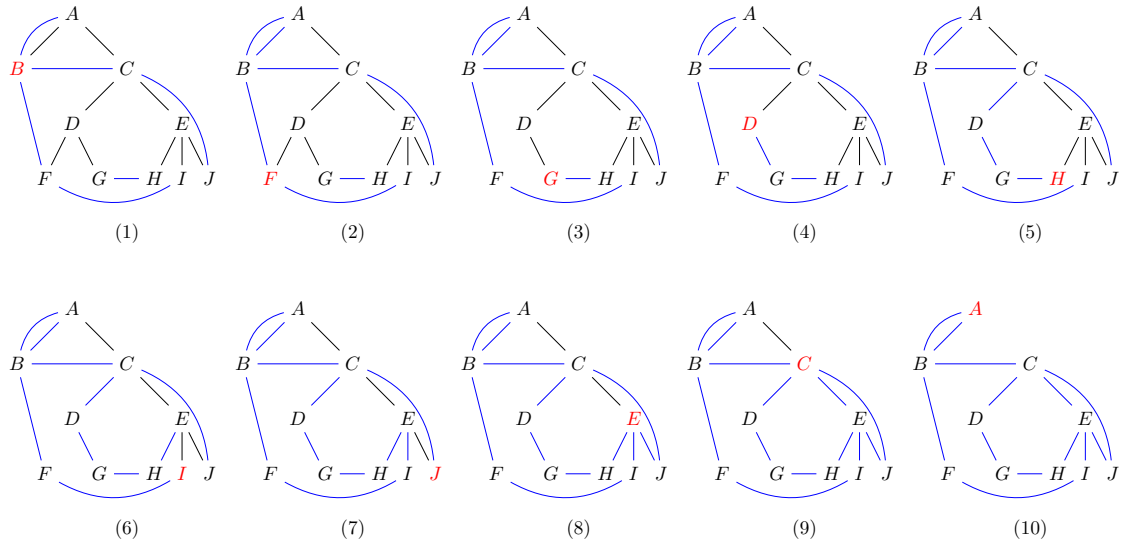


Figure 3.11: Example of an execution of the k-cutset pruning algorithm on the graph (1). Blue edges are mandatory paths, dark edges are bridges. The red node is the current 2-edge-connected component of the algorithm step.

we associate e with C by setting C.bridge=e. For instance, in (1) of Figure 3.11, each node is a 2-edge-connected component knowing its parent, *i.e.* a bridge. Thus, the only node having no parent is the root node and all the other nodes have a single parent that is an optional edge.

Secondly, we count the number of outgoing mandatory edges for each connected component of CC and for each 2-edge-connected component of $2CC$. We then consider the connected components $C \in CC$ such that $|C| > 1$. If $|C| = 1$, then C has two adjacent mandatory edges and C contains a single node. Thus, considering C such that $|C| > 1$ is equivalent of considering the path-merged graph of G. In addition, we know that each node i of C has at most one adjacent mandatory edge (i, j). Therefore, $|M(i)| \leq 1$. Otherwise, the node i would not belong to C. We note $M(i).firstEdge()$ the first edge in the list of the adjacent mandatory edges of i. Since we are looking for the outgoing mandatory edges, we only consider the nodes with $M(i) = 1$. If j is an outgoing edge of C, then we increase the number of outgoing mandatory edges of C noted $M^+(C)$. If i and j do not belong to the same 2-edge-connected component, then we increase the number of outgoing mandatory edges of $2CC_i$ noted $M^+(2CC_i)$ ($M^+(2CC_j)$ is increased when the node j is considered by the foreach). Note that we can check if the node i belongs to the nodes set C' with the function $C'.isIn(i)$. In (1) of Figure 3.11, there is a single connected component C so $M^+(C) = 0$. However, there are 10 2-edge-connected components ($\{A, B, C, D, E, F, G, H, I, J\}$). For example, $M^+(2CC_A) = 1$ and $M^+(2CC_B) = 3$. Afterwards, we check the consistency. If there is $C \in CC$ such that $M^+(C)$ is odd, then there is a failing k-cutset and we return False. In (1) of Figure 3.11, there is no $C \in CC$ such that $M^+(C)$ is odd, so (1) is consistent with the k-cutset constraint.

Thirdly, we perform the pruning step. We iterate on all the 2-edge connected components C of $2CC$. We note (i, j) the bridge associated to C. If $M^+(C)$ is odd, then (i, j) becomes mandatory (*i.e.* it is added to M) and $M^+(2CC_i)$ and $M^+(2CC_j)$ are increased by 1. Whether

$M^+(C)$ is even or odd, (i, j) is removed from O. Indeed, if the edge becomes mandatory, then it must not be in the set of the optional edges. For instance, in (1) of Figure 3.11, we consider the node B and the bridge (B, A). Note that in (1), $M^+(2CC_A) = 1$ and $M^+(2CC_B) = 3$. Since $M^+(2CC_B)$ is odd, (B, A) becomes mandatory and $M^+(2CC_A) = 2$ and $M^+(2CC_B) = 4$. Next, we consider the node F and its bridge (F, D). $M^+(2CC_F) = 2$ so (F, D) is removed and $M^+(2CC_F)$ remains unchanged. Then, we repeat this process until (10). We note that the nodes are chosen in the postorder.

3.4.5 Experiments

In this section we will experimentally show that using the k-cutset constraint reduces the search space and therefore it improves the solving times. To do so, we use the configuration introduced in Section 1.1. Specifically, we always use WCC. We will compare WCC combined with the k-cutset constraint with different search strategies in order to discuss the improvements. In addition, we will study the size of the founded k-cutsets.

Table 3.1 shows the solving times and the number of search nodes for the WCC (1), the WCC and the non-complete quadratic algorithm for the k-cutsets of Section 3.4.3 (2) and the WCC and the complete linear algorithm for the k-cutsets of Section 3.4.4 (3). A ratio column display both solving times and backtrack numbers gain for the k-cutset constraint over the WCC only. We use the static search strategy max cost, selecting edges by decreasing costs. Note that a static search strategy allows us to compare the performance of the filtering without any disruption due to the search strategy.

For the 40 considered instances, we observe that (1) solve 29 instances, (2) solve 32 instances and (3) solve 33 instances. Therefore, the use of the k-cutset constraint allows solving more problems. For instance, gr229 remains unsolved in 86,400s with (1) whereas it is solved in 2,291s with (2) and 1,506s with (3). For most instances, the use of the k-cutset constraint allows improving both the solving times and the number of search nodes. For a280, using (2) instead of (1) leads to a gain of a factor 10.5 in solving time and 33.7 in search nodes. Note that (3) allows obtaining a factor 56.4 in solving time and 170.7 in search nodes. In addition, the geometric mean of (1) is at least 175.4s, (2) is at least 74.2 and (3) is at least 53.5. Moreover, we observe that both (2) and (3) allow reducing the number of search nodes. Therefore, it is clear that the use of the k-cutset constraint is interesting in order to improve the overall solving times with this search strategy.

Comparing (2) and (3), we observe that the results are quite close. Indeed, if we do not consider the instances that have reached the timeout, (2) has a mean solving time of 820s and (3) has a mean solving time of 548s. That is an average gain in solving time of 33% and in number of search nodes of 18%. One would have expected that finding the k-cutsets for any k would lead to much more filtering, but it is not the case. Therefore, it suggests that most of the cutsets are in fact k-cutsets with $k \leq 3$, and therefore (2) already finds most of the cutsets. However, (3) finds it faster since it is on average more improved in solving time than in search space reduction.

Nevertheless, the WCC includes a Lagrangian relaxation and the relation between the filtering algorithms and the Lagrangian relaxation is not clear [Sellmann, 2004, Isoart and Régin, 2020a].

Next, we show the results of the integration of the k-cutset constraint with the state-of-the-art search strategy LCFirst. In the paper of Fages et al. [Fages et al., 2016], there are two dominant search strategy: LCFirst maxCost and LCFirst minDeltaDeg. Then, we show the results of the integration of the k-cutset constraint with these search strategy in Table 3.2 and Table 3.3. In Table 3.2, we observe that a lot of instances are not solved faster using the k-cutset constraint.

Instances	WCC (1) time(s)	WCC (1) #sn	WCC + non-complete quadratic k-cutset (2) time(s)	WCC + non-complete quadratic k-cutset (2) #sn	ratio (1)/(2) time	ratio (1)/(2) #sn	WCC + complete linear k-cutset (3) time(s)	WCC + complete linear k-cutset (3) #sn	ratio (1)/(3) time(s)	ratio (1)/(3) #sn
a280	485.2	687,129	46.0	20,381	10.5	33.7	8.6	4,025	56.4	170.7
ali535	*t.o.*	*t.o.*	*t.o.*	*t.o.*	-	-	*t.o.*	*t.o.*	-	-
bier127	0.3	361	0.3	133	1.0	2.7	0.2	71	1.5	5.1
brg180	51.4	573,417	92.6	522,873	0.6	1.1	62.3	519,033	0.8	1.1
ch130	5.8	11,851	3.3	2,555	1.8	4.6	2.2	2,071	2.6	5.7
ch150	3.9	5,901	2.4	1,817	1.6	3.2	2.0	1,669	2.0	3.5
d198	128.4	175,719	66.8	52,619	1.9	3.3	45.0	41,433	2.9	4.2
d493	*t.o.*	*t.o.*	*t.o.*	*t.o.*	-	-	*t.o.*	*t.o.*	-	-
eil101	0.2	189	0.3	159	0.7	1.2	0.1	66	2.0	2.9
gil262	*t.o.*	*t.o.*	*t.o.*	*t.o.*	-	-	68,772.4	36,507,158	$>$ 1.3	-
gr120	2.7	4,669	1.6	1,579	1.7	3.0	1.1	1,311	2.5	3.6
gr137	8.1	12,309	7.8	7,351	1.0	1.7	5.8	6,441	1.4	1.9
gr202	5.6	9,309	3.0	2,151	1.9	4.3	2.4	1,933	2.3	4.8
gr229	*t.o.*	*t.o.*	2,291.1	1,955,539	$>$ 37.7	-	1,505.7	1,607,301	$>$ 57.4	-
gr431	*t.o.*	*t.o.*	*t.o.*	*t.o.*	-	-	*t.o.*	*t.o.*	-	-
gr666	*t.o.*	*t.o.*	*t.o.*	*t.o.*	-	-	*t.o.*	*t.o.*	-	-
gr96	3.5	14,847	1.2	1,599	2.9	9.3	1.0	1,225	3.5	12.1
kroA100	24.4	99,523	6.1	11,165	4.0	8.9	4.9	8,585	5.0	11.6
kroA150	317.3	616,773	53.8	55,941	5.9	11.0	39.1	49,017	8.1	12.6
kroA200	*t.o.*	*t.o.*	3,086.3	1,944,313	$>$ 28.0	-	1,641.9	1,209,137	$>$ 52.6	-
kroB100	54.1	281,123	4.6	7,641	11.8	36.8	3.8	7,057	14.2	39.8
kroB150	4,923.0	9,356,117	384.2	344,441	12.8	27.2	286.8	299,729	17.2	31.2
kroB200	1,873.4	2,972,339	482.2	333,441	3.9	8.9	358.5	303,319	5.2	9.8
kroC100	1.8	3,887	1.5	1,627	1.2	2.4	1.3	1,465	1.4	2.7
kroD100	0.4	485	0.4	229	1.0	2.1	0.3	211	1.3	2.3
kroE100	298.9	1,532,813	7.8	11,329	38.3	135.3	3.9	6,137	76.6	249.8
lin318	217.4	232,321	64.8	21,293	3.4	10.9	13.6	5,935	16.0	39.1
pcb442	*t.o.*	*t.o.*	*t.o.*	*t.o.*	-	-	*t.o.*	*t.o.*	-	-
pr124	0.6	499	0.8	317	0.8	1.6	0.6	295	1.0	1.7
pr136	852.2	2,635,755	86.5	86,861	9.9	30.3	69.7	86,095	12.2	30.6
pr144	0.9	503	0.8	417	1.1	1.2	0.8	393	1.1	1.3
pr264	6.1	2,119	6.1	903	1.0	2.3	6.0	749	1.0	2.8
pr299	*t.o.*	*t.o.*	*t.o.*	*t.o.*	-	-	*t.o.*	*t.o.*	-	-
rat195	630.4	806,785	448.1	372,439	1.4	2.2	343.3	358,937	1.8	2.2
rat99	0.1	51	0.1	43	1.0	1.2	0.1	41	1.0	1.2
rd400	*t.o.*	*t.o.*	*t.o.*	*t.o.*	-	-	*t.o.*	*t.o.*	-	-
si175	79,753.5	252,129,109	4,175.9	4,661,507	19.1	54.1	3,105.8	4,305,209	25.7	58.6
st70	0.2	379	0.2	105	1.0	3.6	0.1	75	2.0	5.1
tsp225	*t.o.*	*t.o.*	14,928.6	11,579,075	$>$ 5.8	-	10,010.5	9,209,293	$>$ 8.6	-
u159	0.3	243	0.2	39	1.5	6.2	0.2	41	1.5	5.9
mean	$>$ 26,001.3		$>$ 17,936.4				$>$ 17,277.5			
geo mean	$>$ 175.4		$>$ 74.2				$>$ 53.5			

Table 3.1: Comparison of a static strategy (maxCost) with the integration of the k-cutset constraint.

For instance, gr431 is solved in 3,572s without the k-cutset constraint whereas it is solved in 7,944s with the k-cutset constraint. Here, we obtain a case that we frequently encountered while experimenting in this thesis, leading to giving up a lot of ideas: more pruning does not mean a reduction of the search space in the case of the TSP. Indeed, the Lagrangian relaxation leads to loss the monotonicity so dear to CP. In addition, the "learning" strategy such as LCFirst has a complex behavior in the search tree since the next branching choice is influenced by all the previous branching choice, who are themselves influenced by the filtering algorithms.

Then, we study the second search strategy that is more suited to the k-cutset constraint: LCFirst minDeltaDeg. We recall that given an edge $e = (i, j)$, minDeltaDeg selects the edge with the minimum difference between the sum of the number of optional neighbors of i and j and the sum of the number of mandatory neighbors of i and j. Indeed, the more there are mandatory edges in the graph, the more we get a chance to find a pruning or a failing k-cutset. Then, in Table 3.3, we show the impact of the integration of the k-cutset constraint with LCFirst minDeltaDeg. We notice that the search space reduction is quite good for most of the instances. For instance, a gain of a factor 11.6 is observed for a280, 3.6 for ch150. Since the search space reduction is significant, the solving times are also reduced. For instance, a gain of a factor 5.5 is observed for a280, 1.9 for ch150. In addition, we notice that some problem such as gr431 remain unsolved without the k-cutset constraint in 24h whereas it is solved in 1,308s with the k-cutset constraint. That is at least a gain of a factor 66.1. Then, the search strategy LCFirst minDeltaDeg is well suited for the integration of the k-cutset constraint.

In Table 3.4, we compare the WCC with the best search strategy from the paper of Fages et al. [Fages et al., 2016] and the WCC with LCFirst minDeltaDeg (also from Fages et al.) with the k-cutset constraint. We notice that a clear improvement for most of the instances is obtained for both solving times and reduction of the search space. Note that in (2) all the instances are solved whereas in (1) 2 instances remain unsolved. Most of the time, we obtain a good reduction of the solving times. Considering the mean time of (1) and (2) without considering the instances that have reached the timeout in (1), we obtain 826s for (1) and 158s for (2). The search space is also greatly reduced on average: 946,296 search nodes for (1) and 90,160 search nodes for (2). Looking more closely the results, it seems that the k-cutset constraint allows a better solving of the larger instances. Indeed, on small instances the overhead of the k-cutset constraint sometimes led to slower solving times. Note that most of the time the difference is marginal. On "medium" instances the results are quite good, except for si175. Nevertheless, there is an initial huge difference for this instance without the k-cutset constraint for LCFirst maxCost (67.5s) and for LCFirst minDeltaDeg (1,004s). In Table 3.3, the k-cutset constraint allows reducing the solving time by a factor of 3.6. However, it is not sufficient to overcome the bad branching choice obtained by LCFirst minDeltaDeg.

In Table 3.5, we study the size of the found k-cutsets in linear full k-cutset with LCFirst minDeltaDeg. We can observe that the mean size the found k-cutsets is 2.7. It confirms the fact that the quadratic algorithm for the k-cutset constraint algorithm already finds a large part of the k-cutsets. However, larger k-cutsets exist: the average number of maximum sizes of the k-cutsets is 10.8. This is why we obtain a more interesting gain for the number of search nodes than for the basic model whereas the average k-cutset size is 2.7.

Finally, our linear full k-cutset algorithm is simple to implement and allows us to obtain an improvement of the solving times and the number of backtracks.

	WCC LCFirstMaxCost (1)		WCC + k-cutset LCFirstMaxCost (2)		ratio (1)/(2)	
Instances	time(s)	#sn	time(s)	#sn	time	#sn
a280	10.3	15,199	5.3	2,507	1.9	6.1
bier127	0.3	245	0.2	59	1.5	4.2
brg180	1.4	6,481	1.6	5,075	0.9	1.3
ch130	1.5	2,255	2.6	2,407	0.6	0.9
ch150	1.7	2,257	2.2	2,269	0.8	1.0
d198	20.2	21,297	46.5	46,957	0.4	0.5
d493	*t.o.*	*t.o.*	*t.o.*	*t.o.*	-	-
eil101	0.1	91	0.1	67	1.0	1.4
gil262	21,002.6	26,099,327	61,778.6	39,576,851	0.3	0.7
gr120	0.7	773	0.6	477	1.2	1.6
gr137	2.8	3,379	2.8	2,361	1.0	1.4
gr202	2.1	2,583	2.7	2,369	0.8	1.1
gr229	588.6	852,371	459.7	422,953	1.3	2.0
gr431	3,571.6	1,140,159	7,944.1	1,686,323	0.4	0.7
gr96	0.6	813	0.9	1,099	0.7	0.7
kroA100	1.5	3,159	2.7	4,413	0.6	0.7
kroA150	5.6	7,569	32.6	40,761	0.2	0.2
kroA200	2,099.6	2,882,359	1,250.8	925,947	1.7	3.1
kroB100	2.2	5,255	5.7	9,159	0.4	0.6
kroB150	148.7	250,253	300.5	314,805	0.5	0.8
kroB200	94.5	113,679	308.1	266,233	0.3	0.4
kroC100	0.6	799	1.5	1,895	0.4	0.4
kroD100	0.5	355	0.3	163	1.7	2.2
kroE100	1.6	3,883	2.6	3,967	0.6	1.0
lin318	16.1	9,563	12.7	6,315	1.3	1.5
pcb442	*t.o.*	*t.o.*	*t.o.*	*t.o.*	-	-
pr124	0.6	503	0.7	507	0.9	1.0
pr136	111.8	231,171	107.7	143,517	1.0	1.6
pr144	0.6	353	0.9	335	0.7	1.1
pr264	4.4	599	6.1	1,033	0.7	0.6
rat195	48.1	55,613	242.6	242,079	0.2	0.2
rat99	0.1	55	0.1	49	1.0	1.1
si175	67.5	133,667	528.5	589,901	0.1	0.2
st70	0.1	129	0.1	61	1.0	2.1
tsp225	283.4	327,727	2,167.7	1,881,733	0.1	0.2
u159	0.3	137	0.3	71	1.0	1.9
mean	$>$ 5,580.3		$>$ 6,889.4			
geo mean	$>$ 11.1		$>$ 16.6			

Table 3.2: Comparison of LCFirst maxCost with the integration of the k-cutset constraint.

	WCC		WCC + k-cutset			
	LCFirstMinDeltaDeg (1)		LCFirstMinDeltaDeg (2)		ratio (1)/(2)	
Instances	time(s)	#sn	time(s)	#sn	time	#sn
a280	30.6	28,789	5.6	2,485	5.5	11.6
bier127	0.3	153	0.3	71	1.0	2.2
brg180	0.4	95	0.4	95	1.0	1.0
ch130	2.6	3,661	1.3	1,069	2.0	3.4
ch150	2.5	3,187	1.3	873	1.9	3.7
d198	6.5	4,573	7.8	4,783	0.8	1.0
d493	*t.o.*	*t.o.*	67,961.8	11,346,181	> 1.3	-
eil101	0.1	89	0.1	65	1.0	1.4
gil262	7,160.4	6,345,749	2,842.1	1,711,411	2.5	3.7
gr120	0.6	585	0.5	279	1.2	2.1
gr137	2.2	3,135	1.4	1,083	1.6	2.9
gr202	2.0	2,023	1.4	831	1.4	2.4
gr229	382.8	437,417	132.5	114,435	2.9	3.8
gr431	*t.o.*	*t.o.*	1,307.9	247,091	> 66.1	-
gr96	0.6	555	0.6	549	1.0	1.0
kroA100	2.1	4,469	1.1	1,259	1.9	3.5
kroA150	5.8	7,277	3.0	2,799	1.9	2.6
kroA200	2,639.3	2,846,977	312.6	200,393	8.4	14.2
kroB100	2.2	4,087	2.6	3,309	0.8	1.2
kroB150	113.7	169,545	146.0	154,003	0.8	1.1
kroB200	189.7	189,965	127.1	109,323	1.5	1.7
kroC100	0.8	1,259	0.9	1,035	0.9	1.2
kroD100	0.3	283	0.4	247	0.8	1.1
kroE100	3.2	9,947	1.7	2,213	1.9	4.5
lin318	33.7	17,601	10.3	3,457	3.3	5.1
pcb442	*t.o.*	*t.o.*	15,578.5	5,555,757	> 5.5	-
pr124	1.7	2,031	2.6	2,483	0.7	0.8
pr136	12.3	18,075	16.8	21,447	0.7	0.8
pr144	1.0	795	1.3	975	0.8	0.8
pr264	3.4	679	4.3	643	0.8	1.1
rat195	23.0	22,583	32.9	27,513	0.7	0.8
rat99	0.1	95	0.1	81	1.0	1.2
si175	1,003.7	2,044,011	276.4	358,677	3.6	5.7
st70	0.1	87	0.2	77	0.5	1.1
tsp225	266.7	267,537	128.3	89,947	2.1	3.0
u159	0.5	459	0.6	421	0.8	1.1
mean	$> 7,530.4$		2,469.8			
geo mean	> 12.6		8.0			

Table 3.3: Comparison of LCFirst minDeltaDeg with the integration of the k-cutset constraint.

	WCC LCFirstMaxCost (1)		WCC + k-cutset LCFirstMinDeltaDeg (2)		ratio (1)/(2)	
Instances	time(s)	#sn	time(s)	#sn	time	#sn
a280	10.3	15,199	5.6	2,485	1.8	6.1
bier127	0.3	245	0.3	71	1.0	3.5
brg180	1.4	6,481	0.4	95	3.5	68.2
ch130	1.5	2,255	1.3	1,069	1.2	2.1
ch150	1.7	2,257	1.3	873	1.3	2.6
d198	20.2	21,297	7.8	4,783	2.6	4.5
d493	*t.o.*	*t.o.*	67,961.8	11,346,181	> 1.3	-
eil101	0.1	91	0.1	65	1.0	1.4
gil262	21,002.6	26,099,327	2,842.1	1,711,411	7.4	15.3
gr120	0.7	773	0.5	279	1.4	2.8
gr137	2.8	3,379	1.4	1,083	2.0	3.1
gr202	2.1	2,583	1.4	831	1.5	3.1
gr229	588.6	852,371	132.5	114,435	4.4	7.4
gr431	3,571.6	1,140,159	1,307.9	247,091	2.7	4.6
gr96	0.6	813	0.6	549	1.0	1.5
kroA100	1.5	3,159	1.1	1,259	1.4	2.5
kroA150	5.6	7,569	3.0	2,799	1.9	2.7
kroA200	2,099.6	2,882,359	312.6	200,393	6.7	14.4
kroB100	2.2	5,255	2.6	3,309	0.8	1.6
kroB150	148.7	250,253	146.0	154,003	1.0	1.6
kroB200	94.5	113,679	127.1	109,323	0.7	1.0
kroC100	0.6	799	0.9	1,035	0.7	0.8
kroD100	0.5	355	0.4	247	1.3	1.4
kroE100	1.6	3,883	1.7	2,213	0.9	1.8
lin318	16.1	9,563	10.3	3,457	1.6	2.8
pcb442	*t.o.*	*t.o.*	15,578.5	5,555,757	> 5.5	-
pr124	0.6	503	2.6	2,483	0.2	0.2
pr136	111.8	231,171	16.8	21,447	6.7	10.8
pr144	0.6	353	1.3	975	0.5	0.4
pr264	4.4	599	4.3	643	1.0	0.9
rat195	48.1	55,613	32.9	27,513	1.5	2.0
rat99	0.1	55	0.1	81	1.0	0.7
si175	67.5	133,667	276.4	358,677	0.2	0.4
st70	0.1	129	0.2	77	0.5	1.7
tsp225	283.4	327,727	128.3	89,947	2.2	3.6
u159	0.3	137	0.6	421	0.5	0.3
mean	> 5580.3		2469.8			
geo mean	> 11.1		8.0			

Table 3.4: Comparison of the state-of-the-art model with its best search strategy (WCC and LCFirstMaxCost) and the WCC with the k-cutset constraint and its best search strategy (LCFirst minDeltaDeg).

	k-cutset size	
Instances	mean	max
a280	3.31	9
bier127	2.53	6
brg180	0	0
ch130	2.46	7
ch150	3.32	10
d198	2.4	13
d493	2.43	26
eil101	2.4	4
gil262	2.55	17
gr120	2.5	6
gr137	2.5	8
gr202	2.57	9
gr229	3.08	14
gr431	2.45	16
gr96	2.29	8
kroA100	2.46	6
kroA150	2.86	8
kroA200	2.79	10
kroB100	2.63	7
kroB150	2.21	11
kroB200	2.96	13
kroC100	2.19	6
kroD100	2.39	6
kroE100	2.72	7
lin318	5.47	14
pcb442	3.53	24
pr124	2.5	10
pr136	2.57	14
pr144	2.49	8
pr264	2.48	25
rat195	4.77	26
rat99	2.75	6
si175	2.63	13
st70	2.68	4
tsp225	2.93	13
u159	2.28	6
mean	2.7	10.8

Table 3.5: Comparison of mean and max k-cutsets size.

3.4.6 Conclusion

In this section, we introduced the k-cutset constraint that is a constraint based on the cutsets in a graph. It enforces that each cutset of the graph has a positive even number of edges in any TSP solution. For this constraint, we explained our first works, that is a quadratic algorithm handling the k-cutset constraint such that $k \leq 3$ [Isoart and Régin, 2019]. Next, we introduced a linear algorithm handling any k [Isoart and Régin, 2021b]. With a static search strategy, we observed that the number of search nodes is reduced by an order of magnitude and the solving times are globally significantly reduced. In addition, we have shown that on average most of the cutsets are of size 2.7 even if we found some much larger cutsets. In future works, we hope that a better understanding between LCFirst and the k-cutset constraint will be achieved. It should also be noted that correcting structural defects of the LP relaxation is the game changer for Concorde [Applegate et al., 2006] and the k-cutset constraint which is based on the structure of the graph allows a great improvement of the results. Thus, there is a strong relationship between solving the TSP and used the structural property of the TSP [Isoart and Régin, 2020].

3.5 Mandatory Hamiltonian path constraint

In this section, we introduce our works on another structural constraint [Isoart and Régin, 2021a]. This constraint is based on the integration of 2-opt and 3-opt concepts into CP. Unlike tour improvement algorithms, the CP model does not have a tour to improve. However, the CP model has mandatory edges that can form paths and try to find a tour going through these paths. Therefore, we introduce the search for 2-opt and 3-opt in the paths of mandatory edges.

For each node i, $|M(i)| \leq 2$ because of the degree constraint. Thus, the mandatory edges form disjoint paths. Without loss of generality, we assume that the current assignment of G is consistent with the degree constraint.

Definition 3.5.1 (Mandatory Hamiltonian path)**.** A mandatory Hamiltonian path p is a path such that p is a Hamiltonian path in a subgraph of G and for each edge $e = (x_i, x_{i+1})$ of p, $e \in M$.

We note $p_1 = [x_1, x_2, \ldots, x_t]$ a mandatory Hamiltonian path of G.

3.5.1 Consistency Check

In this section, we will study if the current assignment of the mandatory edges can lead to optimal solutions in G.

Definition 3.5.2 (Alternative path)**.** An **alternative path** $p_2 = [x'_1, x'_2, \ldots, x'_t]$ of p_1 is a permutation of the nodes of p_1 such that $p_1 \neq p_2$, $x_1 = x'_1$, $x_t = x'_t$ and for each $i \in [1, k-1]$, $(x'_i, x'_{i+1}) \in U$.

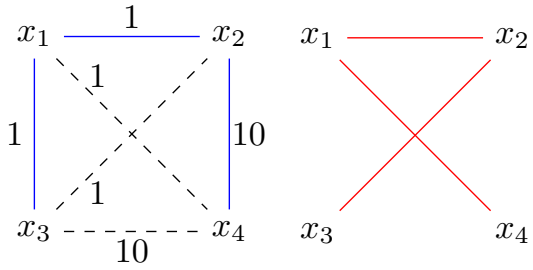


Figure 3.12: The left graph is a subgraph of G. The blue edges are from M, they form a mandatory Hamiltonian path going from x_3 to x_4. The dashed edges are from D (the deleted edges). The red edges from the right graph represent an alternative path of the blue edges from left graph.

Figure 3.12 shows an example of an alternative path. Thus, an alternative path can be composed of edges in $M \cup O \cup D$, *i.e.* in U.

Definition 3.5.3 (Minimal mandatory Hamiltonian path)**.** The mandatory Hamiltonian path p_1 is minimal if and only if there is no alternative path p_2 of p_1 such that $w(p_2) < w(p_1)$.

In Figure 3.12, the mandatory Hamiltonian path going from x_3 to x_4 is not minimal. The right graph represents an alternative path with a cost of 3 whereas the mandatory Hamiltonian path has a cost of 12. The idea is to search for a non-minimal mandatory Hamiltonian path p_2 in

the connected components of $G_{mand} = (X, M, \emptyset)$. In Proposition 3.5.1, we show that if such a path p_2 exists, then the cost of the TSP in $G = (X, M, O)$ is greater than the cost of the TSP in $G_{init} = (X, M_{init}, O_{init})$.

Proposition 3.5.1. *If there is a mandatory Hamiltonian path p_1 that is not minimal, then p_1 cannot belong to any optimal solution of $TSP(G_{init})$.*

Proof.

Given $w(TSP(G_{init} + p_1))$ the cost of the optimal solution of $TSP(G_{init})$ such that p_1 is in the solution. If there is no solution for $TSP(G_{init} + p_1)$, then p_1 cannot belong to any solution $TSP(G_{init})$. Otherwise, if p_1 is not minimal, then there is an alternative path p_2 of p_1 such that $w(p_2) < w(p_1)$. Thus, $w(TSP(G_{init} + p_2)) < w(TSP(G_{init} + p_1))$ and therefore p_1 cannot belong to any optimal solution of $TSP(G_{init})$. □

In the context of a branch and bound, if there is a mandatory Hamiltonian path p that is not minimal, then from Proposition 3.5.1 we can trigger a failure because the current solution is not minimal. Moreover, it raises a question: how do we verify if a mandatory Hamiltonian path is minimal? A first algorithm consists in checking all the possible permutations for each mandatory Hamiltonian path. Unfortunately, checking all the permutations leads to an impractical algorithm. However, many heuristics improving tours have been designed. Among them, there are the ones introduced in Section 2.3.2. Thus, we can use any of these heuristics on the mandatory Hamiltonian paths. If it finds an improvement, then a mandatory Hamiltonian path is not minimal and therefore we can trigger a failure.

In this paper, we use k-opt heuristics as tour improvement since they are very efficient and easy to implement. In Section 3.5.5, we will show that 2-opt and 3-opt are enough in order to obtain good results.

Definition 3.5.4 (Mandatory Hamiltonian path constraint)**.** Given the set of mandatory Hamiltonian paths P and an integer k. For each mandatory Hamiltonian path $p \in P$, the mandatory Hamiltonian path constraint ensures that there is no alternative path p' obtained by swapping k edges of p such that $w(p') < w(p)$.

Therefore, we define the mandatory Hamiltonian path constraint in Definition 3.5.4 such that if its consistency is not verified, then we trigger a failure.

Algorithm 6: Consistency check of the mandatory Hamiltonian paths.

```
ConsistencyCheck (G_init, G_mand, k)
    Input: The initial graph G_init, the graph of mandatory edges G_mand and an integer k.
    Output: A Boolean specifying whether G_mand contains a mandatory Hamiltonian
            path that is not minimal.
    P ← computeMandatoryHamiltonianPaths(G_mand) ;
    foreach path p ∈ P do
        if k-optPath(G_init, p) then
            return False ;
    return True ;
```

In Algorithm 6, we introduce an implementation of the algorithm checking the consistency of the mandatory Hamiltonian path constraint. We assume that k-optPath(G_{init}, p) returns true if and only if the mandatory Hamiltonian path constraint with the given k is inconsistent. Internally, k-optPath(G_{init}, p) uses a k-opt heuristic. Then, for each mandatory Hamiltonian path p, we run k-optPath(G_{init}, p) in $O(|p|^k)$.

Proposition 3.5.2. *Given P the set of mandatory Hamiltonian paths. Then, $\sum_{p \in P} |p| \leq n$ and $|P| \leq n$.*

Proof.

By definition, each node can only be contained in one mandatory Hamiltonian path and there are n nodes in G. Thus, $\sum_{p \in P} |p| \leq n$. In addition, if each node is contained in a different path, then there are n paths and then therefore $|P| \leq n$. □

Each p of P are disjoint. From Proposition 3.5.2, the sum of $|p|$ for all $p \in P$ is lower or equal to n. Finally, the time complexity of Algorithm 6 is in $O(\sum_{p \in P} |p|^k) < O(n^k)$.

3.5.2 Filtering algorithm

In this section, we will consider that the consistency has been checked. An edge $e = (x_t, x_i)$ is a successor of p_1 and an edge $e = (x_i, x_1)$ is a predecessor of p_1. In addition, we note $x_i + p_1 = [x_i, x_1, x_2, \ldots, x_t]$ and $p_1 + x_i = [x_1, x_2, \ldots, x_t, x_i]$.

From Proposition 3.5.1, we have the two following corollaries:

Corollary 3.5.3. *For each edge $e \in O$ such that e is a predecessor of p_1, if $i + p_1$ is not a minimal mandatory Hamiltonian path, then e cannot belong to any optimal solution of $TSP(G)$.*

Corollary 3.5.4. *For each edge $e \in O$ such that e is a successor of p_1, if $p_1 + i$ is not a minimal mandatory Hamiltonian path, then e cannot belong to any optimal solution of $TSP(G)$.*

Thus, in order to define a filtering algorithm, we are interested in the minimality of $p_1 + i$ and $i + p_1$. If the minimality of p_1 has already been checked, then we can avoid the permutations containing only the elements of p_1. We impose i to be in the considered permutations and we look for permutations of size $(k-1)$ in p_1. Then, for a mandatory Hamiltonian path p and a single successor or predecessor, we can filter the edge in $O(|p|^{k-1}) < O(n^{k-1})$. Performing the filtering for all predecessors and successors of p can be done in $O(|O(p)||p|^{k-1}) < O(n^k)$. From Proposition 3.5.2, there can be at most n paths and the sum of the size of all paths is smaller than or equal to n. Thus, from Corollary 3.5.3 and Corollary 3.5.4 we can filter the edges of all paths with a complexity in $O(\sum_{p \in P} O(|O(p)||p|^{k-1})) < O(n^{k+1})$. Note that the number of checked permutations in practice is much smaller.

A mandatory Hamiltonian path p_1 can have a successor or a predecessor e connecting another mandatory Hamiltonian path p_2. Then, adding e to the solution leads to a minimality check in $p_1 + p_2$. We can then extend the two previous corollaries:

Corollary 3.5.5. *For each edge $e = (x_i, x_1) \in O(x_1)$, if there exists $p_2 = [x'_1, x'_2, \ldots, x'_t]$ a mandatory Hamiltonian path of G such that $x_i = x'_t$ and $p_2 + p_1$ is not a minimal mandatory Hamiltonian path, then e cannot belongs to any optimal solution of $TSP(G)$.*

Corollary 3.5.6. *For each edge $e = (x_i, x_t) \in O(x_t)$, if there exists $p_2 = [x'_1, x'_2, \ldots, x'_t]$ a mandatory Hamiltonian path of G such that $x_i = x'_1$ and $p_1 + p_2$ is not a minimal mandatory Hamiltonian path, then e cannot belongs to any optimal solution of $TSP(G)$.*

Given a mandatory Hamiltonian path p_2 of G connected to p_1 with $e \in O$. Then, we have to check if $p_1 + p_2$ is minimal in order to determine whether e can be in a solution of $TSP(G)$. It can be done with Corollary 3.5.5 and Corollary 3.5.6 in $O((|p_1| + |p_2|)^k - |p_1|^k - |p_2|^k) < O(n^k)$. The number of predecessors and successors of p_1 is at most $2n$. If $P(p_1)$ is the set of mandatory Hamiltonian paths such that each path of $P(p_1)$ is connected to p_1 with a successor or a predecessor of p_1, then the filtering on p_1 can be done in $O(\sum_{p_2 \in P(p_1)}(|p_1| + |p_2|)^k - |p_1|^k - |p_2|^k) < O(n^{k+1})$. Given P the set of mandatory Hamiltonian paths. The filtering for all paths of P can be done in $O(\sum_{p_1 \in P} \sum_{p_2 \in P(p_1)}(|p_1| + |p_2|)^k - |p_1|^k - |p_2|^k) < O(n^{k+2})$. Note that when we solve the TSP, the number of mandatory edges is often much smaller than n. In fact, when the number of mandatory edges is close to n then the problem is almost solved. Algorithm 7 is a possible implementation.

For the sake of clarity, we will use the following notations in the algorithms:

- P: contains all the mandatory Hamiltonian paths of the graph G.
- $P[i]$: if there is a path p containing the node i, then it returns p. Otherwise, it returns i.
- p.first(): returns the first node of the path p.
- p.last(): returns the last node of the path p.

Algorithm 7: Filtering algorithm for the mandatory Hamiltonian paths.

```
Filter (G_init, G = (X, M, O), P, k)
    Input: The initial graph G_init, a graph G, the set of mandatory Hamiltonian paths P
           and an integer k.
    foreach p_1 = [x_1, x_2, ..., x_t] ∈ P do
        foreach edge e = (x_1, j) ∈ O(x_1) do
            p_2 ← P(j) ;
            if p_2.last() ≠ j then reverse(p_2);
            if (p_1 = p_2 and |M| ≠ n − 1) or k-optPath(G_init, p_2, p_1) then
                O ← O − e ;
        foreach edge e = (x_t, j) ∈ O(x_t) do
            p_2 ← P(j) ;
            if p_2.first() ≠ j then reverse(p_2);
            if (p_1 = p_2 and |M| ≠ n − 1) or k-optPath(G_init, p_1, p_2) then
                O ← O − e ;
```

Given P the set of the mandatory Hamiltonian paths. For each path $p_1 \in P$, we perform the filtering on all the predecessor and successor e of p_1. We note p_2 the path connected to p_1 by e (p_2 can be a single node). In addition, we note k-optPath$(graph, p_1, p_2)$ the k-optPath algorithm considering the permutations of $p_1 + p_2$ such that each permutation contains at least one element

of p_1 and at least one element of p_2. When two paths are merged, they must be in the right order. If p_2 must be inserted in front of p_1, then the node j must be the last node of p_2. Otherwise, j must be the first node of p_2. Thus, p_2 is reversed if needed. Note that we can save the reversed path in order to avoid redundant computations. If an improvement is found when p_1 and p_2 are merged, then from Corollary 3.5.5 or Corollary 3.5.6 the edge e cannot belong to a solution of $TSP(G)$ and therefore e is removed from the optional edges of G. In addition, if $p_1 = p_2$ and $|M| \neq n - 1$, then there exists an edge $e = (i, j)$ such that i and j belong to the same mandatory Hamiltonian path and therefore the edge closes a cycle with a size lower than n. Thus, adding e to the solution creates a sub-cycle and then e is removed from the optional edges of G. Finally, we could improve the efficiency of our algorithms by considering only the closest neighbors of each node in the k-opt algorithm such as LKH [Lin and Kernighan, 1973, Helsgaun, 2000].

3.5.3 Maintenance during the search

In this section, we will consider the incremental aspect of this constraint, *i.e.* the consistency of this constraint and its filtering when some edges become mandatory or deleted. Moreover, we will consider the restoration of the data structures introduced for the incremental aspect when a backtrack occurs. In this study, an edge can be deleted or an edge becomes mandatory.

Proposition 3.5.7. *Given $O' \subseteq O$. If each mandatory Hamiltonian path of G is minimal and O' is removed from G, then each mandatory Hamiltonian path of G remain minimal.*

Proof.

By definition, the alternative paths can contain removed edges. Therefore, if a mandatory Hamiltonian path p is minimal, then removing some edges from the graph does not affect the minimality of p. □

From Proposition 3.5.7, if we know that all the mandatory Hamiltonian paths of G are minimal and then some edges are removed, then the mandatory Hamiltonian paths of G remain minimal. In addition, removing some edges does not change the result of the filtering algorithm since new alternative paths cannot be created from removal. Thus, the consistency test and the filtering algorithm are only triggered when there are new mandatory edges.

In the following algorithms, we use the following data structures:

- candidates: a stack of graph nodes such that each node is adjacent to edges that can be filtered.
- deltaMand: a set of the new mandatory edges since the last call of the constraint for the current search node.

3.5.3.1 Consistency check

When an edge e becomes mandatory, there are three cases:

1. e is not connected to any path.
2. e is connected to a mandatory Hamiltonian path p.

3. e is connected to two mandatory Hamiltonian paths p_1 and p_2.

In case 1, e creates a new path only containing its two endpoints. Note that a mandatory Hamiltonian path with two nodes is necessarily minimal.

In case 2, p and e are merged in a not necessarily minimal mandatory Hamiltonian path because new alternative paths may exist.

In case 3, p_1 and p_2 are merged in a not necessarily minimal mandatory Hamiltonian path because new alternative paths may exist.

Thus, for consistency check, we only consider the paths that must be merged. In addition, given a new mandatory edge e connecting p_1 and p_2, we note p_3 the merged path of p_1 and p_2. When two paths are merged, we assume that the minimality check has been performed on the two paths. Therefore, when the k-optPath algorithm is checking the minimality for the path p_3, it can avoid the permutations containing elements belonging to either p_1 or p_2. Then, we consider the permutations that contain at least one element of p_1 and at least one element of p_2. For cases 2 and 3, if the filtering algorithm has been run on the same paths connected to e, then we know that the mandatory Hamiltonian path created when e became mandatory is consistent since e was not removed by the pruning algorithm. Note that if a path connected to e has been modified (a node has been appended to it), then we do not know without running a consistency check if the mandatory Hamiltonian path created when e become mandatory is consistent.

In Algorithm 8, we give a possible implementation of the incremental algorithm checking the minimality of the mandatory Hamiltonian paths. For each edge (i, j) newly mandatory, we have p_1 and p_2 the mandatory Hamiltonian paths such that i and j are respectively an extremity of p_1 and p_2. Therefore, the edge (i, j) merge p_1 and p_2 and p_1 and/or p_2 are accordingly reversed. Note that the $candidates$ stack is filled for the filtering algorithm. Then, we run the k-optPath algorithm in order to find alternative paths in $p_1 + p_2$. Note that we only consider permutations such that each permutation contains at least one element of p_1 and at least one element of p_2. Finally, if no alternative path is found, $p_1 + p_2$ is a minimal mandatory Hamiltonian path and we merge p_1 and p_2 in P. Otherwise, we return False and a failure is triggered.

The overall time complexity of Algorithm 8 is $O(\sum_{(i,j)\in deltaMand}(|P[i]|+|P[j]|)^k-|P[i]|^k-|P[j]|^k) < O(n^k)$. Note that Algorithm 6 has a time complexity in $O(\sum_{p\in P}|p|^k) < O(n^k)$ when all paths are already merged which is equivalent to $O(\sum_{(i,j)\in deltaMand}(|P[i]|+|P[j]|)^k)$ if paths are not merged. Thus, the incremental algorithm improves the time complexity for checking the minimality of the mandatory Hamiltonian paths.

3.5.3.2 Filtering algorithm

When an edge e becomes mandatory, we have the same three cases as for the consistency check. Thus, we will only consider the merged mandatory Hamiltonian paths in the previous consistency check. More precisely, we will only consider the neighborhood of the first node and the last node of these paths.

Algorithm 9 is a possible implementation of an incremental algorithm performing the filtering. First, we iterate on $candidates$. Every time two paths are merged in Algorithm 8, the first and last nodes of the merged path are pushed in $candidates$. Thus, $candidates$ contains the first node and last nodes of all merged paths. In addition, $candidates$ may contain some nodes that are "intermediate" merged paths. For example, if merging p_1 and p_2 results in p_3 such that $p_3.first() = x$ and $p_3.last() = y$, then x and y are pushed in $candidates$. Merging p_3 with p_4 results in p_5 such

Algorithm 8: Incremental minimality check of the mandatory Hamiltonian paths

IncrementalConsistencyCheck $(G_{init}, P, deltaMand, candidates, k)$

Input: The initial graph G_{init}, the set of mandatory Hamiltonian paths P, the set of new mandatory edges $deltaMand$, $candidates$ a filtering used stack and an integer k.

Output: A Boolean specifying whether P contains a mandatory Hamiltonian path that is not minimal.

foreach $(i, j) \in deltaMand$ **do**
- $p_1 \leftarrow P[i]$;
- $p_2 \leftarrow P[j]$;
- **if** $p_1.last() \neq i$ **then** reverse(p_1);
- **if** $p_2.first() \neq j$ **then** reverse(p_2);
- $candidates$.push($p_1.first()$) ;
- $candidates$.push($p_2.last()$) ;
- **if** k-*optPath*(G_{init}, p_1, p_2) **then**
  - **return** *False* ;
- `// merge` p_1 `and` p_2 `in` P
- merge(P, p_1, p_2) ;

return *True* ;

that $p_5.first() = x'$ and $p_5.last() = y'$. Then, x' and y' are pushed in $candidates$. However, x and y still are in $candidates$ while x or y is no longer the first node or the last node of a merged path. Then, while we iterate on candidates, we need to avoid these nodes. To do so, for each node i of candidates, we check if i is either the first node or the last node of the mandatory Hamiltonian path containing i. Note that we could also delete these nodes when the paths are merged with a proper data structure. Finally, if a node i is an endpoint of a mandatory Hamiltonian path, then we check in the neighborhood of the node i (same as for Algorithm 7).

If $P(i)$ is the set of mandatory Hamiltonian paths such that each path of $P(i)$ is connected with a successor or a predecessor of $P[i]$, then the time complexity of Algorithm 9 is in $O(\sum_{i \in candidates} \sum_{p_2 \in P(i)} (|P[i]| + |p_2|)^k - |P[i]|^k - |p_2|^k)) < O(n^{k+2})$.

3.5.3.3 Restoration

In order to save more computations, we maintain the set P of mandatory Hamiltonian paths. When a backtrack occurs, the difference between the backtracked state and the current state is that some mandatory edges could have been found and therefore some mandatory Hamiltonian paths of P could have been merged. Thus, in order to restore P, the merged mandatory Hamiltonian paths should be split. To do so, we define a stack S such that S contains the added mandatory edges from the root to the current state. In addition, we save the size of the stack for each open search node. Then, when a backtrack occurs, we iteratively pop the mandatory e edges of S until the wanted size is obtained. For each e, we split the mandatory Hamiltonian path in P containing e.

Algorithm 9: Incremental filtering of the mandatory Hamiltonian paths.

IncrFiltering $(G_{init}, G = (X, M, O), P, candidates, k)$
  Input: The initial graph G_{init}, a graph G, the set of mandatory Hamiltonian paths P, the stack of nodes to consider for the filtering $candidates$ and an integer k.
  while *candidates.isNotEmpty()* **do**
    $i \leftarrow candidates$.pop() ;
    $p_1 \leftarrow P[i]$;
    if i = p_1.*first()* ***or*** i = p_1.*last()* **then**
      foreach *edge* $e = (i, j) \in O(i)$ **do**
        $p_2 \leftarrow P[j]$;
        if $p_1.last() \neq i$ **then** reverse(p_1);
        if $p_2.first() \neq j$ **then** reverse(p_2);
        if ($p_1 = p_2$ ***and*** $|M| \neq n - 1$) ***or*** k-*optPath*(G_{init}, p_1, p_2) **then**
          $O \leftarrow O - e$;

3.5.4 Discussion

For this constraint, we used the k-opt algorithm in order to find if a mandatory Hamiltonian path is minimal. However, we could have used any other tour improvement algorithm.

This idea can be generalized for many other optimization problems. Indeed, the purpose is to detect if the current solution can be re-optimized. If it does, then the current solution cannot be the optimal one.

For instance, consider the linear ordering problem. Given a cost matrix $C_{n,n}$ and list L of n elements, it consists in finding an order L such that $\sum_{i=1}^{n} \sum_{j=i}^{n} c_{i,j}$ is minimized or maximized. As an example, let us take $L = [3, 5, 2, 1, 4]$. Then, if we expand the sum of the objective function we obtain:

$$c_{3,5} + c_{3,2} + c_{3,1} + c_{3,4} + c_{5,2} + c_{5,1} + c_{5,4} + c_{2,1} + c_{2,4} + c_{1,4} \tag{3.1}$$

In the minimization variant of this problem, a classical dominance rule is that if $c_{i,j} < c_{j,i}$ and both i and j are consecutive in L, then i must be placed before j in L.

Then, considering $i = L[2] = 2$ and $j = L[3] = 1$, swapping i and j, we obtain the list $L' = [3, 5, 1, 2, 4]$. Again, if we expand the sum of the objective function we obtain:

$$c_{3,5} + c_{3,1} + c_{3,2} + c_{3,4} + c_{5,1} + c_{5,2} + c_{5,4} + c_{1,2} + c_{1,4} + c_{2,4} \tag{3.2}$$

We observe here that the only difference in the sum is that L considers $c_{2,1}$ whereas L' considers $c_{1,2}$. Therefore, if $c_{2,1} < c_{1,2}$, then the cost of the objective function of L will be lower than the one of L'.

Thus, we can apply the idea we used for the mandatory Hamiltonian path constraint for this problem. In CP, we will model this problem with n variables such that each variable has an initial domain in $[1, n]$ that is the position in L. Then, an allDifferent constraint [Régin, 1994], on these variables will be added. Therefore, through the solving process, variable assignment will be made thanks to the branch-and-bound and the filtering algorithm of the allDifferent constraint. The idea is then to use the dominance rule on the assigned variables, if a re-optimization is found, then

the current branch of the search tree can be safely aborted. For instance, imagine that $5, 2, 1$ are assigned in $L = [3, 5, 2, 1, 4]$. If $c_{5,2} > c_{2,5}$ or $c_{5,1} > c_{1,5}$ or $c_{2,1} > c_{1,2}$, then the current branch of the search tree can be safely aborted.

This dominance rule can be generalized using any local search heuristic: if any heuristic shows that 5,2,1 is not optimal, then the current search tree branch can be aborted. For instance, make the hypothesis that 1,2,5 has a cost lower than 5,2,1 and that 1,5,2 is in the optimal solution. Then, finding that 1,2,5 has a cost lower than 5,2,1 is enough in order to prune the search tree: the optimal solution is not needed. In the same spirit of the mandatory Hamiltonian path constraint, a filtering algorithm can be developed. For each non-assigned variable x, for each value v of x such that v is consecutive to an assigned value in L. If assigning x to v leads to a re-optimization, then v can be safely removed from x. Note that an incremental algorithm can be easily developed.

Finally, we believe that this method could be extended to many other problems and that many constraints based on local search heuristics could be developed.

3.5.5 Experiments

In this section we will experimentally show that the mandatory Hamiltonian path constraint allows reducing the search space and therefore improve the solving times. To do so, we use the configuration introduced in Section 1.1. Specifically, we always use WCC and the k-cutset constraint, named "basic model". In addition, we note "MHP 2-opt" the basic model combined with the mandatory Hamiltonian path constraint searching for 2-opt and "MHP 3-opt" the basic model combined with the mandatory Hamiltonian path constraint searching for 3-opt. Unless specified, we will use the implementation given in Algorithm 8 and 9.

Table 3.6 shows the solving times and the number of search nodes for the basic model, MHP 2-opt and MHP 3-opt. In addition, we display a ratio column in order to show the gain factor for each instance by using 2-opt and 3-opt.

For the basic model, we notice that 4 instances over 40 have reached the timeout. For MHP 2-opt, we notice that no instance has reached the timeout. Indeed, ali535 is solved in 14,659s, gr666 is solved in 55,898s, pr299 is solved in 8,773s and rd400 is solved in 13,605s with 2-opt whereas they remain unsolved in 86,400s with the basic model. Most of the time, we notice that the use of 2-opt allows us to improve the solving times. For example, gr431 is improved by a factor of 3.0 in solving time and by a factor of 3.8 in search nodes. Some problems have higher gain factors: d493 gains a factor 5.4 in solving time and a factor 4.5 in search nodes. Moreover, 38 over 40 instances have improved solving times. Only two instances have a degraded solving time when used 2-opt: brg180 is 0.1s slower and pcb442 is 24% slower. As we previously said in Section 3.4.5, the Lagrangian relaxation interferes with the filtering algorithms, and the search strategy is dynamic, it can happen that more pruning does not leads to a reduction of the search space.

The mandatory Hamiltonian path constraint combined with 3-opt allow us to obtain an additional improvement to the use of 2-opt only. Indeed, 3-opt can be slower than 2-opt in terms of search nodes/second but it greatly reduces the number of search nodes. Note that in the paper introducing the Mandatory Hamiltonian path constraint [Isoart and Régin, 2021a], we were able to solve larger instances only with the use of 3-opt by considering a timeout of 1 week. Some instances are solved much faster with 3-opt than with 2-opt: gr666 reach the timeout with the basic model, it is solved in 55,899s with 2-opt and 19,102s with 3-opt. Some other instances are solved with almost the same number of search nodes for 2-opt and 3-opt: for ali535 with 2-opt there are

	Basic model (1)		MHP 2-opt (2)		ratio (1)/(2)		MHP 3-opt (3)		ratio (1)/(3)	
Instances	time(s)	#sn	time(s)	#sn	time	#sn	time(s)	#sn	time(s)	#sn
a280	5.6	2,485	4.7	2,027	1.2	1.2	3.6	1,267	1.6	2.0
ali535	*t.o.*		14,658.8	2,442,469	> 5.9	-	23,757.1	2,716,581	> 3.6	-
bier127	0.3	71	0.2	53	1.5	1.3	0.2	53	1.5	1.3
brg180	0.4	95	0.5	101	0.8	0.9	1.1	101	0.4	0.9
ch130	1.3	1,069	0.7	389	1.9	2.7	0.8	401	1.6	2.7
ch150	1.3	873	1.0	523	1.3	1.7	0.9	493	1.4	1.8
d198	7.8	4,783	6.3	3,451	1.2	1.4	6.6	2,791	1.2	1.7
d493	67,961.8	11,346,181	12,508.3	2,523,995	5.4	4.5	7,715.3	746,881	8.8	15.2
eil101	0.1	65	0.1	65	1.0	1.0	0.2	65	0.5	1.0
gil262	2,842.1	1,711,411	4,159.0	2,518,087	0.7	0.7	2,792.2	1,519,741	1.0	1.1
gr120	0.5	279	0.5	263	1.0	1.1	0.5	285	1.0	1.0
gr137	1.4	1,083	1.0	711	1.4	1.5	1.1	721	1.3	1.5
gr202	1.4	831	1.3	517	1.1	1.6	1.3	531	1.1	1.6
gr229	132.5	114,435	45.2	32,363	2.9	3.5	40.2	27,725	3.3	4.1
gr431	1,307.9	247,091	429.2	65,549	3.0	3.8	379.2	51,317	3.4	4.8
gr666	*t.o.*		55,898.6	5,741,233	> 1.5	-	19,101.7	1,669,653	> 4.5	-
gr96	0.6	549	0.5	311	1.2	1.8	0.5	283	1.2	1.9
kroA100	1.1	1,259	0.7	557	1.6	2.3	0.7	425	1.6	3.0
kroA150	3.0	2,799	3.0	2,395	1.0	1.2	2.6	2,035	1.2	1.4
kroA200	312.6	200,393	58.3	35,821	5.4	5.6	70.6	39,305	4.4	5.1
kroB100	2.6	3,309	1.3	1,387	2.0	2.4	1.1	1,271	2.4	2.6
kroB150	146.0	154,003	10.7	8,745	13.6	17.6	10.8	8,481	13.5	18.2
kroB200	127.1	109,323	18.3	12,851	6.9	8.5	14.9	10,045	8.5	10.9
kroC100	0.9	1,035	0.5	305	1.8	3.4	0.6	305	1.5	3.4
kroD100	0.4	247	0.3	173	1.3	1.4	0.4	159	1.0	1.6
kroE100	1.7	2,213	0.9	889	1.9	2.5	0.9	793	1.9	2.8
lin318	10.3	3,457	5.6	1,711	1.8	2.0	4.7	1,341	2.2	2.6
pcb442	15,578.5	5,555,757	19,319.6	6,784,259	0.8	0.8	79,639.0	11,865,739	0.2	0.5
pr124	2.6	2,483	1.3	1,037	2.0	2.4	1.3	915	2.0	2.7
pr136	16.8	21,447	7.7	7,557	2.2	2.8	8.9	8,899	1.9	2.4
pr144	1.3	975	1.0	579	1.3	1.7	1.1	557	1.2	1.8
pr264	4.3	643	4.0	825	1.1	0.8	3.4	397	1.3	1.6
pr299	*t.o.*		8,772.9	3,040,171	> 9.8	-	2,429.7	737,971	> 35.6	-
rat195	32.9	27,513	23.3	17,915	1.4	1.5	15.4	10,661	2.1	2.6
rat99	0.1	81	0.1	75	1.0	1.1	0.2	75	0.5	1.1
rd400	*t.o.*		13,604.8	4,140,613	> 6.4	-	6,496.9	1,882,547	> 13.3	-
si175	276.4	358,677	168.8	196,943	1.6	1.8	169.9	163,917	1.6	2.2
st70	0.2	77	0.1	73	2.0	1.1	0.2	73	1.0	1.1
tsp225	128.3	89,947	76.2	44,567	1.7	2.0	30.1	16,129	4.3	5.6
u159	0.6	421	0.4	203	1.5	2.1	0.4	193	1.5	2.2
mean	$> 10{,}862.8$		3,244.9	690,794.0			3,567.7	537,278.1		

Table 3.6: General results comparing the mandatory Hamiltonian path constraint combined with either 2-opt or 3-opt and the basic model.

3,442,470 search nodes and there are 2,716,582 search nodes with 3-opt. However, it has a much slower solving time with 3-opt than with 2-opt: 14,658s vs 23,757s. In addition, pcb442 is solved faster with 2-opt than with 3-opt: 19,320s vs 79,639s. This can be due to several reasons: the extra cost of using an algorithm in $O(n^3)$ compared to an algorithm in $O(n^2)$. Therefore, if the solving of an instance does not contain that much Mandatory Hamiltonian path with lower cost alternative path, some solving time can be lost. The Lagrangian relaxation can also be impacted by the filtered edges. In addition, LCFirst can be disrupted by this filtering algorithm. Note that some instances such as pcb442 are very particular problems (drilling problem). However, since 3-opt allows a better reduction of the number of search nodes (especially on the larger instances), we will consider the version with 3-opt. Note that we could use some other heuristics such as 2.5-opt that compute 2-opt and some 3-opt. The improvement over the number of search nodes is not as much important as for the 3-opt method but the number of search nodes per second is higher. In practice, we have observed on average a 10% difference on the solving times between 3-opt and 2.5-opt.

Instances	(1) 3-opt not incremental time(s)	(2) 3-opt incremental time(s)	ratio (1) / (2) time
a280	6.0	3.6	1.7
ali535	*t.o.*	23,757.1	$>$ 3.6
brg180	4.0	1.1	3.6
d493	*t.o.*	7,715.3	$>$ 11.2
eil101	0.3	0.2	1.5
gil262	3,785.7	2,792.2	1.4
gr120	0.7	0.5	1.3
gr202	1.8	1.3	1.4
gr229	59.4	40.2	1.5
gr431	554.1	379.2	1.5
pcb442	*t.o.*	79,639.0	$>$ 1.1
pr136	15.7	8.9	1.8
rat195	19.4	15.4	1.3
rat99	0.3	0.2	1.5
si175	333.7	169.9	2.0
u159	0.8	0.4	2.0
mean	$>$ 16,498.9	7,157.8	

Table 3.7: Comparison of solving times for the non-incremental and the incremental version of the mandatory Hamiltonian path constraint.

In Table 3.7, we show the impact of using the incremental version of the mandatory Hamiltonian path constraint on the instances of Table 3.6 such that there is at least 30% of gain. Note that almost half of the instances are solved at least 30% faster with the incremental version of the algorithm. With the incremental version, the solving times of some instances such as rat195 are improved of 30% whereas for other instances such as si175 the solving times are improved by a factor of 2. In addition, some instances of this instance set reached the timeout with the non-incremental version such as ali535, d493 and pcb442 and it is on average at least 2.3 times faster

than the non-incremental one. Thus, the benefit of avoiding recalculations is interesting for this constraint due to the time complexity of the k-opt algorithm.

In Table 3.8, we are interested in the use of k-opt algorithms with k greater than 3. For the number of search nodes, we notice that the number of search nodes is comparable for 3-opt, 4-opt and 5-opt. For instance, kroB200 is solved with 10,045 search nodes for 3-opt, 10,779 search nodes for 4-opt and 12,707 search nodes for 5-opt. The main difference is that it is solved in 14.9s for 3-opt, 55.8s for 4-opt and 8,423s for 5-opt. Therefore, we notice that the use of 4-opt and 5-opt degrades the solving times compared to 2-opt and 3-opt. Indeed, for 4-opt we observe a loss of at least a factor almost equal to 3 in mean. For 5-opt, we observe a loss of at least a factor almost equal to 7 in mean with huge loss factor for some instances: 6,704 for a280, 12,722 for br180. Thus, the solving times and number of search nodes trade-off is bad when $k > 3$.

3.5.6 Conclusion

In this section, we introduced a new constraint based on the k-opt algorithm, named mandatory Hamiltonian path constraint, into to the TSP model in CP. We also introduced an incremental version of this constraint. Experiments have shown that the use of this constraint leads to an improvement of at least a factor of 3 in solving times. In addition, it shows that the use of 3-opt is well suited for our constraint. Moreover, we have been able to solve some instances that remains unsolved with the WCC in combination with the k-cutset constraint. The k-opt algorithm is embedded in most of the solving methods of the TSP and therefore now in the CP. In future work, we would like to study an extension of this constraint not only considering the mandatory Hamiltonian paths but the mandatory cutsets in the graph.

	2-opt		3-opt		4-opt		5-opt	
	time(s)	#sn	time(s)	#sn	time(s)	#sn	time(s)	#sn
a280	4.7	2,027	3.6	1,267	448.3	2,349	24,132.9	2,491
ali535	14,658.8	2,442,469	23,757.1	2,716,581	*t.o.*	*t.o.*	*t.o.*	*t.o.*
bier127	0.2	53	0.2	53	0.9	53	12.8	71
brg180	0.5	101	1.1	101	64.2	101	13,994.3	101
ch130	0.7	389	0.8	401	5.1	449	425.9	375
ch150	1.0	523	0.9	493	3.9	593	581.7	569
d198	6.3	3,451	6.6	2,791	80.1	2,945	15,659.4	3,415
d493	12,508.3	2,523,995	7,715.3	746,881	*t.o.*	*t.o.*	*t.o.*	*t.o.*
eil101	0.1	65	0.2	65	0.9	65	43.2	65
gil262	4,159.0	2,518,087	2,792.2	1,519,741	11,572.2	1,351,149	*t.o.*	*t.o.*
gr120	0.5	263	0.5	285	1.3	273	225.3	297
gr137	1.0	711	1.1	721	2.2	513	308.8	945
gr202	1.3	517	1.3	531	26.1	655	2,060.6	531
gr229	45.2	32,363	40.2	27,725	176.3	18,145	26,460.4	24,579
gr431	429.2	65,549	379.2	51,317	3,497.2	29,159	*t.o.*	*t.o.*
gr666	55,898.6	5,741,233	19,101.7	1,669,653	*t.o.*	*t.o.*	*t.o.*	*t.o.*
gr96	0.5	311	0.5	283	1.1	347	61.9	273
kroA100	0.7	557	0.7	425	1.6	499	9.7	591
kroA150	3.0	2,395	2.6	2,035	3.6	1,175	210.1	1,209
kroA200	58.3	35,821	70.6	39,305	112.1	43,041	2,840.0	55,619
kroB100	1.3	1,387	1.1	1,271	2.3	1,451	58.1	1,441
kroB150	10.7	8,745	10.8	8,481	20.4	7,113	747.6	8,985
kroB200	18.3	12,851	14.9	10,045	55.8	10,779	8,423.0	12,707
kroC100	0.5	305	0.6	305	1.9	319	4.6	285
kroD100	0.3	173	0.4	159	1.5	173	12.9	173
kroE100	0.9	889	0.9	793	2.0	713	18.6	795
lin318	5.6	1,711	4.7	1,341	14.9	1,737	10,004.7	1,743
pcb442	19,319.6	6,784,259	79,639.0	11,865,739	*t.o.*	*t.o.*	*t.o.*	*t.o.*
pr124	1.3	1,037	1.3	915	3.9	775	87.4	2,043
pr136	7.7	7,557	8.9	8,899	98.3	6,745	22,868.5	9,971
pr144	1.0	579	1.1	557	2.1	551	59.9	971
pr264	4.0	825	3.4	397	33.0	475	23,430.6	477
pr299	8,772.9	3,040,171	2,429.7	737,971	5,907.2	1,073,671	*t.o.*	*t.o.*
rat195	23.3	17,915	15.4	10,661	85.3	7,677	11,374.0	6,645
rat99	0.1	75	0.2	75	0.8	75	19.5	75
rd400	13,604.8	4,140,613	6,496.9	1,882,547	10,317.5	1,410,859	*t.o.*	*t.o.*
si175	168.8	196,943	169.9	163,917	1,389.8	149,541	*t.o.*	*t.o.*
st70	0.1	73	0.2	73	0.7	73	15.1	73
tsp225	76.2	44,567	30.1	16,129	81.6	15,339	4,410.7	14,465
u159	0.4	203	0.4	193	6.1	183	858.8	183
mean	3,244.9		3,567.7		$>$ 9,490.6		$>$23,675.5	

Table 3.8: Comparison of solving times for mandatory Hamiltonian path constraint with 2-opt, 3-opt, 4-opt and 5-opt.

3.6 One-Tree constraint

In this section, we will define a constraint based on the 2-cutsets and the 1-tree. As we stated in Section 2.2.1.3, the cost of any solution of the TSP is greater than or equal to the cost of a 1-tree. In addition, a strong property of the 1-tree is the following: if all nodes of the 1-tree have exactly two neighbors, then the 1-tree is a solution to the TSP. In some cases, the TSP can be decomposed into independent sub-problems. Therefore, we will use this property in order to define a filtering algorithm.

3.6.1 The constraint

Definition 3.6.1 (Minimum cost Hamiltonian path)**.** A Minimum cost Hamiltonian path $MHP(G, i, j)$ is the Hamiltonian Path P in G going from i to j such that the cost of P is minimized.

Definition 3.6.2 (k-vertex-connected graph)**.** A connected graph is k-vertex-connected if it remains connected whenever fewer than k nodes are removed.

Definition 3.6.3 (k-vertex-cutset)**.** A k-vertex-cutset of a connected graph is a subset of nodes S such that $|S| = k$ and removing S from the graph disconnects it.

Without loss of generality, we will consider that G is 2-vertex-connected.

In Proposition 3.6.1, we show that if there is a 2-vertex-cutset disconnecting the graph in more than 2 connected components, then there is no TSP solution in this graph. Without loss of generality, we will consider that each 2-vertex-cutset of the graph disconnect it in exactly two connected components.

Proposition 3.6.1. *Given the 2-vertex-cutset $\{i, j\} \in X^2$. If $G - \{i, j\}$ is disconnected in more than two connected components, then there is no TSP solution in G.*

Proof.

For each subset of nodes C in a graph, any TSP solution contains at least two edges with exactly one endpoint in C. Let k be the number of connected components of $G - \{i, j\}$. If we have k connected components in $G - \{i, j\}$, then we have at least $2k$ edges with one endpoint in C and the other in $\{i, j\}$. Thus, it means that we have $2k$ edges connected to $\{i, j\}$.

The degree constraint of the TSP enforces that each node has two adjacent edges in the solution. Thus, there are at most four edges connected to $\{i, j\}$ in any TSP solution. However, if $k > 2$, then $2k > 4$ and therefore if $G - \{i, j\}$ is disconnected in more than two connected components, then there is no TSP solution in G. □

In Proposition 3.6.2, we show that if a pair of nodes $\{i, j\}$ is a 2-vertex-cutset of G and (G', G'') its cut (both containing i and j), then the solutions set of the $TSP(G)$ is equivalent to the Cartesian product of the solutions set of $MHP(G', i, j)$ and the solutions set $MHP(G'', i, j)$. An example is depicted in Figure 3.13.

Proposition 3.6.2. *Given the 2-vertex-cutset $\{i, j\} \in X^2$ of the cut (G', G''). Then, the solutions set of both $TSP(G)$ and $MHP(G', i, j) \times MHP(G'', i, j)$ are equivalent.*

Proof.

By definition of a cutset, G' and G'' are independent and all the nodes of G' and G'' must be visited to obtain a solution to the TSP. In addition, each solution of the TSP is 2-vertex-connected. Since $\{i, j\}$ is a 2-vertex-cutset, each solution of the TSP must use i and j to go from G' to G''. Thus, each solution of $TSP(G)$ can be constructed by a combination of solutions of $MHP(G', i, j)$ and $MHP(G'', i, j)$.

Next, each solution of $TSP(G)$ necessarily contains a Hamiltonian path from i to j in G' and a Hamiltonian path from i to j in G''. Moreover, these two Hamiltonian paths are minimal cost paths. Otherwise, the cost of the TSP solution could be decreased, which is not possible since the cost of the TSP is minimized. Thus, each solution of $TSP(G)$ gives the solutions of $MHP(G', i, j)$ and of $MHP(G'', i, j)$.

Finally, given the 2-vertex-cutset $\{i, j\} \in X^2$ of the cut (G', G''), the solutions set of both $TSP(G)$ and $MHP(G', i, j) \times MHP(G'', i, j)$ are equivalent. □

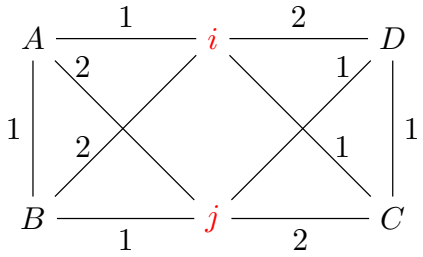


Figure 3.13: In this graph, we consider the 2-vertex-cutset $\{i, j\}$ and its cut (G', G'') such that the nodes set of G' is $\{i, j, A, B\}$ and the nodes set of G'' is $\{i, j, C, D\}$. The optimal solution of the TSP is $[i, A, B, j, D, C, i]$. The optimal solution of $MHP(G', i, j)$ is $[i, A, B, j]$ and the optimal solution of $MHP(G'', i, j)$ is $[j, D, C, i]$. Therefore, we can see that the solutions set of the $TSP(G)$ is equivalent to the Cartesian product of the solutions set of $MHP(G', i, j)$ and the solutions set $MHP(G'', i, j)$ for the a 2-vertex-cutset $\{i, j\}$ in G.

Moreover, we note that a Hamiltonian path from i to j is a spanning tree such that all nodes of the spanning tree have exactly two neighbors except i and j which have only one neighbor. Thus, an MST is a lower bound on a minimum cost Hamiltonian path.

Proposition 3.6.3. *Given $T(G)$ the minimum spanning tree of G. If each node of G has exactly two adjacent edges in $T(G)$ except for i and j having exactly one adjacent edge, then $T(G)$ is a solution of $MHP(G, i, j)$.*

Proof.

By definition, if each node of G has exactly two adjacent edges in $T(G)$ except for i and j having exactly one adjacent edge, then $T(G)$ is a Hamiltonian path going from i to j in G. In addition, since the total cost is minimized in a minimum spanning tree, then this Hamiltonian path has a minimum cost. Thus, $T(G)$ is a solution of $MHP(G, i, j)$. □

In addition, if the MST is a solution of MHP, then all the edges of the MST can safely become mandatory edges. In order to define a filtering algorithm, we define this properly in Proposition 3.6.4.

Definition 3.6.4 (Graphs intersection)**.** Given $G_1 = (X_1, U_1)$ and $G_2 = (X_2, U_2)$. We define $G_3 = G_1 \cap G_2$ such that $G_3 = (X_1 \cap X_2, U_1 \cap U_2)$.

Proposition 3.6.4. *Given $T(G)$ the minimum spanning tree of G, the 2-vertex-cutset $\{i, j\} \in X^2$ of the cut (G', G''). If $T(G) \cap G'$ (resp. $T(G) \cap G''$) is a Hamiltonian path in G' (resp. G''), then it is a solution of $MHP(G', i, j)$ (resp. $MHP(G'', i, j)$).*

Proof.

The 1-tree ensures a minimal coverage of G' and the existence of a single cycle. Since G' is independent of G'', if it has existed a cover of G' with a cost lower than $T(G) \cap G'$, then this cover would belong to $T(G)$ in order to reduce the global cost of $T(G)$. Thus, if $T(G) \cap G'$ is a Hamiltonian path in G', then it is a solution of $MHP(G', i, j)$. Note that the proof is the same for G''. □

For instance, in Figure 3.14 we show a graph and its minimum 1-tree. We notice that there is a Hamiltonian path in the component induced by the red circle. By Proposition 3.6.4, all the orange edges in the red circle can become mandatory.

In order to find the 2-vertex-cutsets, we can use a SPQR tree [Hopcroft and Tarjan, 1973, Gutwenger and Mutzel, 2000]. It is a linear time algorithm decomposing a graph in 3-vertex-connected components. However, the algorithm is not trivial and the trade-off filtering/solving times is not as good as we expected.

Therefore, we consider the 2-cutsets (*i.e.* cutsets of edges of size 2). Indeed, if a 3-vertex-connected component C is connected to the whole graph by a 2-cutset, then the nodes of C connected to the 2-cutset are a 2-vertex-cutset. Thus, 2-cutset is a special case of 2-vertex-cutset. We can then introduce Corollary 3.6.5.

Corollary 3.6.5. *Given $T(G)$ the minimum spanning tree of G, the 2-cutset of the cut (G', G'') and {i,j} the nodes belonging to both the 2-cutset and G' (resp. G''). If $T(G) \cap G'$ (resp. $T(G) \cap G''$) is a Hamiltonian path in G' (resp. G''), then it is a solution of $MHP(G', i, j)$ (resp. $MHP(G'', i, j)$).*

For instance, Figure 3.14 shows a case such that Corollary 3.6.5 can be applied if G' is the graph induced by the nodes in the red circle and the 2-cutset is the set of edges crossing the red circle.

In practice, the 2-cutsets are already computed in the k-cutset constraint. A first idea is to keep the 2-cutsets between the call of the k-cutset constraint and the call of this constraint. In a more intrusive but more efficient way, we can perform the filtering of this constraint while performing the k-cutset constraint. This approach results in a very efficient algorithm with a much better trade-off filtering/solving times.

The main advantage of this filtering algorithm is that it avoids the branching choices in areas where the 1-tree already "knew" the solution and therefore it reduces the search space. Moreover, it reduces the number of optional edges in the input graph. This leads to a slight improvement of the computation times of the side algorithms (1-tree computations, k-cutset constraint, mandatory Hamiltonian path constraint, ...).

One could now ask the following question: what if we have a k-cutset with $k > 2$ and k mandatory edges?

If k is odd, then the k-cutset constraint handles this (a failure is triggered). Without loss of generality, we assume that k is even.

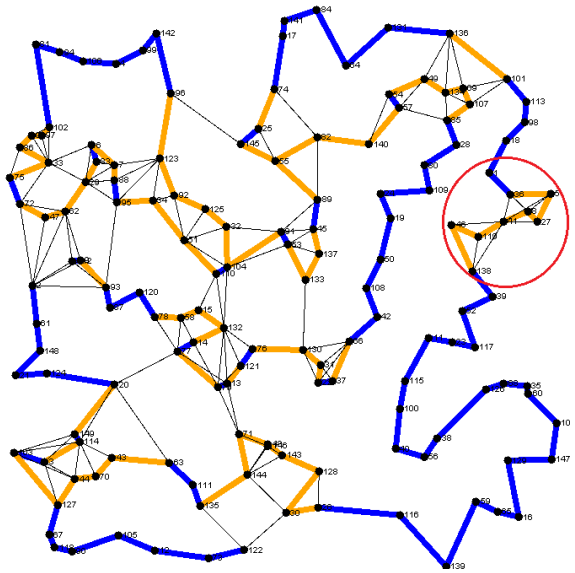


Figure 3.14: An example of a graph where the blue edges are mandatory, the dark edges are optional and the orange edges are optional edges belonging to the 1-tree.

First, for each cut (S, T) associated with the k-cutset K, a solution of the TSP goes through all nodes of S and T. If K contains k mandatory edges, then S and T are connected to the whole graph by $|K|$ connection points that can be split into exactly $|K|/2$ entry points and $|K|/2$ exit points. Thus, the optimal solution of the TSP is composed of $|K|/2$ disjoint paths in S and $|K|/2$ disjoint paths in T.

If $|K| = 2$, then the TSP solutions are composed of exactly $|K|/2 = 1$ path from one connection point to the other in S and in T. In this case, we have shown in Corollary 3.6.5 that if we find a Hamiltonian path of minimum cost going from one connection point to the other, then this path necessarily belongs to the optimal solution.

If $|K| > 2$, then the TSP solutions are composed of exactly $|K|/2$ disjoint paths in S and T. However, finding the $|K|/2$ disjoint paths of minimal cost covering the whole component is not enough. Indeed, the choice of entry points and exit points in a component rely on the choices of edges in the 1-tree outside this component. In Figure 3.15, we show two examples and we give the solution of the TSP in this graph. In OT_1, we notice that the coverage obtained by the 1-tree is not minimal in the cut represented by a red circle (3+3 > 2+2). In OT_2, we notice that even if the coverage is minimal (1+1 < 2+2), these edges do not belong to the global optimal solution. Indeed, the "good" edges are the edges whose cost is 2. However, constructing the 1-tree all the edges with a cost equal to 1 are added in OT_1 and OT_2. We then can see in both 1-trees that adding the "good" edges will close a cycle and therefore will never be added to the 1-tree. Thus, for $|K| > 2$, we cannot naively assign the paths of a cut if they all verify the degree constraint because we do not know which points are the entry and the exit points for every disjoint path. Splitting the problem in two parts is therefore non-trivial when $|K| > 2$. For this reason, the filtering algorithm of this constraint only consider the case of $|K| = 2$.

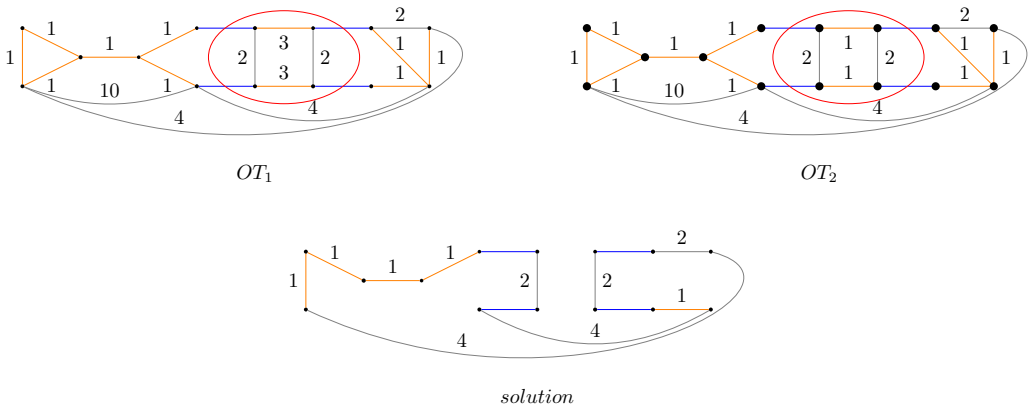


Figure 3.15: An example of a graph where the blue edges are mandatory, the gray edges are optional and the orange edges are optional edges belonging to the 1-tree. Note that OT_1 and OT_2 are two 1-trees such that the 1-tree edges in the red circle of both OT_1 and OT_2 are different. The bottom graph is an optimal solution of both OT_1 and OT_2. The choice of the 1-tree edges in the red circle does not depend on the cost of the edges but on the choices that we have made outside of the red circle. Thus, the 1-tree does not choose in any case the edges of cost 2 even if they belong to the TSP solution.

Definition 3.6.5 (One-tree constraint)**.** For each cut (S, T) associated with the k-cutset K such that $|K| = 2$, the one-tree constraint ensure that each TSP solution contains a single minimum cost Hamiltonian path in both S and T.

3.6.2 Experiments

In this section we will experimentally show that the one-tree constraint allows a slight reduction of the search space leading to a reduction of solving times. To do so, we use the configuration introduced in Section 1.1. Specifically, we always use WCC, the k-cutset constraint and the mandatory Hamiltonian path constraint with 3-opt, named "basic model".

In Table 3.9, we observe a slight improvement of the solving times for most instances (from 10% to 80%). In mean, the solving times are reduced of 5% and the number of search nodes are reduced of 1%. However, the geometric mean is reduced of 1% in solving time and the number of search nodes of 14%. Note that the geometric is less sensitive to huge difference between values. Obtaining such a little improvement is not unexpected here. Indeed, this constraint only enforces paths in order to avoid wrong branching choice and reduce a little bit the number of considered edges for the algorithms. This constraint is therefore not essential, but it cost almost nothing in computation time: it can be done in the k-cutset constraint and if it leads to a more "stable" search tree, we should use it. For instance, ali535 is solved in 23,757s with the basic model whereas (2) allows solving it in 13,014s. Inversely, pcb442 is solved in 79,639s with the basic model whereas it is solved in 91,138s with (2). Note that there is no timeout for Table 3.9 since only pcb442 reached it whereas it is solved just after the timeout of 86,400s. Note that the more we add constraints, the more pcb442 is hard to solve. We will discuss that instance in Section 3.8.

	Basic model (1)		One-tree constraint (2)		ratio (1)/(2)	
Instances	time(s)	#sn	time(s)	#sn	time	#sn
a280	3.6	1,267	8.1	2,351	0.4	0.5
ali535	23,757.1	2,716,581	13,013.9	1,339,925	1.8	2.0
bier127	0.2	53	0.3	53	0.7	1.0
brg180	1.1	101	1.2	101	0.9	1.0
ch130	0.8	401	0.7	383	1.1	1.0
ch150	0.9	493	0.8	303	1.1	1.6
d198	6.6	2,791	7.0	3,093	0.9	0.9
d493	7,715.3	746,881	4,989.9	488,439	1.5	1.5
eil101	0.2	65	0.2	65	1.0	1.0
gil262	2,792.2	1,519,741	2,042.2	1,040,943	1.4	1.5
gr120	0.5	285	0.7	279	0.7	1.0
gr137	1.1	721	1.0	513	1.1	1.4
gr202	1.3	531	1.2	383	1.1	1.4
gr229	40.2	27,725	41.2	26,565	1.0	1.0
gr431	379.2	51,317	520.6	60,537	0.7	0.8
gr666	19,101.7	1,669,653	14,719.6	1,106,125	1.3	1.5
gr96	0.5	283	0.5	227	1.0	1.2
kroA100	0.7	425	0.8	483	0.9	0.9
kroA150	2.6	2,035	1.9	1,269	1.4	1.6
kroA200	70.6	39,305	63.7	35,209	1.1	1.1
kroB100	1.1	1,271	1.0	1,045	1.1	1.2
kroB150	10.8	8,481	8.7	6,165	1.2	1.4
kroB200	14.9	10,045	15.1	9,381	1.0	1.1
kroC100	0.6	305	0.5	233	1.2	1.3
kroD100	0.4	159	0.3	149	1.3	1.1
kroE100	0.9	793	0.9	641	1.0	1.2
lin318	4.7	1,341	6.4	1,817	0.7	0.7
pcb442	79,639.0	11,865,739	91,137.7	14,852,219	0.9	0.8
pr124	1.3	915	1.2	827	1.1	1.1
pr136	8.9	8,899	8.9	7,229	1.0	1.2
pr144	1.1	557	1.0	345	1.1	1.6
pr264	3.4	397	3.8	575	0.9	0.7
pr299	2,429.7	737,971	1,596.5	489,039	1.5	1.5
rat195	15.4	10,661	17.9	11,601	0.9	0.9
rat99	0.2	75	0.2	61	1.0	1.2
rd400	6,496.9	1,882,547	6,721.0	1,727,475	1.0	1.1
si175	169.9	163,917	100.7	92,573	1.7	1.8
st70	0.2	73	0.2	65	1.0	1.1
tsp225	30.1	16,129	23.5	11,557	1.3	1.4
u159	0.4	193	0.4	133	1.0	1.5
mean	3567.7	537278.1	3376.5	533009.4		
geo mean	10.5	4208.6	10.0	3621.8		

Table 3.9: General results showing the results while integrating the one-tree constraint to the basic model.

3.6.3 Conclusion

In this section, we introduced a new constraint based on the lower bound obtained by the 1-tree. It uses the fact that the TSP can sometimes be decomposed into independent sub-problems. Experiments have shown that it leads to a slight improvement by avoiding wrong branching choice.

3.7 Lagrangian Relaxation

In Section 3.2, we introduced the TSP model in CP that is a combination of the Held and Karp Lagrangian relaxation (LR) and some filtering algorithms (FAs).

For any set of multipliers μ, the optimal value of the LR of a problem P is a lower bound of the optimal value of P, then it can be used to remove some values of the variables. Consider UB, an upper bound of the optimal solution of P (for example any solution of P, therefore not necessarily optimal), and $x = a$ an assignment. If for $x = a$ the optimal value of the LR of P is greater than UB, then a can be removed from $D(x)$ since we know that $x = a$ does not belong to the optimal solution. From this idea, the CP-based Lagrangian relaxation has been introduced [Sellmann and Fahle, 2003] and successfully used to solve many problems [Khemmoudj et al., 2005, Menana, 2011, Fontaine et al., 2014, Bergman et al., 2015, Cambazard and Fages, 2015, Demassey, 2017]. It consists in modeling the problem so that one or more cost-based FAs can be used on the easy part of the problem. Difficult constraints are moved to the objective function and the cost-based FAs are used when looking for good multipliers.

Sellmann made two important observations about the relationship between the LR and FAs [Sellmann, 2004]:

- Suboptimal multipliers can be more efficient for filtering than the optimal multipliers.
- It is not clear whether FAs should actually take place during the optimization of the Lagrangian multipliers, because the standard approach for the optimization of the multipliers are not guaranteed to be robust enough to enable a change (*i.e.* the removal of a value) of the underlying sub-problem during the optimization.

These observations show the complexity of the interactions between FAs and multipliers, which have important consequences, such as losing the monotonicity so dear to CP*.

In addition, it is important to note that the CP-based LR is usually associated with a branch-and-bound algorithm. We therefore have no reason to seek to converge the LR towards the optimum. The lower bound it provides is sufficient for our purpose (*i.e.* having the most effective FAs) and we will obtain the optimal solution thanks to the search algorithm.

From these considerations, we can formulate the problem of the interaction between LR and FAs by the following question: for which set of multipliers should FAs be called and how do we get them?

Most of the articles in the literature using CP-based LR do not address this issue and it is by reading the source code of the programs that we discover precisely when FAs are called. Some authors (L-M Rousseau and X. Lorca) have confirmed that the call conditions were determined experimentally after numerous tests.

*Normally, in CP, when F_2, a FA, is added to F_1, another FA, all values eliminated by F_1 are also eliminated by the combination of F_1 and F_2.

In this section, we study some of the interactions between FAs and LR and introduce a method determining when FAs should be called.

First, we use a subgradient optimization algorithm, because it gives us access to suboptimal multipliers that can be quickly computed. The issue of the slow convergence of this type of algorithm does not arise in our case since we also use a search procedure.

By doing so, the problem that needs to be solved becomes: when are the FAs called in the subgradient algorithm?

The subgradient algorithms used in CP-based LR are most often variants of Beasley's algorithm [Beasley, 1993] introduced in Algorithm 1. Conceptually, it proceeds by successive iterations for different calculation accuracy, named agility. At each iteration the agility is divided by a power of 2. For a given agility value, the subgradient algorithm iteratively adjust the Lagrangian multipliers to find values that improve the lower bound. For a given agility value, the number of internal iterations, which we call scope, is the unknown we are looking for.

Thus, the problem becomes: which scope value leads to a set of multipliers making the filtering efficient?

To answer this question, we suggest studying the variation in the value of the LR objective function.

3.7.1 CP-based Lagrangian relaxation

According to Sellmann [Sellmann, 2004], CP-based LR consists in the following procedure: Assuming we are given a linear optimization problem that consists in the conjunction of two constraint families A and B for which an efficient filtering algorithm $prop(B)$ is known, we try to optimize Lagrangian multipliers for A and use $prop(B)$ for filtering in each Lagrangian sub-problem $LR(P, \mu)$.

It is not necessary for constraints A or B to be linear (something that is not imposed in CP). We need to ensure that the relaxation we calculate for any multiplier set is a relaxation of P. So, we just need to make sure that $prop(B)$ remains valid when the objective becomes that of the LR.

Sellmann defined a particular consistency based on the continuous relaxation of P, but it does not matter in this thesis. He also defined the following property:

Proposition 3.7.1. *Suboptimal multipliers can be more efficient for filtering than the optimal multipliers.*

This property is explained by the fact that a value $x = a$ can be removed when the optimal value of $P \wedge (x = a)$ is greater than UB, a given upper bound. By considering the Lagrangian relaxation we consider the problem $LR(P, \mu)$ and not $LR(P \wedge (x = a), \mu)$. Thus, the best multipliers for $LR(P, \mu)$ are not necessarily the best for $LR(P \wedge (x = a), \mu)$.

In CP, it is also possible to express the violation of the constraint in different ways, we can also decide not to measure the distance to the violation [Fontaine et al., 2014]. Since we only relax the degree constraint that is some equality constraints, we will not detail it.

Finally, very recent works of Boudreault and Quimper improved the CP-based LR applied to the TSP [Boudreault and Quimper, 2021]. It suggests to temporarily change the Lagrangian multipliers in order to filter specific values. Note that we did not try this method since it was published almost at the same time of this thesis.

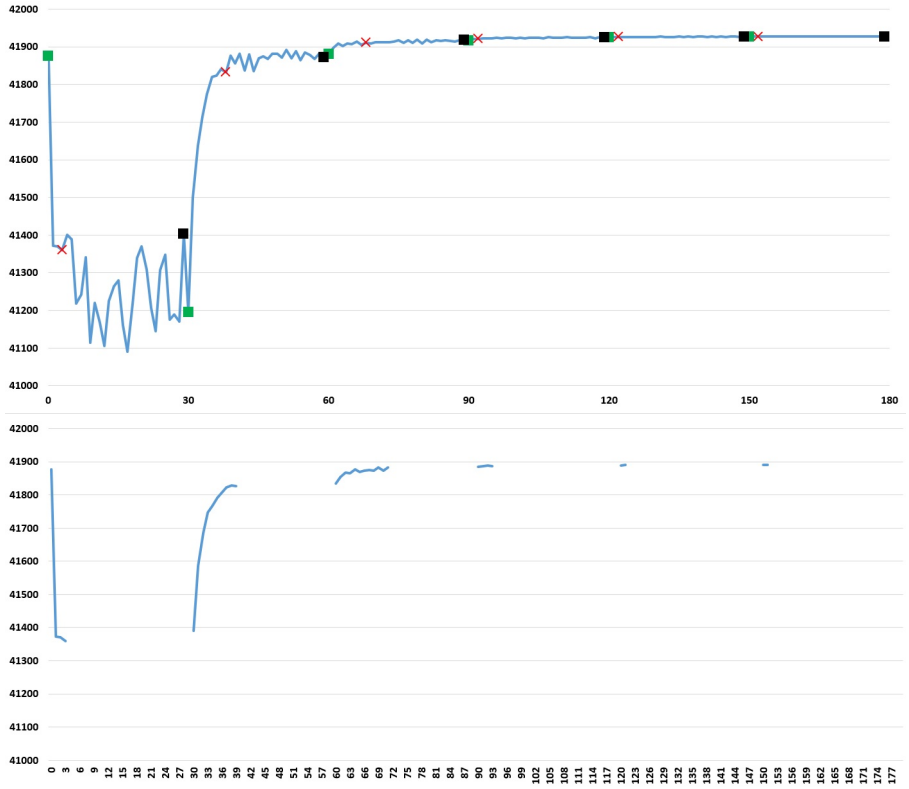


Figure 3.16: Evolution of the LR optimal value (on the y-axis) according to the $scope$ (on the x-axis). (top graph) Beasley's algorithm with $\#scope = 30$. A segment between green and dark squares corresponds to one agility value. Computations after red crosses are identified as useless. (bottom graph) Scope sizing subgradient algorithm. Computations are stopped at red crosses of the top graph.

3.7.2 Scope Sizing Subgradient Algorithm

In order to find good scope values for Algorithm 1, we suggest observing the optimal values computed by Algorithm 1 for some "classical" scope values. For instance, the subgradient algorithm (FLR) used by Fages *et al.* [Fages et al., 2016] in their experiments corresponds to the values of the parameters $\#agility = 5$ and $\#scope = 30$ of Algorithm 1 and makes the agility slightly different since it uses the following update formula: $\pi \leftarrow \pi/\beta$; $\beta \leftarrow \beta/2$ with $\beta = 1/2$ at initialization. It should also be noted that Fages et al. repeat the call to the algorithm as long as the lower bound of the 1-tree is increased and the FAs are triggered for each agility value.

In our experiments, we notice that $\#agility = 6$ tends to give slightly better results, thus we can define a first configuration of Algorithm 1 which is $\#agility = 6$ and $\#scope = 30$.

Figure 3.16 (top) is quite representative of what is frequently observed, namely:

- a strong variation followed by oscillations (the first two agility values).
- a weak variation followed by oscillations (third agility value).
- almost no variation (the last three agility values).

We have also observed weak or strong variations without oscillations.

For FAs to be effective, successive FAs must be called with relatively large variations, otherwise it is unlikely that successive filtering will remove more values than the previous ones. Moreover, in the particular case of the WCC, all the FAs integrated in the LR are based on the objective value. The orientation of the variations should not play a role since we are mainly looking for various multipliers. It is therefore certainly interesting to trigger the FAs after a strong variation in the objective value.

What about oscillations? They do not really provide any information. The multipliers are changed but very slightly between two oscillations. Nor do they provide much in terms of boundaries. We therefore suggest avoiding them as much as possible.

There is still the case of the absence of variations or very small variations (*i.e.* stagnation). It is in our interest to stop the calculations as soon as possible because the multipliers change very little.

We therefore propose to proceed as follows: as soon as we measure an oscillation or stagnation in the objective value, we stop changing the scope value.

We can then decide to call the FAs between the agility rounds or not. We will show in Section 3.7.3 that it is more interesting in terms of computation time to call the FAs after all the agility rounds.

Figure 3.16 represents our choice using red crosses. Indeed, the red crosses mark the end of the search for new multipliers. If we implement our proposal then we get the results of the bottom graphic of Figure 3.16 that we made coincide with the top graphic to show the difference. We can see that there is little difference in the calculated objective values in the end. There is a slight decrease for many of the avoided calculations. It can be explained by the fact that we stop earlier the computations and therefore we do not obtain exactly the same sub-problems, and therefore the LR convergence is impacted.

Algorithm 10 is a possible implementation of the Scope Sizing Subgradient Algorithm (SSSA). It is a direct adaptation of Beasley's algorithm (Algorithm 1). Note that the number of agility values that are considered is 6. We tested different values but we did not observe enough changes to justify the introduction of an additional parameter. The maximum value of scope is 12 because we almost always observe oscillations or stagnation for larger values. The stop conditions are not tested for all scope values but only one time out of two (internal loop of the q variable). This allows us to detect a large part of the oscillations. We measure for two iterations the variations and if the sum of these two variations does not deviate enough from the value at the beginning of the two iterations, then we no longer change the scope value. In this case, either an absence of variations or two variations of the same amplitude in opposite directions (*i.e.* an oscillation) will be detected. We empirically defined the minimum deviation as 1% of the difference between the current upper bound and the current objective value.

3.7.3 Experiments

In the initial published paper [Isoart and Régin, 2020a], we performed the experiments with the WCC in conjunction with the k-cutset constraint. In this thesis, we perform the experiments with the up-to-date TSP model in CP, that is the WCC in conjunction with the k-cutset constraint [Isoart and Régin, 2021b], the k-opt constraint [Isoart and Régin, 2021a] and the one-tree constraint.

Algorithm 10: Scope Sizing Subgradient Algorithm.

SSSA *(P,Z_{ub},μ)*
Input: A problem P, an upper bound Z_{ub} of P and a set of multipliers μ.
Output: A set of multipliers μ^{k+1} and x^k the optimal solution of LR(P,μ^k).
$\pi \leftarrow 2$; // Subgradient agility
$k \leftarrow 0$;
$\mu^0 \leftarrow \mu$; // Start with the current values of multipliers
foreach $iterAgility = 1 \dots 6$ **do**
  $scope \leftarrow 0$;
  while $scope < 12$ **do**
    $mean \leftarrow 0$;
    $d \leftarrow Z_{ub} - Z^k$; // Distance to UB
    foreach $q = 1..2$ **do**
      $prevBound \leftarrow Z^k$;
      $(\mu^{k+1}, x^k, Z^k) \leftarrow$ SOLVELR(P,Z_{ub}, π, μ^k, k) ;
      // Return the optimal solution of P if reached
      if $Z^k = Z_{ub}$ **then return** (μ^{k+1}, x^k);
      $mean \leftarrow mean + Z^k - prevBound$;
      $k \leftarrow k + 1$;
      $scope \leftarrow scope + 1$;
    if $|mean/2| \leq 0.01d$ **then** break;
  if *solver failed* **then return** *(nil,nil)*;
  $\pi \leftarrow \pi/2$;
// trigger the FAs
RUNPROPAGATION($P, x^{k-1}, Z_{ub}, \mu^k$) ;
return (μ^{k+1}, x^k)

In Table 3.10, we give the mean and geometric mean of the solving times and the number of search nodes for different scope values in Algorithm 1. Note that the scope value given in the first column is always the same for all agility values and therefore there is no attempt to prevent oscillations in Algorithm 1.

The results show that it is necessary to find a trade-off in the mean time and number of search nodes. Filtering with refined boundaries (*i.e.* with the largest scope) reduces the number of search nodes but increase the solving times. It is also important to note that there is no monotonicity. For instance, scope equal to 24 leads to fewer search nodes than 22 and 26.

For each instance, Table 3.11 shows solving times for the best scope, the worst scope and the best mean scope (scope=14) of Table 3.10. We observe that the best scope per instance changes quite often. It can be a high scope value as 30 for brg180 or a low scope value as 2 for kroE100. The best mean scope is also clearly dependent on the set of instances considered. For this instance set, the best scope is 14, but it is easy to build an instance set whose best mean scope value is not 14. The instances brg180, ch130, gr137, kroB200 form such a set since scope=30 is the best scope for each of them. In addition, by choosing scope=14, the solving times are in mean 34% slower than the best scope per instance and in geometric mean 62% slower than the best scope per

scope	mean		geo mean	
	time	search nodes	time	search nodes
2	30.7	22,656.4	5.4	2,105.7
4	20.8	13,098.0	4.5	1,332.6
6	18.9	8,704.1	4.6	1,028.9
8	19.5	9,051.1	4.6	921.8
10	20.4	8,911.1	5.1	943.2
12	18.7	7,866.7	4.5	868.3
14	17.5	7,264.4	4.7	842.0
16	22.9	8,612.8	3.4	800.0
18	21.7	7,003.7	4.0	780.4
20	28.9	8,941.8	4.6	767.0
22	29.8	9,236.4	5.3	786.0
24	22.9	6,096.5	5.1	694.1
26	25.2	6,353.8	5.5	760.7
28	23.0	5,988.9	5.3	686.5
30	21.7	6,294.6	3.8	715.1
50	34.2	6,786.9	4.9	731.7
100	51.5	5,715.6	6.7	684.1

Table 3.10: Comparison of solving times (in s) and the number of search nodes between scope values for Algorithm 1.

instance. However, in order to find the best mean scope, we have to solve all the instances for a given instance set. The w/b column indicates the time ratio between the worst and best scope that is very often beyond the factor 2. All these observations show that it is not easy to determine *a priori* a good scope value and that a bad scope value can have a strong impact on solving times.

In Table 3.12, we give the mean and geometric mean of the solving times and the number of search nodes for different scope values in Algorithm 10. We observe that the results are much more robust than the ones of Table 3.10. The main reason is that in Algorithm 10 the scope value is dynamic: if an oscillation or stagnation in the objective value is detected, then we go to the next agility round. Thus, even for high scope values such as scope=100, we observe a variation of only 51% with the best mean scope value (scope=12) whereas in Table 3.11 we observe a variation of 294%. In addition, scope values higher than 10 achieve similar results and increasing the scope value beyond 12 does not reduce the number of search nodes.

For each instance, Table 3.13 shows solving times for the best scope, the worst scope and the best mean scope (scope=12) of Table 3.12. We observe that the best scope per instance is always greater or equal than 12. It can be explained by the fact that a small scope value is needed when the objective value tends to oscillate or stagnate: Algorithm 10 prevents this. On the other hand, a scope value smaller than 12 does not give the algorithm enough flexibility. Indeed, if from one call to the other not enough iterations in the Lagrangian relaxation have been performed, then the obtained sub-problem can be very similar to the initial one and therefore the FAs does not filter much. We also notice that for Algorithm 10 with scope=12, we obtain solving times only 20% (resp. 33%) slower than the best scope per instance on mean (resp. geometric mean). We recall that we obtained in mean 34% and in geometric mean 62% for Table 3.11. In addition,

Instance	best time		worst time		ratio w/b	scope=14
	scope	time	scope	time	time	time
a280	16	7.8	2	75.9	9.8	11.2
bier127	16	0.4	2	1.0	2.6	0.8
brg180	30	1.3	4	2.8	2.1	2.3
ch130	30	1.3	26	2.5	1.9	1.8
ch150	20	1.9	14	4.1	2.2	4.1
d198	8	11.4	26	24.3	2.1	15.6
eil101	4	0.3	28	0.8	2.4	0.7
gr120	16	0.5	10	1.8	3.4	1.0
gr137	30	1.4	2	3.0	2.1	2.6
gr202	16	1.9	24	3.3	1.7	3.1
gr229	14	54.1	20	112.2	2.1	54.1
gr96	16	0.6	10	2.0	3.2	1.4
kroA100	16	0.9	28	3.6	3.8	1.9
kroA150	2	4.2	26	8.0	1.9	5.1
kroA200	24	46.8	2	186.5	4.0	111.0
kroB100	16	1.6	26	4.9	3.1	3.5
kroB150	4	13.9	26	51.9	3.7	20.8
kroB200	30	20.3	28	47.7	2.3	21.8
kroC100	16	0.9	26	1.8	2.0	1.5
kroD100	20	0.7	4	1.3	1.8	1.0
kroE100	2	1.5	28	3.6	2.4	1.9
lin318	20	7.1	24	20.4	2.9	14.8
pr124	4	1.9	24	5.4	2.8	3.8
pr136	18	12.7	2	22.4	1.8	15.5
pr144	30	1.3	26	2.6	2.0	2.2
pr264	16	5.6	2	16.3	2.9	7.8
rat195	16	22.5	22	54.3	2.4	33.1
rat99	16	0.2	22	0.6	2.5	0.5
si175	6	159.7	22	410.1	2.6	179.6
st70	18	0.2	24	0.6	2.5	0.5
tsp225	14	32.8	24	88.6	2.7	32.8
u159	16	0.7	26	1.2	1.8	0.8
mean		13.1		36.4		17.5
geo mean		2.9		7.5		4.7

Table 3.11: Comparison of solving times (in s) between best, worst and best mean scope values for Algorithm 1.

	mean		geo mean	
scope	time	search nodes	time	search nodes
2	31.1	22,656.4	5.3	2,105.7
4	23.4	16,569.4	4.4	1,244.3
6	17.6	8,932.4	4.1	1,019.1
8	19.6	11,176.5	4.3	948.3
10	15.7	7,543.3	4.4	910.0
12	10.3	6,739.8	2.8	904.1
14	14.8	9,648.5	3.8	831.3
16	16.4	7,390.6	3.1	872.1
18	18.1	8,702.4	3.6	874.7
20	16.9	6,942.1	3.8	868.4
22	17.2	7,800.9	4.0	880.1
24	18.4	8,424.4	4.2	906.5
26	16.3	7,365.4	4.0	831.2
28	16.7	8,284.4	4.3	923.4
30	11.5	7,013.2	2.7	836.2
50	12.3	6,810.1	2.8	825.7
100	15.6	9,559.1	2.9	868.9

Table 3.12: Comparison of solving times (in s) and the number of search nodes between scope values for Algorithm 10.

Algorithm 10 with scope=12 allows us to obtain a better mean and geometric mean than the best scope value per instance in Table 3.11.

In Table 3.14, we compare triggering the FAs for each agility round and triggering the FAs at the end of the optimization phase. We can see that for all scope values > 2, triggering the FAs at the end of the optimization phase is faster than triggering the FAs for each agility round. In addition, we can see that the number of search nodes for scope > 10 is quite close. Thus, the overhead generated by the FAs calls for each agility round does not allow us to obtain a sufficient gain in the number search nodes to reduce the solving times. This is mainly due to the fact that we obtain similar filtering by calling the FAs for each agility round and at the end of the optimization phase. We therefore decide not to call the FAs for each agility rounds but at the end of optimization phase.

Table 3.15 shows the overall improvement. The ratio column shows for each instance the improvement factor obtained by using Algorithm 10. We note that most of the instances have improved solving times by huge factor (up to 23.8). SSSA improves the solving times by a factor 9.1 in mean and 3 in geometric mean. In addition, we notice that the number of search nodes in mean and geometric mean are similar (8,060.6 vs 6,738.8 and 719 vs 904.1). It is a great result because it means that SSSA avoids many useless computations and therefore SSSA allows us to obtain a significant improvement factor of solving times.

Instance	best time		worst time		ratio w/b	scope=12
	scope	time	scope	time	time	time
a280	30	4.5	2	80.7	17.8	8.4
bier127	30	0.3	2	1.2	4.4	0.4
brg180	30	1.3	4	2.9	2.2	1.7
ch130	12	1.3	22	2.0	1.6	1.3
ch150	12	1.3	14	3.2	2.4	1.3
d198	12	8.0	2	22.2	2.8	8.0
eil101	12	0.3	14	0.6	2.2	0.3
gr120	20	0.5	28	1.4	3.0	1.0
gr137	16	1.4	2	3.0	2.1	1.5
gr202	12	1.8	2	3.0	1.7	1.8
gr229	12	41.3	2	109.0	2.6	41.3
gr96	16	0.6	8	1.6	2.7	0.8
kroA100	16	1.0	14	1.9	1.9	1.3
kroA150	12	2.9	24	6.6	2.3	2.9
kroA200	14	21.9	2	182.4	8.3	63.2
kroB100	12	1.7	26	3.5	2.1	1.7
kroB150	12	9.2	24	30.6	3.3	9.2
kroB200	30	12.3	2	43.8	3.6	14.7
kroC100	12	0.8	10	2.0	2.4	0.8
kroD100	30	0.5	2	1.2	2.3	0.6
kroE100	16	1.4	10	2.5	1.8	1.4
lin318	16	5.1	28	11.9	2.3	7.4
pr124	16	1.5	18	4.4	3.0	2.0
pr136	12	10.0	2	22.7	2.3	10.0
pr144	30	1.3	6	2.3	1.8	1.4
pr264	30	4.1	2	16.1	3.9	4.7
rat195	18	14.3	2	50.0	3.5	17.4
rat99	30	0.2	4	0.6	3.0	0.3
si175	12	100.0	4	335.0	3.4	100.0
st70	30	0.2	10	0.5	2.8	0.3
tsp225	12	23.5	2	88.0	3.7	23.5
u159	12	0.6	10	1.4	2.4	0.6
mean		8.6		32.4		10.3
geo mean		2.3		6.5		2.8

Table 3.13: Comparison of solving times (in s) between best, worst and best mean scope values for Algorithm 10.

	no FAs between agility rounds				FAs between agility rounds			
scope	mean		geo mean		mean		geo mean	
	time	search nodes	time	search nodes	time	search nodes	time	search nodes
2	31.1	22,656.4	5.3	2,105.7	27.4	14,901.1	4.9	1,501.9
4	23.4	16,569.4	4.4	1,244.3	21.8	10,336.8	4.3	1,073.6
6	17.6	8,932.4	4.1	1,019.1	22.1	10,371.1	4.7	1,011.7
8	19.6	11,176.5	4.3	948.3	21.2	9,455.6	4.7	920.0
10	15.7	7,543.3	4.4	910.0	21.8	10,206.7	4.9	939.5
12	10.3	6,739.8	2.8	904.1	18.7	7,788.9	5.1	935.1
14	14.8	9,648.5	3.8	831.3	19.9	10,526.1	4.8	925.9
16	16.4	7,390.6	3.1	872.1	18.6	6,684.4	3.4	856.7
18	18.1	8,702.4	3.6	874.7	18.4	6,637.7	3.8	814.4
20	16.9	6,942.1	3.8	868.4	20.3	7,501.3	4.3	865.7
22	17.2	7,800.9	4.0	880.1	23.4	9,153.9	4.7	841.7
24	18.4	8,424.4	4.2	906.5	18.7	6,039.6	4.7	808.7
26	16.3	7,365.4	4.0	831.2	19.5	6,677.3	5.1	813.9
28	16.7	8,284.4	4.3	923.4	18.4	6,701.3	5.0	874.9
30	11.5	7,013.2	2.7	836.2	17.3	7,309.3	4.9	821.1

Table 3.14: Comparison of solving times (in s) and the number of search nodes between scope values for Algorithm 10 when the FAs are triggered between the agility rounds or not.

3.7.4 Conclusion

Finally, we have introduced SSSA, a Scope Sizing Subgradient Algorithm, which is an adaptive algorithm for the CP-based Lagrangian relaxation. It stops local multiplier optimization when the objective value no longer varies or oscillates and trigger the cost-based FAs only at the end of the optimization phase. In addition, experiments suggest using scope=12 for the TSP. The experimental results we presented show the interest of our approach. The solving times of the best CP model known so far for solving TSP are improved by a factor 9.1 in mean and 3 in geometric mean.

We believe that SSSA is taking a first step towards a better understanding of the interactions between FAs and LRs and that others will follow.

We hope that it will also allow similar results to be obtained for other problems than the TSP that may lead to a general improvement of the CP-based LR.

Instance	FLR		SSSA scope=12		ratio	
	time	search nodes	time	search nodes	time	search nodes
a280	36.5	1,873	8.4	2,351	4.3	0.8
bier127	0.4	39	0.4	53	1.0	0.7
brg180	0.7	87	1.7	101	0.4	0.9
ch130	3.2	361	1.3	383	2.6	0.9
ch150	3.7	355	1.3	303	2.9	1.2
d198	48.6	3,057	8.0	3,093	6.0	1.0
eil101	0.3	57	0.3	65	1.1	0.9
gr120	0.8	57	1.0	279	0.8	0.2
gr137	7.5	539	1.5	513	4.8	1.1
gr202	4.3	331	1.8	383	2.4	0.9
gr229	367.7	17,171	41.3	26,565	8.9	0.6
gr96	1.5	137	0.8	227	1.9	0.6
kroA100	2.9	417	1.3	483	2.3	0.9
kroA150	19.1	1,619	2.9	1,269	6.6	1.3
kroA200	362.3	14,527	63.2	35,209	5.7	0.4
kroB100	3.6	567	1.7	1,045	2.2	0.5
kroB150	90.2	7,041	9.2	6,165	9.8	1.1
kroB200	98.2	6,513	14.7	9,381	6.7	0.7
kroC100	2.4	313	0.8	233	3.0	1.3
kroD100	1.1	163	0.6	149	1.8	1.1
kroE100	3.2	439	1.4	641	2.3	0.7
lin318	75.9	1,723	7.4	1,817	10.3	0.9
pr124	7.5	545	2.0	827	3.8	0.7
pr136	31.4	3,977	10.0	7,229	3.1	0.6
pr144	4.4	247	1.4	345	3.1	0.7
pr264	3.4	97	4.7	575	0.7	0.2
rat195	132.6	10,469	17.4	11,601	7.6	0.9
rat99	0.3	55	0.3	61	1.2	0.9
si175	1,137.7	160,297	100.0	92,573	11.4	1.7
st70	0.4	77	0.3	65	1.4	1.2
tsp225	557.9	25,683	23.5	11,557	23.8	2.2
u159	0.9	97	0.6	133	1.5	0.7
mean	94.1	8,091.6	10.3	6,739.8		
geo mean	8.4	722.7	2.8	904.1		

Table 3.15: Overall improvement. Comparison of solving times (in s) and the number of search nodes between FLR and Algorithm 10 with scope=12.

3.8 General results

In this section, we will discuss some general results from before the thesis and now. For the record, the instances we consider from the TSPLib are: cities in one or more countries in 2D (prefixed by bier, ch, e, gil, kro, rd and st), drilling problems in 2D (prefixed by a, d, lin, pcb, pr and u), rattled grid in 2D (prefixed by rat) and 3D cities (prefixed by ali, brg, gr). In Section A, we display all the nodes set for each instance we considered along this thesis except the 3D cities since displaying them in 2D is not relevant. Note that all the graphs are initially complete. In order to be as general as possible, we did not use some Euclidean property for the graph in Euclidean norm: we did not make any assumption on the instances we consider.

3.8.1 Analysis of the instances

Some problems are structurally more difficult than others. We can see in Section A that the drilling problems are very particular. Indeed, they are often constructed with some clusters of nodes that are very close to each other while the clusters can be very far away. Therefore, the cost-based filtering algorithms do not filter that much in a cluster since introducing an edge in a cluster often does not lead to a great increase of the lower bound. Another issue is that there are some pathological cases. For instance, take the left column of Figure A.19. In this column, we observe that there are many "squares" of points. In practice, the 1-tree handle this really badly since all the sides of the square have equivalent length and then the Lagrangian relaxation only transfer the costs from a node to another. In addition, it is really hard to filter the diagonal edge since adding one diagonal to the lower bound usually did not increase the lower bound that much. Other solving methods such as the MIP handle that much better since they can add constraints if such a pathological case occurs. In our case, we can branch on this zone or try to filter them. Branching on them is a bad idea since it can lead to a combinatorial enumeration of the squares and filtering them are really hard just as we said. We could try to fix this issue with the Lagrangian relaxation by integrating some constraints in the objective function. Since it is often inequality constraints, it introduces a gap in the objective function and therefore the filtering based on the costs become inefficient.

3.8.2 Large instances

The complexity of most of the algorithms we used is related to the number of edges. The number of edges in a complete graph grows quadratically with respect to the nodes. Then, a complete graph with 100 nodes has 4,950 edges whereas a complete graph with 1,000 nodes has 499,500 edges. We mainly focused on the graph from 100 to 1,000 nodes. However, trying to solve larger graphs, we encountered problems containing 750,000 edges after the first propagation step. For instance, we need to sort all the edges for Kruskal's algorithm and then create the spanning tree at each iteration of the Lagrangian relaxation. In order to obtain good results, systematic algorithms should no longer be applied. Incremental algorithms, such as the one for the mandatory Hamiltonian path constraint, should be developed.

3.8.3 Experiments

For the sake of clarity, we will name "best model" the WCC combined with the k-cutset constraint, the mandatory Hamiltonian path constraint and the one-tree constraint. In addition, we extended

the timeout to 100,000s in order to have all the experiments in the scope of the best model (*i.e.* for pcb442 which has a solving time that increases strongly when filtering is added).

In Table 3.16, we compare the WCC and the best model. Note that we used the search strategy LCFirst minDeltaDeg for both (1) and (2). First, we notice that 7 over 40 instances remain unsolved in 100,000s with the WCC whereas with our best model we solve all the instances in less than 100,000s. Indeed, ali535 is solved in 13,014s, d493 is solved in 4,990s, gr431 is solved in 521s, gr666 is solved in 14,720s pcb442 is solved in 91,138s, pr299 is solved in 1,596s and rd400 is solved in 6,721s. In addition, all the instances except the one having a solving time in less than 0.5s have a greatly reduced solving time with our best model: a factor 3.7 for ch130, 41.4 for kroA200, ... Thus, we obtain a mean solving time of at least 17,797s for this instance set whereas our best model obtains a mean solving time of 3,377. That is at least 5.3 times faster.

A lot of work has been done on the implementation of the whole model. Therefore, in Table 3.17, we compare the state-of-the-art performances obtained by Fages et al. [Fages et al., 2016], that is the WCC implemented in Choco-3.1.0 in Java, with our implementation of the WCC. Note that the results for Choco are those they obtained in their paper on a Macbook Pro with a 6-core Intel Xeon at 2.93 Ghz running MacOS 10.6.8 with Java 1.7. In addition, they set a timeout of 30,000s. Therefore, we perform the experiments for Table 3.17 and Table 3.18 with a timeout of 30,000s. In addition to the different implementation, note that we use SSSA [Isoart and Régin, 2020a] which is helping us to largely improve our solving times. We observe that a lot of instances have a huge improvement factor for the same model but a different implementation: a280 is improved by a factor 26.4, ch130 by 27.7, gr229 by 36.6, ... It leads to a mean solving time of at least 4,363s for the WCC of Choco and a mean solving time of at least 2,113s for our implementation of the WCC. For the geometric means, they obtain at least 82.4 whereas we obtain at least 9.4. Here, the geometric is more relevant since many instances are improved by a huge factor (sometimes more than an order of magnitude) and therefore there is a lot of variation of the values. In addition, four instances reached the timeout for Choco whereas only two instances reached the timeout with our implementation.

In Table 3.18, we compare the implementation of the WCC in Choco with our best model. First, we observe that only one instance remains unsolved (ts225) with our model in the timeout of Choco (30,000s). Next, we notice that most of the instances are solved with huge improvement factors going from 1.7 (pr107) to 471 (kroA200). The improvement for the mean and the geometric mean is quite interesting: at least 4,363 for the mean of Choco and 82 for the geometric mean of Choco whereas we obtain a mean of at least 998 and a geometric mean of 3.9. In addition, we solve all the instances but one and Choco do not solve four instances.

In Table 3.19, we compare the MIP solver Concorde [Applegate et al., 2006] and our best model. Note that the results for Concorde are obtained on our machine. For medium size instances such as ch130 or d198, the difference between Concorde and us is small. We solve ch130 in 0.7s whereas Concorde solves it in 0.3s and we solve d198 in 7.0s whereas Concorde solved it in 2.6s. In addition, some instances are solved faster with our best model such as gr202 (1.2s vs 1.9s). Nevertheless, Concorde scales very well. Unfortunately, we did not. Indeed, Concorde solves d493 in 47.8s whereas we solve it in 4,990s: that is two orders of magnitude. This result can be seen as a bad result. However, we saw in Table 3.18 that the implementation for this kind of methods have a huge impact: Concorde has been implemented over several years by top researchers. Thus, match the Concorde results in a thesis is hard. However, considering the geometric mean for this instance set, we only lost a magnitude order. It is encouraging since before the thesis multiple orders of magnitude were observed for the same kinds of problems.

	WCC (1)		Best model (2)		ratio (1)/(2)	
Instances	time(s)	#sn	time(s)	#sn	time	#sn
a280	30.6	28,789	8.1	2,351	3.8	12.2
ali535	*t.o.*	*t.o.*	13,013.9	1,339,925	> 7.7	-
bier127	0.3	153	0.3	53	1.0	2.9
brg180	0.4	95	1.2	101	0.3	0.9
ch130	2.6	3,661	0.7	383	3.7	9.6
ch150	2.5	3,187	0.8	303	3.1	10.5
d198	6.5	4,573	7.0	3,093	0.9	1.5
d493	*t.o.*	*t.o.*	4,989.9	488,439	> 20.0	-
eil101	0.1	89	0.2	65	0.5	1.4
gil262	7,160.4	6,345,749	2,042.2	1,040,943	3.5	6.1
gr120	0.6	585	0.7	279	0.9	2.1
gr137	2.2	3,135	1.0	513	2.2	6.1
gr202	2.0	2,023	1.2	383	1.7	5.3
gr229	382.8	437,417	41.2	26,565	9.3	16.5
gr431	*t.o.*	*t.o.*	520.6	60,537	> 192.1	-
gr666	*t.o.*	*t.o.*	14,719.6	1,106,125	> 6.8	-
gr96	0.6	555	0.5	227	1.2	2.4
kroA100	2.1	4,469	0.8	483	2.6	9.3
kroA150	5.8	7,277	1.9	1,269	3.1	5.7
kroA200	2,639.3	2,846,977	63.7	35,209	41.4	80.9
kroB100	2.2	4,087	1.0	1,045	2.2	3.9
kroB150	113.7	169,545	8.7	6,165	13.1	27.5
kroB200	189.7	189,965	15.1	9,381	12.6	20.2
kroC100	0.8	1,259	0.5	233	1.6	5.4
kroD100	0.3	283	0.3	149	1.0	1.9
kroE100	3.2	9,947	0.9	641	3.6	15.5
lin318	33.7	17,601	6.4	1,817	5.3	9.7
pcb442	*t.o.*	*t.o.*	91,137.7	14,852,219	> 1.1	-
pr124	1.7	2,031	1.2	827	1.4	2.5
pr136	12.3	18,075	8.9	7,229	1.4	2.5
pr144	1.0	795	1.0	345	1.0	2.3
pr264	3.4	679	3.8	575	0.9	1.2
pr299	*t.o.*	*t.o.*	1,596.5	489,039	> 62.6	-
rat195	23.0	22,583	17.9	11,601	1.3	1.9
rat99	0.1	95	0.2	61	0.5	1.6
rd400	*t.o.*	*t.o.*	6,721.0	1,727,475	> 14.9	-
si175	1,003.7	2,044,011	100.7	92,573	10.0	22.1
st70	0.1	87	0.2	65	0.5	1.3
tsp225	266.7	267,537	23.5	11,557	11.3	23.1
u159	0.5	459	0.4	133	1.3	3.5
mean	$> 17,797.4$		3,376.54			
geo mean	> 31.2		10.04			

Table 3.16: Overall results obtained in this thesis for the sequential solving of the TSP in CP.

Instances	Choco WCC (1) time(s)	Thesis WCC (2) time(s)	ratio (1)/(2) time
a280	806.6	30.6	26.4
bier127	1.8	0.3	6.0
ch130	72.0	2.6	27.7
ch150	28.0	2.5	11.2
d198	76.1	6.5	11.7
eil101	1.0	0.1	10.0
gil262	*t.o.*	7,160.4	4.2
gr120	3.3	0.6	5.5
gr137	33.7	2.2	15.3
gr202	24.4	2.0	12.2
gr229	14,025.4	382.8	36.6
gr96	3.3	0.6	5.5
kroA100	20.5	2.1	9.8
kroA150	119.9	5.8	20.7
kroA200	*t.o.*	2,639.3	11.4
kroB100	17.0	2.2	7.7
kroB150	1,609.0	113.7	14.2
kroB200	2,218.9	189.7	11.7
kroC100	5.9	0.8	7.4
kroD100	0.9	0.3	3.0
kroE100	69.2	3.2	21.6
pr107	1.5	0.9	1.7
pr124	12.7	1.7	7.5
pr144	12.9	1.0	12.9
pr152	89.2	4.4	20.3
pr226	6.4	1.1	5.8
pr264	21.6	3.4	6.4
pr299	*t.o.*	*t.o.*	*t.o.*
rat195	330.5	23.0	14.4
rat99	0.8	0.1	8.0
si175	4,544.1	1,003.7	4.5
ts225	*t.o.*	*t.o.*	*t.o.*
tsp225	4,171.4	266.7	15.6
u159	7.3	0.5	14.6
mean	$>$ 4,362.8	$>$ 2,113.4	
geo mean	$>$ 82.4	$>$ 9.4	

Table 3.17: Comparison of the state-of-the-art code before the thesis and our implementation of the same model.

Instances	Choco WCC (1) time(s)	Best model (2) time(s)	ratio (1)/(2) time
a280	806.6	8.1	99.6
bier127	1.8	0.3	6.0
ch130	72.0	0.7	102.9
ch150	28.0	0.8	35.0
d198	76.1	7.0	10.9
eil101	1.0	0.2	5.0
gil262	*t.o.*	2,042.2	14.7
gr120	3.3	0.7	4.7
gr137	33.7	1.0	33.7
gr202	24.4	1.2	20.3
gr229	14,025.4	41.2	340.4
gr96	3.3	0.5	6.6
kroA100	20.5	0.8	25.6
kroA150	119.9	1.9	63.1
kroA200	*t.o.*	63.7	471.0
kroB100	17.0	1.0	17.0
kroB150	1,609.0	8.7	184.9
kroB200	2,218.9	15.1	146.9
kroC100	5.9	0.5	11.8
kroD100	0.9	0.3	3.0
kroE100	69.2	0.9	76.9
pr107	1.5	0.9	1.7
pr124	12.7	1.2	10.6
pr144	12.9	1.0	12.9
pr152	89.2	2.4	37.2
pr226	6.4	1.0	6.4
pr264	21.6	3.8	5.7
pr299	*t.o.*	1,596.5	18.8
rat195	330.5	17.9	18.5
rat99	0.8	0.2	4.0
si175	4,544.1	100.7	45.1
ts225	*t.o.*	*t.o.*	*t.o.*
tsp225	4,171.4	23.5	177.5
u159	7.3	0.4	18.3
mean	> 4362.8	> 998.4	
geo mean	> 82.4	> 3.9	

Table 3.18: Comparison of the state-of-the-art code before the thesis [Fages et al., 2016] and now.

Instances	Concorde (1) time(s)	Best model (2) time(s)	ratio (1)/(2) time
a280	0.9	8.1	0.1
ali535	5.2	13,013.9	0.0
bier127	0.4	0.3	1.2
brg180	0.3	1.2	0.2
ch130	0.4	0.7	0.6
ch150	0.3	0.8	0.4
d198	2.6	7.0	0.4
d493	47.8	4,989.9	0.0
eil101	0.2	0.2	0.8
gil262	3.7	2,042.2	0.0
gr120	0.3	0.7	0.4
gr137	1.0	1.0	1.0
gr202	1.9	1.2	1.6
gr229	8.7	41.2	0.2
gr431	30.2	520.6	0.1
gr666	13.4	14,719.6	0.0
gr96	0.8	0.5	1.5
kroA100	0.2	0.8	0.3
kroA150	1.0	1.9	0.6
kroA200	1.1	63.7	0.0
kroB100	0.4	1.0	0.4
kroB150	0.8	8.7	0.1
kroB200	0.4	15.1	0.0
kroC100	0.2	0.5	0.3
kroD100	0.2	0.3	0.6
kroE100	0.4	0.9	0.4
lin318	2.6	6.4	0.4
pcb442	13.1	91,137.7	0.0
pr124	0.4	1.2	0.4
pr136	0.6	8.9	0.1
pr144	0.5	1.0	0.5
pr264	0.6	3.8	0.2
pr299	4.2	1,596.5	0.0
rat195	2.6	17.9	0.1
rat99	0.2	0.2	1.1
rd400	23.1	6,721.0	0.0
si175	3.6	100.7	0.0
st70	0.1	0.2	0.7
tsp225	3.9	23.5	0.2
u159	0.2	0.4	0.5
mean	4.5	3376.5	
geo mean	1.1	10.0	

Table 3.19: Comparison of Concorde and our best model for all the considered graphs of this thesis.

Moreover, there is no record of a graph with more than 300 nodes solved with CP before this thesis. Therefore, we still believe that there is a hope to push further the competitiveness of the TSP model in CP.

In Table 3.20, we show a comparison between Concorde and our best model for all the instances from the TSPLib with fewer than 100 nodes. For those instances, the results are quite good: we are in mean two times faster than Concorde (0.05s vs 0.1s). However, these results should be carefully analyzed since it is very small solving times. Indeed, for small instances, Concorde can be slower since it can involve more sophisticated methods. Nevertheless, the most important thing is that the results are comparable for both the CP model and Concorde for the graphs with fewer than a hundred nodes. Therefore, we are competitive for these graphs size.

Instances	Concorde (1) time(s)	Best model (2) time(s)
burma14	0.03	0.00
ulysses16	0.07	0.00
gr17	0.02	0.00
gr21	0.03	0.00
ulysses22	0.12	0.00
gr24	0.03	0.00
fri26	0.03	0.00
bays29	0.02	0.00
dantzig42	0.04	0.03
swiss42	0.03	0.01
att48	0.12	0.03
gr48	0.06	0.08
hk48	0.05	0.00
eil51	0.04	0.05
berlin52	0.06	0.00
brazil58	0.11	0.06
st70	0.13	0.17
eil76	0.04	0.04
gr96	0.75	0.46
rat99	0.20	0.16
mean	0.10	0.05

Table 3.20: Comparison of Concorde and our best model for all the symmetric graphs with fewer than a hundred nodes in the TSPLib.

3.9 Conclusion

In this section, we introduced and discussed the state-of-the-art TSP model in CP. Then, we introduced the k-cutset constraint, the mandatory Hamiltonian path constraint and the one-tree constraints. These three constraints allow improving the results with a factor greater than a magnitude order in geometric mean on the selected instances set. It should be noted that the three constraints are based on the fact that there are mandatory edges in the graph: that information is generally underused. In addition, we studied the variation of the objective function and we engineered the CP-based Lagrangian relaxation in order to improve solving times. All this work makes the solving of the TSP much more efficient and improve the competitiveness of the TSP model in CP with Concorde.

CHAPTER 4

Parallelization of the TSP solving in CP

Embarrassingly Parallel Search (EPS) parallelizes the search for solutions in CP by decomposing the initial problem into a huge number of sub-problems that are consistent with propagation. Then, each waiting worker takes a sub-problem and solves it. The process is repeated until all the sub-problems have been solved. Unfortunately, EPS decomposition is a depth-bounded process unlike the search strategy used for solving the TSP (LCFirst) which is depth-first. We then introduce Bound-Backtrack-and-Dive, a method which solves this issue. First, we run a sequential solving of the problem with a bounded number of backtracks in order to extract key information from LCFirst, then we decompose with EPS using that information rather than LCFirst. The experimental results show that we obtain almost a linear gain on the number of cores and that Bound-Backtrack-and-Dive may considerably reduce the number of backtracks performed for some problems. When hundreds of cores are implied, sub-problems with extremely different solving times may appear, for example one requiring a huge part of the total solving time. In this case the load balancing is poor. We show that a general increase in the number of sub-problems does not solve this imbalance. We present a method that identifies the presence of difficult sub-problems during the solving process and decompose them again. Experimental results for the TSP show a good improvement of load balancing and a better scaling with a hundred cores.

4.1 Introduction

In constraint programming, there are two main approaches for solving a problem in parallel: the search space splitting method (*i.e.* the work-stealing approach) and problem decomposition method (*i.e.* embarrassingly parallel search).

The work-stealing method dynamically splits the search space during the solving [Burton and Sleep, 1981, Halstead, 1984]. When a worker has finished exploring a sub-problem, it asks other workers for another sub-problem. If another worker agrees to the demand, then it dynamically splits its current sub-problem into two disjoint sub-problems and sends one sub-problem to the starving worker. The starving worker "steals" some work from the busy one. Several implementations of work stealing approach has been designed [Perron, 1999, Vidal et al., 2010]. The most recent ones such as like Bobpp tries to be as independent as possible from the solver [Galea and Le Cun, 2007, Le Cun et al., 2007].

The Embarrassingly Parallel Search (EPS) [Régin et al., 2013, Malapert et al., 2016, Régin and Malapert., 2017] is a more recent method. It statically decomposes the initial problem into a huge number of sub-problems that are consistent with propagation (*i.e.* running the propagation mechanism on them does not detect any inconsistency). Then, each waiting worker takes a sub-problem and solves it. The process is repeated until all the sub-problems have been solved. The assignment of the sub-problems to workers is dynamic, and there is no communication between the workers. EPS is based on the idea that if there are many sub-problems to solve, then the solving times of the workers will be balanced even if the solving times of the sub-problems are not. In other words, the load balancing should be automatically obtained in a statistical sense.

As EPS gives very good results on many problems, it was legitimate to use it to solve the TSP. In order to do so, some modifications of the EPS decomposition mechanism have been required for two reasons. First, the model of the TSP in CP contains a set-variable with the mandatory edges as lower bound and optional edges as upper bound. Decomposing with a set-variable is not as trivial as a classical Cartesian product because the order must be carefully handled while enumerating. Next, a new phenomenon appeared in the decomposition while experimenting: the increase of the assignment limit in the enumeration may not lead to an increase in the number of sub-problems. It turns out that the TSP search tree is extremely heterogeneous. Thus, we introduce an extended mechanism for coherent enumeration of set-variables and for stopping the decomposition in case of non-progression. Finally, the decomposition mechanism is a depth-bounded process unlike the LCFirst search strategy which is depth-first. Therefore, we introduced the method *Bound-Backtrack-and-Dive* [Isoart and Régin, 2020b]. It proceeds in two steps. A sequential solving of the problem with a bounded number of backtracks is run to extract key information from LCFirst. Then, the EPS decomposition uses that information rather than LCFirst. The experimental results show that almost a linear gain on the number of cores is obtained and that *Bound-Backtrack-and-Dive* may considerably reduce the number of search nodes performed for some problems. For many problems, the same kind of results are obtained when increasing the number of cores. Unfortunately, this is not true for all problems and often wrong for very difficult problems. It is precisely for those problems that we would like to be able to use the combined power of many computational cores. Therefore, we will introduce a method detecting this kind of difficult problems and to improve their solving in parallel. In other words, we suggest to slightly modify EPS in order to improve its robustness. EPS has many qualities, such as its simplicity, its independence from the solvers used and search strategies, the absence of communication between

workers and weak communication between the master and the workers. It is important to respect these advantages and not to make any changes that could jeopardize them.

4.2 EPS

Embarrassingly Parallel Search (EPS) decomposes the initial problem into a huge number of sub-problems that are consistent with propagation (*i.e.* running the propagation mechanism on them does not detect any inconsistency). Then, each waiting worker takes a sub-problem and solves it until all sub-problems have been solved. The main challenge of the decomposition is not to define equivalent problems, it is to avoid having some workers without work whereas some others are running during the solving step. In order to increase our chances to obtain well-balanced activity times for the workers*, EPS decomposes the initial problem into a lot of sub-problems. It is usually considered that a good number of sub-problems per worker is between 30 to 300 [Régin et al., 2013, Malapert et al., 2016, Régin and Malapert., 2017].

The generation of q sub-problems is not straightforward because the number of sub-problems consistent with the propagation may not be related to the Cartesian product of some domains. A simple algorithm could be to perform a Breadth First Search (BFS) in the search tree until the desired number of sub-problems consistent with the propagation is reached. Unfortunately, it is not easy to perform a BFS efficiently mainly because the BFS is not an incremental algorithm like Depth-First Search (DFS). Therefore, EPS uses a process resembling an iterative deepening depth-first search [Korf, 1985]: we consider a set $Y \subseteq X$ of variables and we only assign the variables of Y. The search is stopped when all the variables of Y are assigned. In other words, we never try to assign a variable that is not in Y. This process is repeated until all assignments of Y consistent with the propagation have been found. Each branch of a search tree computed by this search defines an assignment (*i.e.* a sub-problem). To generate q sub-problems, we repeat the previous method by adding variables to Y if necessary, until the number of sub-problems is greater than or equal to q. Note that the decomposition can be performed in parallel.

4.2.1 Modifications of EPS mechanisms

The search strategy for the TSP only considers edges represented by a set-variable. A decomposition for a set-variable works by iteratively increasing its cardinality value (*i.e.* the assignment enumeration limit). More precisely, the decomposition starts by setting the cardinality to a chosen value. Then, it solves the problem with this cardinality value and keep the branches of the search tree as a sub-problem such that the number of mandatory edges is greater than the cardinality. If we have obtained q sub-problems, then the decomposition is finished. Otherwise, the cardinality is increased. Finally, this process is repeated until q sub-problems are found.

In Figure 4.1, we show an example of the decomposition step such that we look for 4 sub-problems ($q = 4$). At the beginning, the cardinality is set to 2 and three sub-problems consistent with the propagation are obtained (*i.e.* the black squares). Then, as we are looking for 4 sub-problems, the cardinality is increased. We set it to 4 and we obtain 5 sub-problems (*i.e.* the orange squares). Thus, the required number of sub-problems is reached ($5 > q$). The process is stopped, and the parallel solving of the sub-problems can begin.

*The activity time of a worker is the sum of the solving times of its sub-problems.

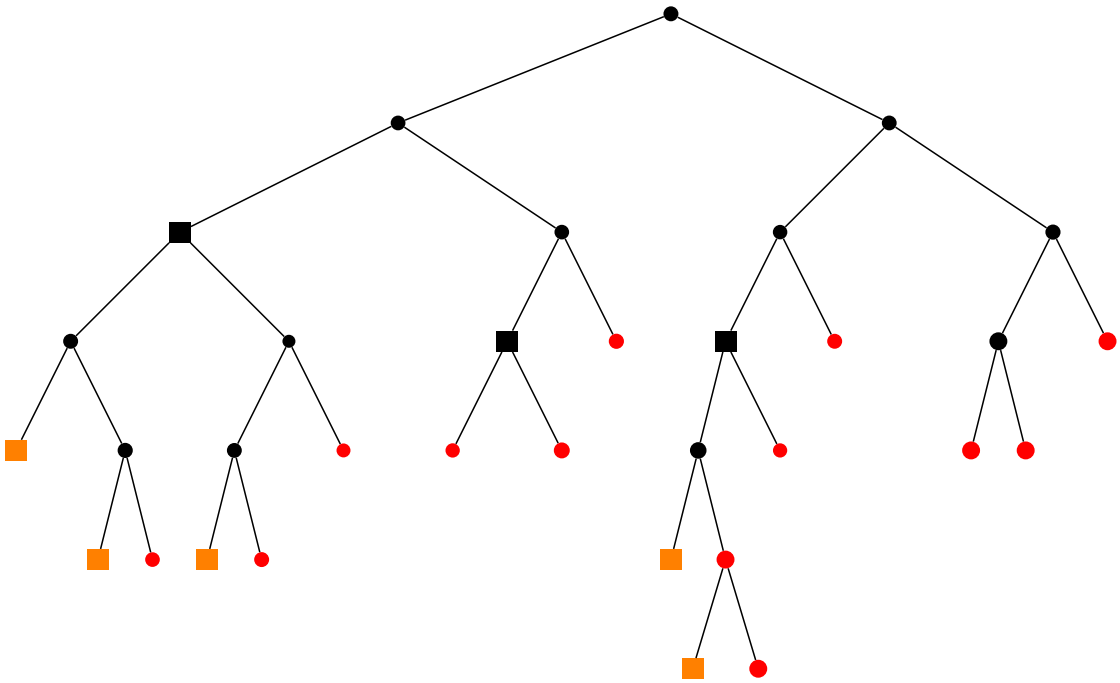


Figure 4.1: Search for four sub-problems (q=4). In this search tree, the black squares are the sub-problems with two mandatory edges assigned by the search (assignment enumeration limit is equal to two), the orange squares are the sub-problems with four mandatory edges by the search (assignment enumeration limit is equal to four) and the red circles are the inconsistent sub-problems.

In some cases, frequently encountered with the TSP, the increase of variables to Y leads to a reduction in the number of sub-problems generated. This phenomenon is named: non-monotonic decomposition. If suddenly many branches fail in the search tree, it may be possible that more problems have been removed than generated. For special cases such as this one, it is preferable to stop the decomposition. Indeed, spending a large part the overall solving time in the decomposition is a bad idea. The main reason is that the decomposition expands the search tree in breadth, that is hardly incremental. In addition, the decomposition works by step and therefore at each step it must transmit the sub-problems found to the master which contain a lot of data. Then, if the non-monotonic decomposition leads to many steps, stopping the decomposition is worthwhile. Thus, we defined a stopping criterion other than the number of sub-problems generated [Isoart and Régin, 2020b]: if two successive additions of variables in Y decreases the number of generated sub-problems, then the decomposition is stopped. An example is given in Figure 4.2 such that $q = 4$. First, there are 3 sub-problems generated. Since $q > 3$, the cardinality is increased to three and 2 sub-problems are generated. Note than the number of sub-problems is decreased. Again, $q > 2$ and the cardinality is increased leading to one sub-problem. Therefore, there are two successions of increasing cardinality leading to a decreasing number of sub-problems. Then, the decomposition step is stopped, and the solving begin.

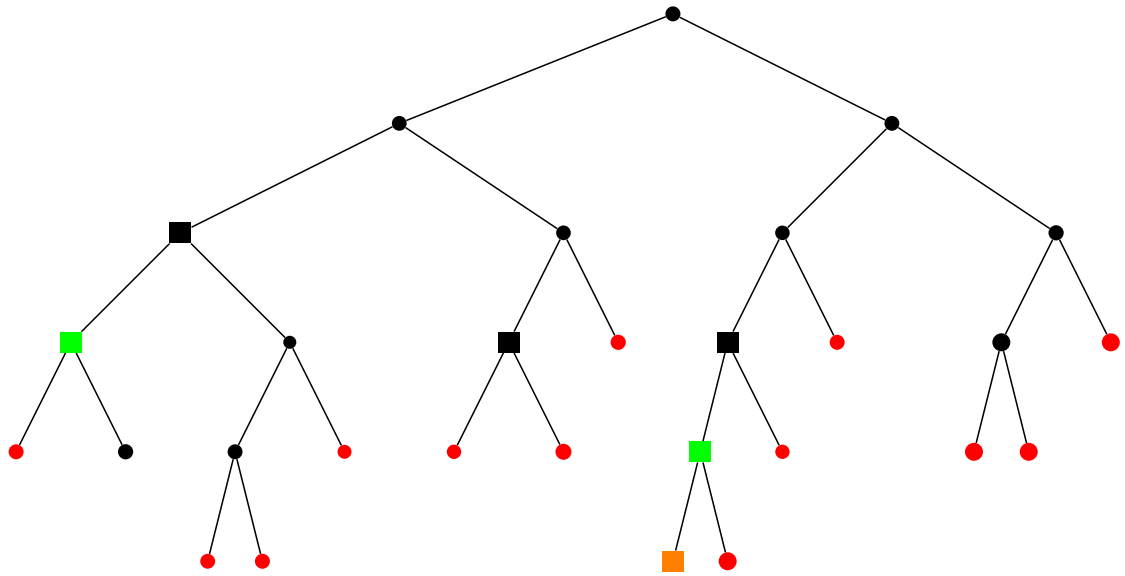


Figure 4.2: Search for four sub-problems (q=4). In this search tree, the black squares are the sub-problems with two mandatory edges by the search, green squares are the sub-problems with three mandatory edges by the search , the oranges square are the sub-problems with four mandatory edges by the search and the red circles are the inconsistent sub-problems.

When decomposing with EPS, we have two possibilities for the set-variables: we try to transform them into a set of classic variables and constraints (to break symmetries), or we adapt the usual algorithm to the set-variables. In the first case, it is important to break the symmetries. Indeed, a set-variable sx involving the values $\{a, b, c, d\}$ and whose cardinality is equal to 3 has only one solution which implies the values a, b and c, whereas if we replace sx by 3 variables x_1, x_2 and x_3 we risk generating the solutions $x_1 = a$, $x_2 = b$, $x_3 = c$, and $x_1 = b$, $x_2 = a$, $x_3 = c$, *etc.* The introduction of an order between the variables avoids this concern. Nevertheless, when the cardinality is not fixed, this transformation is more delicate. It is therefore preferable to adapt the decomposition algorithm to the set-variable. When we consider a set $Y \subseteq X$ of variables, we pay attention to the set-variables. A classical variable is instantiated by a single value, whereas for a set-variable we will determine how many of its values should be instantiated at most. For instance, sx can be instantiated with at most 1, 2 or 3 values. Its cardinality defines only the maximum because we search for partial assignments. In general, all but one set-variable of Y will be potentially instantiated with their maximum possible values (*i.e.* the maximum cardinality).

Optimization problems, like the TSP, deserve a little more attention. EPS manages the value of the objective function as follows: when a worker takes a sub-problem, it also takes the best objective value that a worker has obtained so far, and when a worker solves a sub-problem, it communicates the best objective value found in order to update it for the next sub-problems. EPS does not use any additional communications. Note that it is not possible to use the objective value while decomposing the problem. Indeed, the objective value associated with the assignment of a subset of variables is not necessarily a valid bound for the problem in general and that assignment cannot necessarily be extended to a solution.

4.3 Decomposition issue for the TSP and LCFirst

In order to parallelize algorithms, it is important to avoid as much as possible parallel calculations that would not have been done sequentially. A simple example is the one we gave on the assignment of the set-variables: we do not want to generate symmetry in the search tree.

As explained in Section 3.3, the search strategy for the TSP uses a graph interpretation of Last Conflict heuristics named LCFirst. Such a search strategy learns from left branches for the right branches. If the depth is bounded, such as in the decomposition of EPS, then it can happen that we do not reach the bottom of the search tree. Therefore, the right branches are performed before fully exploring the left branches. For LCFirst, it means the use of a different LCNode than the one used if the left branch is completely performed. This leads to different search tree and therefore different results (see Figure 4.3).

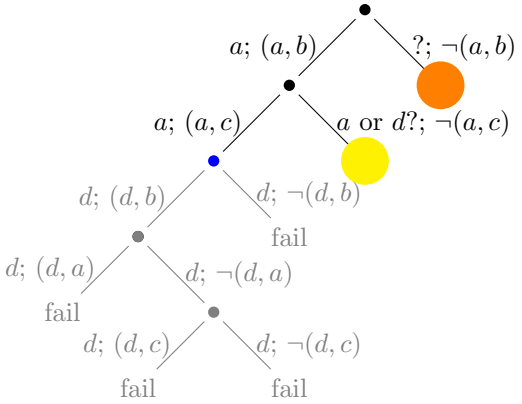


Figure 4.3: A search tree. The black area is the zone where the cardinality is lower or equal than 2, the blue node represents a search node such that the cardinality is equal to two. If the search tree is continued after the blue node, then we obtain the gray area of the search tree. Yellow area needs to know the LCNode of the previous branching node (*i.e.* d if the gray area is performed, a otherwise). Orange area needs to know the LCNode from the yellow area.

In Figure 4.3, when the cardinality is set to 2, the gray area is not visited since the size of all assigned edges ($\{ab, ac\}$) is equal to 2. When solving sequentially, the gray area is visited since there is no assignment enumeration limit. Thus, the LCNode value (yellow zone) for the branch $root - ab - \neg ac$ depends on the visit of the gray area. If it is visited, then LCNode=d, otherwise LCNode=a. For the branch $root - \neg ab$ (orange zone) the LCNode value depends on the result of the previous LCNode value (yellow zone). Hence, the decomposition can generate a very different search tree than a sequential execution.

One can legitimately ask the following question: how is the solving impacted if we get the wrong LCNode?

The impact can be very bad. For example, if we take the instance ali535 of TSPLib [Reinelt, 1991], its sequential solving time is $13,014$s for $1,339,925$ search nodes, but its solving time on 8 cores with EPS is $18,564$s for $1,742,497$ search nodes. So, the parallel version is 43% slower and it performs 30% more search nodes whereas it involves 8 more cores. Later, the various experiments will show that this problem is not unusual. These results can be ex-

plained by the remarks made for Figure 4.3. This shows that a few bad LCNode can significantly increase combinatorics. In addition, disabling LCFirst from the decomposition strongly degrades the results. In practice, it leads to a search strategy that jumps in the graph and does not exploit the structure of the graph.

Thus, the classical EPS decomposition is not enough for a dynamic search strategy such as LCFirst.

4.3.1 Bound-Backtrack-and-Dive and decomposition

To solve the LCFirst issue in the decomposition, we suggest using a diving heuristic, named *Bound-Backtrack-and-Dive*. It consists in running a sequential solving of the problem with a bounded number of backtracks in order to build an ordered set of nodes, called α, that we will use to represent the impact of LCNode choice. Then, we run EPS with α as a parameter and we use it for LCNode selection during decomposition.

4.3.1.1 Computation of α

For each node $i \in X$, let α_i be the value representing the failure impact of branching around i. Thus, the higher α_i is, the more important the node is for LCFirst. More exactly, when a node i is backtracked in the search tree, we update α_i with $\alpha_i \leftarrow \alpha_i + C/depth^2$ where C is a constant such that $C > depth^2_{max}$. Therefore, a failure thrown at a shallow depth in the search tree will increase the value of α_{LCNode} much more than if the failure is thrown at a deep depth in the search tree. In order to represent this priority, a non-linear denominator has been chosen. The idea of prioritizing LCNodes causing shallow failures in the search tree allows us to narrow the search strategy to the difficult areas of the graph. Thus, we suggest computing α with the above method for a limited number of backtracks. When the limit is reached, the solving is stopped and α is returned.

4.3.1.2 Decomposition

After computing α, we decompose the initial problem with a modified LCFirst algorithm. When a node is emptied (*i.e.* there are no more optional edges around LCNode) LCFirst will usually look for an edge in the graph using a heuristic, keep one end and empty it again. Here, we suggest that when a node is emptied, instead of looking for an edge in the graph and keeping one extremity, we select the non-emptied node with the highest value in α. Then, with a classical search strategy (*i.e.* minDeltaDeg), we select an edge around this node until we have emptied it. A possible implementation of modified LCFirst is described in Algorithm 11.

At the end of the decomposition, we drop α and go back to classical LCFirst. We mainly use α because the decomposition is done in breadth and LCFirst gets good LCNode by depth. Thus, while solving EPS sub-problems we can use depth-based methods (*i.e.* classical LCFirst) since we solve them. It allows learning locally about each sub-problem and provide better results.

4.3.2 Experiments

In this section, we will experimentally study *Bound-Backtrack-and-Dive* method. The parallelization is motivated by solving larger instances or solving faster slow instances. Therefore, we consider a subset of the instances of the previous sections. Note that we keep all the instances that are solved in more than 2 seconds. The *Bound-Backtrack-and-Dive* algorithm is noted BBD. For

Algorithm 11: $LCFirst(G = (X), \alpha)$

```
decompositionLCFirst (G, α)
    Input: The current graph to solve G and an ordered set of nodes α
    Output: An edge (u, v)
    global LCNode;
    if LCNode ≠ nil then
        // Select an optional edge adjacent to LCNode in G
        if O(LCNode) ≠ ∅ then return select(O(LCNode)) ;
    idMax ← nil;
    valMax ← −MAX_VALUE;
    foreach α_i ∈ α do
        // search for the node with the highest α_i having
           optional neighbors
        if α_i > valMax and O(i) ≠ ∅ then
            idMax ← i;
            valMax ← α_i;
    if idMax ≠ nil then
        (u, v) ← select(O(idMax));
        LCNode ← idMax;
    else
        (u, v) ← select(O);
        LCNode ← u;
    return (u, v)
```

these experiments, we used 8 threads on our machine in order to simulate a classical laptop. In the next section, we will show how to scale to a large number of threads.

Initially, the goal is to make the overall same number of search nodes with and without EPS. In practice, it is very difficult to achieve it because of the dynamic strategies, the Lagrangian relaxation and the order of each sub-problems are taken. However, we show through the experiments that the use of *Bound-Backtrack-and-Dive* allows obtaining a comparable/improving number of search nodes. Because of load balancing problems in parallelism, looking at the number of search nodes is a good metric.

4.3.2.1 Behavior of EPS

In Table 4.1 we compare the sequential solving and EPS without BBD in which we integrate or not LCFirst for the decomposition step. Compared to the sequential solving, EPS improves the results on average with or without LCFirst during the decomposition. However, many instances are solved with more search nodes when EPS is used than without EPS. For example, pr299 is solved with $489,039$ search nodes for a sequential solving compared to $993,101$ search nodes for EPS with LCFirst during the decomposition and $753,791$ search nodes for EPS without LCFirst during the decomposition. Due to the large increase in the number of search nodes, this instance with these configurations is slower to solve with EPS than in sequential. In addition, we notice that (1) performs 306 search nodes per seconds whereas (2) performs 314 nodes per seconds and (3) performs 322 nodes per seconds. However, we use 8 threads therefore we could have expected a gain of a factor of almost 8 on the performances. Then, the issue is that a bad load balancing is observed: in practice it is an extremely heterogeneous problem. The last sub-problem is solved on 1 thread whereas the other threads have no more work to do. Note that obtaining a factor gain of the number of threads is a really hard thing to obtain in parallelism. In addition, the decomposition of a problem in many sub-problems can be time consuming here since the search tree is very particular (very deep depth on some branches such that the branches are not that large and shallow for the others). We will discuss that in the next section. Nevertheless, other instances such as kroA200 are better parallelized. While its sequential execution performs $35,209$ search nodes, EPS noDivLCDec performs $30,421$ search nodes and EPS noDiveNoLCDec performs $46,897$ search nodes. Thus, a gain of a factor of 3.8 and 3.0 respectively in solving time is observed with almost the same number of search nodes. Next, we will see how to improve the results for the solving times and the robustness with the diving.

4.3.2.2 Impact of the backtrack limit on the solving

In Table 4.2 we introduce BBD and we study the impact of different backtracks limit while computing α.

First, for problems with a low solving time, the time spent diving can become significantly high compared to the overall solving time. For example, for a280 with limitBk=$1,000$, we spend 2.9s in the diving and 3.2s for decomposing and solving the problem. Increasing limitBk here increases the overall solving time significantly since the value of limitBk exceeds the number of backtracks performed sequentially. Thus, the instance is completely solved in sequential before being solved with EPS. The lin318 instance has the same issue, increasing the backtrack limit fully solves the problem in sequential during diving before doing the decomposition and solving, which is actually very fast (about 4s) while diving takes 3.8s for limitBk=$1,000$, then about 10s for

	Sequential (1)		EPS noDiveLCDec (2)		ratio (1) / (2)		EPS noDiveNoLCDec (3)		ratio (1) / (3)	
Instances	time(s)	#sn	time(s)	#sn	time	#sn	time(s)	#sn	time	#sn
a280	8.1	2,351	3.0	2,255	2.7	1.0	4.2	2,473	1.9	1.0
ali535	13,013.9	1,339,925	3,595.6	2,180,621	3.6	0.6	18,563.7	1,742,497	0.7	0.8
d198	7.0	3,093	10.4	4,331	0.7	0.7	9.5	5,835	0.7	0.5
d493	4,989.9	488,439	8,429.1	912,905	0.6	0.5	4,485.1	637,677	1.1	0.8
gil262	2,042.2	1,040,943	772.5	1,632,833	2.6	0.6	1,514.1	1,271,733	1.3	0.8
gr229	41.2	26,565	24.5	21,793	1.7	1.2	32.2	27,551	1.3	1.0
gr431	520.6	60,537	177.4	27,209	2.9	2.2	294.6	40,851	1.8	1.5
gr666	14,719.6	1,106,125	13,208.1	1,703,945	1.1	0.6	5,533.8	1,244,949	2.7	0.9
kroA200	63.7	35,209	16.8	30,421	3.8	1.2	21.5	46,897	3.0	0.8
kroB150	8.7	6,165	5.2	7,129	1.7	0.9	6.1	11,059	1.4	0.6
kroB200	15.1	9,381	5.7	7,579	2.6	1.2	6.4	9,283	2.4	1.0
lin318	6.4	1,817	6.4	1,643	1.0	1.1	6.2	1,697	1.0	1.1
pcb442	91,137.7	14,852,219	20,481.4	13,183,769	4.4	1.1	17,919.2	15,176,125	5.1	1.0
pr136	8.9	7,229	3.4	7,465	2.6	1.0	3.5	7,905	2.5	0.9
pr264	3.8	575	9.4	2,669	0.4	0.2	10.5	2,797	0.4	0.2
pr299	1,596.5	489,039	3,158.5	993,101	0.5	0.5	2,341.1	753,791	0.7	0.6
rat195	17.9	11,601	14.8	17,397	1.2	0.7	16.1	19,673	1.1	0.6
rd400	6,721.0	1,727,475	3,226.4	1,795,849	2.1	1.0	3,398.2	1,849,117	2.0	0.9
si175	100.7	92,573	17.9	93,929	5.6	1.0	13.2	61,083	7.6	1.5
tsp225	23.5	11,557	14.3	14,431	1.6	0.8	13.0	12,439	1.8	0.9
mean	6,752.3	1,065,640.9	2,659.0	1,132,063.7			2,709.6	1,146,271.6		
geo mean	163.1	52,025.9	94.7	62,956.9			103.5	64,659.9		

Table 4.1: Comparison of sequential solving, EPS without diving, EPS without diving and without LCFirst during decomposition. A ratio column compares the sequential method with each EPS method. Greater ratio is better.

	limitBk=1,000 (1)			limitBk=5,000 (2)			limitBk=10,000 (3)		
Instances	div. time(s)	time(s)	#sn	div. time(s)	time(s)	#sn	div. time(s)	time(s)	#sn
a280	2.9	3.2	2,125	7.1	2.1	853	7.2	2.1	1,041
ali535	26.1	3,339.0	1,049,467	115.7	3,806.8	1,228,463	195.0	3,629.4	1,153,801
d198	3.8	9.5	3,935	6.8	9.8	4,433	6.9	9.0	3,993
d493	12.8	6,451.3	889,805	64.4	6,806.7	999,233	128.1	5,500.2	802,251
gil262	4.3	219.3	535,595	15.4	486.5	798,505	26.4	531.6	899,261
gr229	2.0	47.9	89,257	7.7	40.5	99,817	15.0	27.7	56,253
gr431	11.6	127.4	54,197	45.6	180.1	53,763	88.8	181.8	64,897
gr666	19.4	4,581.1	1,499,235	78.8	3,917.1	1,234,091	149.9	4,368.1	1,207,703
kroA200	2.1	31.6	55,345	9.5	36.0	97,261	17.3	38.4	95,925
kroB150	1.8	10.7	16,437	6.6	9.0	27,181	8.1	10.5	20,945
kroB200	1.8	14.6	27,743	7.5	10.1	13,273	13.7	13.4	14,069
lin318	3.8	4.0	1,055	6.3	3.6	809	6.2	3.8	747
pcb442	5.1	8,240.2	4,251,747	25.6	13,938.6	2,481,369	49.0	9,064.0	1,846,195
pr136	1.2	5.5	9,805	5.4	4.4	6,559	7.7	3.0	5,321
pr264	3.4	8.1	3,331	3.4	8.5	3,379	3.4	8.7	3,397
pr299	5.3	2,313.6	792,041	20.5	691.0	566,139	40.1	683.8	645,759
rat195	1.9	8.6	10,797	7.4	16.0	13,521	14.1	24.8	18,197
rd400	6.0	1,767.1	1,571,817	24.1	900.4	1,526,415	47.7	1,117.2	1,688,565
si175	1.5	15.7	70,297	5.9	22.5	108,157	11.2	23.0	114,971
tsp225	2.1	18.1	30,117	10.2	24.7	26,987	19.1	23.2	27,109
mean	5.9	1,360.8	548,207.4	23.7	1,545.7	464,510.4	42.7	1,263.2	433,520.0
geo mean	3.9	83.8	66,865.1	13.6	83.2	63,945.8	21.8	82.6	61,820.6

Table 4.2: Comparison of the results according to the limit of the number of backtracks allowed for diving.

limitBk=$5,000$ and $10,000$. Thus, it is better to use a very small limitBk for problems having very low solving times because the diving should not take a significant amount of time from the overall solving time. To do so, sampling methods could be used. Then, for bigger problems, limitBk also has an impact. Indeed, we notice that the best mean solving time is obtained with limitBk=$10,000$. However, the geometric mean shows that the differences between them are actually quite small in terms of solving time and backtracks.

In Table 4.3, we compare the ratio between limitBk of Table 4.2 and the sequential solving. We observe that the most stable configuration is limitBk=1,000 since it allows us to obtain stable results on both small and large instances. Thus, we recommend using limitBk=1,000 and use this configuration for the next experiments.

4.3.2.3 Impact of the sub-problems per worker on the solving

In Table 4.4, we compare the impact of the number of sub-problems per worker (sppw) on the results. We observe that the mean number of search nodes increases with the number of sppw. However, it remains quite close (548,201.4 vs 580,782.2 vs 686,760.9). Therefore, the best configuration is on mean sppw=100. However, it is not always the case for all the instances: gil262 is solved in 219s with sppw=100, 181s with sppw=200 and 221s with sppw=400. This result is quite interesting since decomposing more does not lead to improved results whereas a better decomposition leads to better results. Therefore, the way we decompose still have an impact on the results. However, a good improvement is observed for all the sppw.

Instances	Seq./limitBk=1k (1) time(s)	#sn	Seq./limitBk=5k (2) time(s)	#sn	Seq./limitBk=10k (3) time(s)	#sn
a280	1.3	1.1	0.9	2.8	0.9	2.3
ali535	3.9	1.3	3.3	1.1	3.4	1.2
d198	0.5	0.8	0.4	0.7	0.4	0.8
d493	0.8	0.5	0.7	0.5	0.9	0.6
gil262	9.1	1.9	4.1	1.3	3.7	1.2
gr229	0.8	0.3	0.9	0.3	1.0	0.5
gr431	3.7	1.1	2.3	1.1	1.9	0.9
gr666	3.2	0.7	3.7	0.9	3.3	0.9
kroA200	1.9	0.6	1.4	0.4	1.1	0.4
kroB150	0.7	0.4	0.6	0.2	0.5	0.3
kroB200	0.9	0.3	0.9	0.7	0.6	0.7
lin318	0.8	1.7	0.6	2.3	0.6	2.4
pcb442	11.1	3.5	6.5	6.0	10.0	8.0
pr136	1.3	0.7	0.9	1.1	0.8	1.4
pr264	0.3	0.2	0.3	0.2	0.3	0.2
pr299	0.7	0.6	2.2	0.9	2.2	0.8
rat195	1.7	1.1	0.8	0.9	0.5	0.6
rd400	3.8	1.1	7.3	1.1	5.8	1.0
si175	5.9	1.3	3.5	0.9	2.9	0.8
tsp225	1.2	0.4	0.7	0.4	0.6	0.4
mean	2.7	1.0	2.1	1.2	2.1	1.3
geo mean	1.7	0.8	1.4	0.8	1.3	0.8

Table 4.3: Ratio of solving time and number of search nodes. It is calculated by respectively dividing the results of a sequential execution with the results of *Bound-Backtrack-and-Dive* execution.

In order to increase the number of workers, the overall number of sub-problems generally must be increased. In Table 4.4, we show that the solving times are good for 400 sub-problems per workers and 8 workers, *i.e.* $3,200$ sub-problems. In EPS, communication times are negligible. Then, we observe results of the same order although not as good for up to 64 workers by setting sppw=50 (64 workers $\times$ 50 sub-problems equal 3,200 sub-problems). In the next section we will try to increase the number of workers to 192 in order to verify that.

4.3.2.4 General results

In Table 4.5, we compare the sequential solving, EPS without BBD and LCFirst for the decomposition step, and BBD with our best configuration. First, we observe that EPS without BBD allows obtaining a mean improvement of 2.5 in solving time and 0.9 for the search nodes compared with a sequential execution. The average number of search nodes gain shows a lack of robustness, many instances are solved with more search nodes in parallel than in sequential, and therefore does not exploit the set-up resources as desired. Thus, times can be bad with a naive EPS application at the TSP. Finally, *Bound-Backtrack-and-Dive* allows a better decomposition of the TSP by simulating LCFirst. Indeed, EPS with BBD obtain a mean improvement of 4.9 in solving time and 1.9 in for the search nodes. In the initial paper of *Bound-Backtrack-and-Dive* [Isoart and Régin, 2020b], we considered less filtering (only the WCC and the quadratic algorithm for the k-cutset constraint) and only 4 threads. We observed that for some instances (such as pcb442) the use of EPS noDiveLCDec increases the number of search nodes by a factor of 3. With the actual model, it is not the case: 14,852,219 search nodes in sequential vs 15,176,125 with EPS. It means that the more we have strong filtering algorithms the less the search strategy could be disrupted. Finally, the ex-

Instances	Diving time(s)	sppw=100 (1) time(s)	sppw=100 (1) #sn	sppw=200 (2) time(s)	sppw=200 (2) #sn	sppw=400 (3) time(s)	sppw=400 (3) #sn
a280	2.9	3.2	2,125	3.9	2,805	1.7	699
ali535	26.1	3,339.0	1,049,467	4,382.5	1,303,425	2,855.7	1,398,627
d198	3.8	9.5	3,935	9.9	4,339	9.9	4,131
d493	12.8	6,451.3	889,805	9,639.3	1,139,943	9,793.1	1,180,607
gil262	4.3	219.3	535,595	180.6	366,975	220.8	255,185
gr229	2.0	47.9	89,257	39.8	89,445	87.8	189,325
gr431	11.6	127.4	54,197	191.5	61,985	265.8	81,315
gr666	19.4	4,581.1	1,499,235	3,697.1	1,570,529	3,804.1	1,618,883
kroA200	2.1	31.6	55,345	43.3	71,133	74.5	73,977
kroB150	1.8	10.7	16,437	11.7	16,695	10.8	16,381
kroB200	1.8	14.6	27,743	35.7	32,847	28.5	31,083
lin318	3.8	4.0	1,055	4.0	1,065	4.3	951
pcb442	5.1	8,240.2	4,251,747	14,782.1	4,422,705	23,473.0	6,290,959
pr136	1.2	5.5	9,805	6.0	10,377	5.7	10,561
pr264	3.4	8.1	3,331	8.2	3,337	8.2	3,187
pr299	5.3	2,313.6	792,041	2,599.3	882,199	3,282.3	1,137,643
rat195	1.9	8.6	10,797	23.9	18,807	24.5	18,957
rd400	6.0	1,767.1	1,571,817	1,045.8	1,507,043	808.2	1,300,449
si175	1.5	15.7	70,297	20.8	66,769	26.4	75,109
tsp225	2.1	18.1	30,117	40.3	43,221	40.5	47,189
mean	5.9	1,360.8	548,207.4	1,838.3	580,782.2	2,241.3	686,760.9
geo mean	3.9	83.8	66,865.1	104.4	74,333.7	110.2	73,783.1

Table 4.4: Comparison of the number of sub-problems per worker (sppw) with limitBk=$1,000$.

periments are slightly different, but the conclusion is the same: the few nodes we perform during the decomposition are very important for the whole solving.

4.3.3 Conclusion

We have shown that the application of EPS to the TSP is not trivial. Indeed, EPS decomposition is breadth-based whereas TSP embeds LCFirst, a depth-based search strategy, so the two methods are incompatible. In order to combine the two approaches, we introduced *Bound-Backtrack-and-Dive*, a diving algorithm, which consists in a first step of performing a sequential execution with a bounded number of backtrack in order to study the behavior of LCFirst. Then, run EPS, simulate LCFirst during the decomposition using our preliminary study and finally solve with a classical LCFirst the generated sub-problems in parallel.

Experimental results show that the use of *Bound-Backtrack-and-Dive* allows obtaining robust results. Thus, the efficiency of parallelism applied to TSP with *Bound-Backtrack-and-Dive* allows a mean gain of a factor 4.9 in solving times and 1.9 in number of search nodes with 8 threads.

We think that this method can sometimes allow us to avoid dynamic learning strategies when it is an issue, here for the application of parallelism, and obtain a great improvement of solving times. We hope that similar results will be obtained for other learning search strategies.

	Sequential (1)		noDiveLCDec (2)		ratio (1)/(2)		BBD (3)			ratio (1)/(3)	
							Diving	Decomp and solve			
Instances	time(s)	#sn	time(s)	#sn	time	#sn	time(s)	time(s)	#sn	time	#sn
a280	8.1	2,351	4.2	2,473	1.9	1.0	2.9	3.2	2,125	1.3	1.1
ali535	13,013.9	1,339,925	18,563.7	1,742,497	0.7	0.8	26.1	3,339.0	1,049,467	3.9	1.3
d198	7.0	3,093	9.5	5,835	0.7	0.5	3.8	9.5	3,935	0.5	0.8
d493	4,989.9	488,439	4,485.1	637,677	1.1	0.8	12.8	6,451.3	889,805	0.8	0.5
gil262	2,042.2	1,040,943	1,514.1	1,271,733	1.3	0.8	4.3	219.3	535,595	9.1	1.9
gr229	41.2	26,565	32.2	27,551	1.3	1.0	2.0	47.9	89,257	0.8	0.3
gr431	520.6	60,537	294.6	40,851	1.8	1.5	11.6	127.4	54,197	3.7	1.1
gr666	14,719.6	1,106,125	5,533.8	1,244,949	2.7	0.9	19.4	4,581.1	1,499,235	3.2	0.7
kroA200	63.7	35,209	21.5	46,897	3.0	0.8	2.1	31.6	55,345	1.9	0.6
kroB150	8.7	6,165	6.1	11,059	1.4	0.6	1.8	10.7	16,437	0.7	0.4
kroB200	15.1	9,381	6.4	9,283	2.4	1.0	1.8	14.6	27,743	0.9	0.3
lin318	6.4	1,817	6.2	1,697	1.0	1.1	3.8	4.0	1,055	0.8	1.7
pcb442	91,137.7	14,852,219	17,919.2	15,176,125	5.1	1.0	5.1	8,240.2	4,251,747	11.1	3.5
pr136	8.9	7,229	3.5	7,905	2.5	0.9	1.2	5.5	9,805	1.3	0.7
pr264	3.8	575	10.5	2,797	0.4	0.2	3.4	8.1	3,331	0.3	0.2
pr299	1,596.5	489,039	2,341.1	753,791	0.7	0.6	5.3	2,313.6	792,041	0.7	0.6
rat195	17.9	11,601	16.1	19,673	1.1	0.6	1.9	8.6	10,797	1.7	1.1
rd400	6,721.0	1,727,475	3,398.2	1,849,117	2.0	0.9	6.0	1,767.1	1,571,817	3.8	1.1
si175	100.7	92,573	13.2	61,083	7.6	1.5	1.5	15.7	70,297	5.9	1.3
tsp225	23.5	11,557	13.0	12,439	1.8	0.9	2.1	18.1	30,117	1.2	0.4
mean	6,752.3	1,065,640.9	2,709.6	1,146,271.6			5.9	1,360.8	548,207.4		
geo mean	163.1	52,025.9	103.5	64,659.9			3.9	83.8	66,865.1		

Table 4.5: General Results. It shows the differences between sequential execution, naive EPS application and *Bound-Backtrack-and-Dive* with limitBk=$1,000$ and sppw=100.

4.4 Performance with a hundred cores

As it is usual in parallelism, one has the right to wonder if the scaling we observe for a few cores can be verified in practice with about a hundred cores. EPS has already been modified for improving its scaling while used on data centers [Régin et al., 2014]. However, the study of the behavior of EPS with a hundred cores for the solving of the TSP showed rather unexpected results. Two major issues were observed:

1. *Unstable decomposition*. The decomposition is no more stable in the sense that decomposing more may lead to a strong degradation of the performance. This means that the decomposition can interact unfavorably with the increase in the number of cores. In other words, increasing the number of cores can lead to worse results.

2. *Extremely heterogeneous sub-problems*. A very small number of sub-problems can take a very large part of the overall solving time. For instance, the decomposition of the problem gr431 of the TSPLIB [Reinelt, 1991] into 300 sub-problems, leads to 5 sub-problems which take 50% of the overall solving time. The consequence is that at the end of the solving, only 5 workers remain active, independently of the number of available workers.

In order to remedy the unstable decomposition issue, we suggest considering it carefully and avoiding decomposing too much. That is instead of trying to decompose in a lot of sub-problems, we suggest considering fewer sub-problems. The risk of reducing the number of sub-problems is that it can be difficult to ensure a good load balancing between the workers. However, when there

are a hundred workers it is less important to have 2, 5 or 10 workers that are not active than when you have only 8 workers. For instance, imagine that only 1 worker is working. If the total number of workers is 100, then 1% of the workers are working whereas if the total number of workers is 8, then 12.5% of the workers are working. On the other hand, the second problem we have to solve is that of extremely heterogeneous sub-problems which therefore lead *de facto* to bad load balancing.

In order to remedy the bad load balancing caused by the presence of extremely heterogeneous sub-problems, we have no choice but to re-decompose these sub-problems into many other sub-problems which are more homogeneous. In other words, we must be prepared to do several decomposition steps.

All this must be done without disturbing the functioning of EPS for problems that do not show these behaviors and that are very well solved by EPS even with a hundred cores. Therefore, systematically perform several decomposition steps worsens the results. One must identify if there are some extremely heterogeneous sub-problems and, in this case, prepare to restart a decomposition.

This also must be done while keeping the advantages of EPS: a very reduced communication (the workers do not communicate with each other) and independence from the solver used.

This approach will lead to many questions that can be summarized in the form of three main questions:

1. Under which conditions should we re-decompose?
2. How to avoid redoing some part of work already done when decomposing again?
3. What is the right number of sub-problems per worker to consider for a decomposition?

We will answer these questions in the next sections.

4.4.1 Re-decomposition

First, we try to estimate under which conditions it could be worthwhile to re-decompose the unsolved sub-problems. Then, we try to avoid redoing the same work when the solving of a sub-problem is interrupted in order to re-decompose.

The challenge can be resumed as follows: if we wait too long for performing a decomposition then the risk is that very few workers, or even one, will be left to solve a problem while all the others have no more work to do. However, re-decomposing for better distribution means losing part of the previous work already done and requires a certain minimum time that may not be profitable.

Notation 4.4.1 and Notation 4.4.2 describe some information about the search. Note that the values of Notation 4.4.1 are known and the values of Notation 4.4.2 are unknown.

Notation 4.4.1.

- w: total number of workers.
- a: number of active workers (*i.e.* workers which are currently solving a sub-problem).
- c_R: wall clock solving time of the set of the unsolved sub-problems R. Precisely, this is the solving time already done for the remaining sub-problems.

Notation 4.4.2.

- t_R: wall clock total solving time for the set of remaining sub-problems R. It is the sum of the solving times of the sub-problems not yet solved.
- d_R: wall clock time needed to decompose the set of remaining sub-problems R. We also called it the time to re-decompose.
- rt_R: wall clock time needed to solve the set of remaining sub-problems R after a re-decomposition.

We immediately have the following proposition:

Proposition 4.4.1. *The minimum remaining computation time is:* $\frac{t_R - c_R}{a}$

This is undeniably a lower bound of the real value because it assumes that the remaining time for the set of the remaining sub-problems is perfectly distributed among the workers if $a = w$ and if $a < w$ then it means that the remaining computation time for each sub-problem is the same.

Proposition 4.4.2. *The minimum remaining computation time for re-decomposing and solving the remaining sub-problems is:* $\frac{rt_R}{w} + d_R$

From these two propositions we introduce a new proposition:

Proposition 4.4.3. *Performing a re-decomposition may become worthwhile if*

$$\frac{rt_R}{w} + d_R < \frac{t_R - c_R}{a} \tag{4.1}$$

If after a decomposition no work of the stopped worker is recovered, then it means that $c_R = 0$ and $rt_R = t_R$. On the other hand, if we recover all the work already done, then we have $rt_R = t_R - c_R$. Unfortunately, it is very difficult to obtain this result because EPS is independent from the solver and from the search. Nevertheless, we can expect to have $rt_R < t_R$.

Simplifying this inequality is not simple because we are faced with several unknown variables. We do not know precisely t_R, nor rt_R, nor d_R and we have no guarantee that the solving process will be perfectly homogeneous.

Therefore, we suggest making some assumptions:

- We will consider that rt_R can be rewritten as $t_R - qc_R$ with $0 \leq q \leq 1$ and q corresponds to the proportion of work already done that we can recover. For practical reasons, we suggest to simply estimate the value of q from the previous calculations. Let us consider a sub-problem $p \in R$, we look for the value q_p. While solving p, if the search tree proved that k values can be safely removed from the initial domain of some variables, then we know that we were trying a $(k+1)^{th}$ value. Also, avoiding reconsidering the k^{th} values allow recovering $q_p = k/(k+1)$ from the calculations previously made. However if we look a little closer, we do not know where we were in the solving of the $(k+1)^{th}$ branch. On average, we can consider that we were halfway in the solving of this branch. It means that we will recover $q_p = k/(k+0.5)$ from the previous calculations. Thus, we suggest taking as q the average of the values of q_p for each sub-problem p.
- We also observed that the decomposition time does not vary that much. We will therefore consider that it is constant $d_R = d$. In practice, we will consider that a re-decomposition will take the time of the previous decomposition.

- Since no solving time is lost when $w = a$, we only consider a re-decomposition when $w > a$.

We can rewrite the previous proposition:

Proposition 4.4.4. *Performing a re-decomposition may become worthwhile if*

$$t_R > \frac{(w - aq)c_R + wad}{w - a} \tag{4.2}$$

Proof.

With Notation 4.4.1, Equation 4.1 can be rewritten as $\frac{t_R - qc_R}{w} + d < \frac{t_R - c_R}{a}$ which is equivalent to $(w - a)t_R > (w - aq)c_R + wad$. Since $w > a$ we have $(w - a) > 0$ and the proposition holds. □

The main issue is that we do not know the value of t_R. However, for a given value of a we can check whether Equation 4.2 is satisfied or not and stop the computation when it is satisfied. First, we know that $t_R \geq c_R$. Next, during a wall clock period of time T of computation performed by a active workers the total solving time computed is aT and so we have $t_R \geq aT + c_R$. Thus if $aT + c_R > \frac{(w-aq)c_R+wad}{w-a}$, then we know that Equation 4.2 is satisfied. So, we can compute the value of T for which Equation 4.2 is satisfied. This value will become the maximum timeout we accept without performing a new decomposition.

$$\begin{aligned}
aT + c_R > \tfrac{(w-aq)c_R+wad}{w-a} &\Leftrightarrow aT > \tfrac{(w-aq)c_R+wad}{w-a} - c_R \\
&\Leftrightarrow aT > \tfrac{(w-aq)c_R+wad-(w-a)c_R}{w-a} \\
&\Leftrightarrow aT > \tfrac{(a-aq)c_R+wad}{w-a} \\
&\Leftrightarrow aT > \tfrac{a(1-q)c_R+wad}{w-a} \\
&\Leftrightarrow T > \tfrac{(1-q)c_R+wd}{w-a}
\end{aligned}$$

Proposition 4.4.5. *Assume that after T units of computation no sub-problem is solved. It is worthwhile to re-decompose if*

$$T > \frac{(1 - q)c_R + wd}{w - a} \tag{4.3}$$

Proof.

If after $T_1 < T$ units of time we solve 1 task s_1, then we can compute the new inequality. We must consider decrement the number of active workers and work with $R_1 = R - \{s_1\}$ and $c_{R_1} = c_R + aT_1 - t(s_1)$. □

Equation 4.3 suggests that we should wait a certain amount of time to be sure that we should decompose. However, waiting that time may have a cost that leads to poorer results. Thus, we suggest using T as an upper bound of the waiting time before re-decomposing.

Indeed, on the one hand, we do not know how to recover all the calculations performed (we do not recover c_r, but only qc_r). On the other hand, we do not calculate with the power of all the workers, which means that we lose a certain amount if the decision is to decompose again. However, if we re-decompose too early when we should not have, we will also lose time. Therefore, instead of directly using the formula in Equation 4.3, it is more interesting to introduce a decision factor α, with $0 \leq \alpha \leq 1$. The idea behind this value corresponds to a kind of risk sharing.

We have the final proposition:

Proposition 4.4.6. *Assume that after T units of computation no sub-problem is solved. The current active workers are stopped and decomposition is performed if*

$$T > \alpha \frac{(1-q)c_R + wd}{w-a} \tag{4.4}$$

This leads to Algorithm 12. Equation 4.4 will evolve as sub-problems are solved. This will result in a sequence of inequalities, one for each solved sub-problem. Moreover, we need to make sure that each inequality holds. Indeed, when a problem p of R is solved, we compute a new inequality of Equation 4.4 for $R - \{p\}$. However, the time between the last task solved and p can be huge. Therefore, the new timeout must verify the old inequality (*i.e.* $timeout - T$) and the new computed inequality. Thus, we take $\min(timeout - T, \alpha\frac{(1-q)c_R+wd}{w-a})$ as the new timeout. Note that Function checkDecomposition(d, α, q) is run in the main loop of the EPS master which manages the workers and the sub-problems. If it returns true, then a re-decomposition is performed.

Algorithm 12: checkDecomposition algorithm

checkDecomposition (d, α, q)
- **Input:** A constant decomposition time d, a decision factor α and the proportion of work already done that we can recover q
- **Output:** A boolean specifying whether a re-decomposition must be performed
- `// The sub-problems are currently run in parallel`
- Wait until it remains only $a = w - 1$ active workers ;
- $stopTime \leftarrow$wallClockTime() ;
- $R \leftarrow$ set of remaining sub-problems ;
- $C \leftarrow$ Sum of current solving times of the remaining sub-problems ;
- $timeout \leftarrow \alpha\frac{(1-q)C+wd}{w-a}$;
- **foreach** *sub-problem* $p \in R$ *solved before* $timeout$ **do**
  - `// the timeout evolves according to the solved sub-problems`
  - $R \leftarrow R - \{p\}$;
  - $T \leftarrow$ WALLCLOCKTIME() $- stopTime$;
  - $stopTime \leftarrow$wallClockTime() ;
  - `//` $t(p)$ `is the solving time of the sub-problem` p
  - $C \leftarrow C + aT - t(p)$;
  - $a \leftarrow a - 1$;
  - $timeout \leftarrow \min(timeout - T, \alpha\frac{(1-q)C+wd}{w-a})$;
- **if** *all sub-problems are solved* **then return** *False*;
- Abort all the remaining active workers. ;
- `// A re-decomposition will be performed`
- **return** *True*

4.4.1.1 Recovery of previous computations

When a worker is aborted, it has already performed some work. How can we avoid redoing the same work that has just been done?

EPS is based on the concept of independence. EPS always tries to be as independent as possible from solvers, from the search procedure, at least intrusive as possible and to minimize communications. In order to retrieve part of the previously performed calculations, we could try to memorize the boundary of the search already made. The problem with this approach is that this boundary strongly depends on the search strategy. For a DFS the boundary is in $O(n)$, but for other searches like a Best-First Search it is more complex. In addition, with learning search strategies it would be necessary to transmit much more information to be able to continue/resume a search after a stop. Finally, each solver can have its own implementation of the search and resuming a stopped search should be a function provided by the solver and used by EPS. The communication may become specific for each solver and for each search, which is precisely what we want to avoid.

We introduce a simple and general method that requires little intervention in the solver and practically no more transmission of information. This method only assumes that the search can be described by a set of decisions and rejections of decisions (which is most often the case). Conceptually, it is usual to see the search tree as a binary tree where a node has two children: one applies the decision and the other rejects the decision (usually by imposing the opposite). When a sub-tree resulting from a decision is completely finished, then we know that the exploration of that part is completely finished, and we do not need to do it again. We cannot keep all the nodes closed, but we can simply no longer consider the nodes linked to the root of which only the refutation of the decision remains. In this case we know that the decision has been fully processed and that it is not necessary to redo it. This is easy to express because it simply corresponds to the suppression of a value in the domain of the first assigned variable. In the case of TSP, it corresponds to a set of edges that should no longer be used for the tour. In order not to redo all the work done the worker just has to transmit this information to the EPS master. It can be noticed that it is very general and not very intrusive in a solver.

4.4.1.2 Discussion

Instead of an overall decomposition, it is also possible to study the behavior of EPS at the level of each of the sub-problems. Thus, one can try to define criteria to try to know whether a sub-problem should or should not be re-decomposed and then proceed to a specific individual re-decomposition for some sub-problems. We tried this method, but it is not advantageous and raises many questions: when should we look to see if the sub-problems are particularly difficult? Should they be stopped immediately? Moreover, this method leads to a succession of decompositions (one per sub-problem) which quickly becomes expensive and does not bring any particular gain. In fact, EPS always tries to consider the solving of sub-problems as global and there seems to be little interest in questioning this point of view.

In addition, only the master can use such a criterion because we search for sub-problems that are more difficult than others and only the master can have this information. A worker has no information about the other workers. Therefore, it is preferable to let the master manage the solving of the different workers, even if it means interrupting some of them.

4.4.2 Experiments

In this section, we will experimentally show that EPSrd allow obtaining better solving times and more robustness. To do so, we will experiment on the selected instances from Section 4.3.2. In Section 1.1, we said the experiments are performed on Clear Linux with two Intel Xeon E5-2696v2 (12 cores and 24 threads per CPU) and 64 GB of RAM. In this section, the experiments are run in parallel on 4 machines with the exact same configuration. Therefore, we have 4 machines with two CPUs and 24 threads per CPU. We will fully load the 4 machines; therefore the experiments of this section are run on 192 workers.

First, we will search for the best configuration for EPS and EPSrd. In Table 4.6, we consider the number of sub-problems per workers (#sppw). We observe that #sppw=100 for EPS and #sppw=10 for EPSrd are the best configurations in order to obtain the best mean solving times. Note that we used $\alpha = 0.1$ for EPSrd. We observe that most of the time EPSrd improves the results of EPS. EPS is based on the idea that for having a good load balancing we need to have a certain number of sub-problems. The experiments support this idea, even if sometimes decomposing too much may be time consuming. For instance, pcb442 is solved in 22,800s with #sppw=10, 11,025s with #sppw=30, 7,686s with #sppw=50 and 5,168.8s with #sppw=100. However, if we carefully look at the result, we can refine this principle. We observe that when the load balancing is not an issue for a problem (such as gr229 where #sppw=10 is the best), then a small value of #sppw is perfectly fine. This means that we need to have a greater value only when the load balancing is an issue. However, we have no information about the load balancing before the solving, so we need to use set a value of #sppw between 10 and 100 before the solving. Conversely, EPSrd tries to identify during the solving whether the load balancing is an issue or not. If it detects a load balancing issue, then it performs a re-decomposition. Therefore, EPSrd has advantage to start with a low #sppw. Note that this can be seen as a dynamic increase of the #sspw value. Therefore, we observe that the mean solving times with #sppw=10 is 2.3 times faster. In addition, the problems with a huge load balancing issue such as pcb442 are well solved with EPSrd. Indeed, the best solving time on this instance for EPS is 5,166.8s and 159.5s for EPSrd. Finally, we observe that the best configuration for EPSrd in this table is 5.3 times faster than the best one for EPS.

In Table 4.7, we consider the α parameter for EPSrd with #sppw=10. We note s.d. the standard deviation of the solving times for an instance. First, we notice that the importance of the alpha value relies on the number of re-decompositions. For instance, gr229 is re-decomposed between 0 and 1 time for each alpha value. Then, all the solving times are close to each other, and the standard deviation is quite low (2.2). Conversely, the standard deviation for gr666 is equal to 54.7 and we have between 5 and 11 re-decompositions for each alpha value. Then, the more the number of re-decompositions is high the more the alpha value is important. Nevertheless, the alpha value does not have an impact as important as sppw on the solving times. Finally, considering the mean, $\alpha = 0.3$ is the best alpha value whereas considering the geometric mean $\alpha = \{0.1, 0.5, 0.7\}$ are the best alpha values.

In Table 4.8, we study the behavior of EPS and EPSrd when the best configuration is set. That is #sppw=100 for EPS and #sppw=10 and alpha=0.1 for EPSrd. This experiment shows some drawbacks of EPS for some problems. For EPS, we give some wall clock times (in s): the decomposition time, the wall clock time of the solving time when all workers are active, the wall clock time of the solving time when some (or most) workers are inactive and the wall clock time to solve the problem (total). Note that the wall clock time is the sum of decomp., all workers are active and not all workers are active. Next, we give some information about EPSrd: the

Instances	EPS(s)				EPSrd(s)				EPS/EPSrd			
	#sppw				#sppw				#sppw			
	10	30	50	100	10	30	50	100	10	30	50	100
a280	1.8	9.8	17.4	1.7	1.6	1.6	18.3	1.6	1.1	6.2	0.9	1.0
ali535	3,436.0	4,045.6	3,945.4	4,056.3	463.9	1,108.1	560.5	617.5	7.4	3.7	7.0	6.6
d198	12.2	14.6	15.2	14.5	13.3	12.9	12.6	13.5	0.9	1.1	1.2	1.1
d493	4,188.8	4,426.0	3,810.6	2,247.2	1,084.5	1,449.3	2,259.1	3,708.3	3.9	3.1	1.7	0.6
gil262	75.7	79.1	68.9	113.6	30.6	88.5	75.5	106.9	2.5	0.9	0.9	1.1
gr229	12.4	20.3	19.7	41.4	16.2	30.0	35.1	41.3	0.8	0.7	0.6	1.0
gr431	77.6	82.2	105.3	169.4	82.6	91.8	106.3	168.3	0.9	0.9	1.0	1.0
gr666	4,588.5	1,653.5	1,749.0	1,555.2	926.0	998.3	1,122.8	1,075.8	5.0	1.7	1.6	1.4
kroA200	6.4	6.2	7.5	6.1	7.3	7.1	7.4	6.4	0.9	0.9	1.0	0.9
kroB150	2.5	2.3	2.6	2.5	2.6	2.6	2.4	2.5	1.0	0.9	1.1	1.0
kroB200	6.2	5.2	4.4	4.4	6.6	5.6	4.7	4.5	0.9	0.9	0.9	1.0
lin318	3.6	3.9	3.4	3.6	3.4	3.3	3.3	3.7	1.1	1.2	1.0	1.0
pcb442	22,800.1	11,025.4	7,686.4	5,166.8	159.5	289.7	408.9	331.0	143.0	38.1	18.8	15.6
pr136	1.4	1.6	1.8	1.7	1.7	1.6	1.7	1.6	0.8	1.0	1.1	1.1
pr264	11.9	13.8	13.9	12.8	16.5	13.3	13.1	13.8	0.7	1.0	1.1	0.9
pr299	1,345.2	2,183.9	3,807.3	3,160.6	211.3	220.8	303.4	396.8	6.4	9.9	12.6	8.0
rat195	5.9	5.1	5.1	4.8	5.7	4.9	4.4	4.3	1.0	1.0	1.1	1.1
rd400	706.2	496.2	477.7	560.9	135.9	215.9	244.8	816.1	5.2	2.3	2.0	0.7
si175	4.6	6.9	12.2	15.0	5.0	6.7	10.6	13.5	0.9	1.0	1.2	1.1
tsp225	10.5	9.0	8.3	8.8	9.6	8.2	8.1	8.0	1.1	1.1	1.0	1.1
mean	1,864.9	1,204.5	1,088.1	857.4	159.2	228.0	260.1	366.8				

Table 4.6: Comparison between EPS and EPSrd for different values of #sppw.

decomposition time, the wall clock time performed between the end of the first decomposition and the start of the first re-decomposition, the wall clock time of the solving time when all workers are active, the wall clock time of the solving time when some (or most) workers are inactive and the wall clock time to solve the problem (total). In this table, we can see that EPSrd allows obtaining a better load balancing. For instance, with EPS gr666 spend 304.8s with all the workers active and 1,101s without all the workers active. Conversely, with EPSrd gr666 spend 401.6s with all the workers active and 142.9s without all the workers active. Then, when the load balancing is going to be bad, EPSrd quickly performs a first re-decomposition avoiding spending a lot of time with some inactive workers.

In Table 4.9, we show the general results obtained by EPS and EPSrd with their best configurations. We recall that we use 192 workers. We first notice that both EPS and EPSrd allow improving the mean solving times. Indeed, a gain of a factor 5.5 is observed with EPS and a gain of 40.3 is observed with EPSrd compared to the mean sequential solving times. Therefore, EPSrd is much more efficient than EPS: it is 7.3 times faster and the mean improvement ratio is of 4.2. We recall that it is very hard in parallelism to obtain such an improvement equal to the number of workers since in many cases we must perform a warm-up (here, the diving and the decomposition). In addition, the perfect load balancing is quite hard to obtain since either we must know how to perfectly decompose a sub-problem with no overhead (we do not) or we use a work stealing approach that can lead to huge communication times (the purpose of EPS is to avoid the communication times as much as possible).

In Table 4.10, we compare the robustness of EPS and EPSrd. We run several times each instance and we compare the minimum solving time, the maximum solving time, the mean solving time and the standard deviation of the solving times. Note that the best configuration for both EPS

	α						
	0.1	0.3	0.5	0.7	0.9	#re-decomp.	s.d.
a280	1.6	5.2	1.6	1.5	1.5	0	1.6
ali535	463.9	415.2	467.9	471.1	500.0	4 to 7	30.6
d198	13.3	14.5	16.0	12.9	13.2	0 to 2	1.3
d493	1,084.5	1,033.7	1,135.9	1,048.8	1,101.6	6 to 11	41.0
gil262	30.6	26.9	30.0	34.0	31.4	1 to 3	2.6
gr229	16.2	16.4	12.8	11.8	12.6	0 to 1	2.2
gr431	82.6	95.1	86.1	87.2	94.7	1 to 4	5.5
gr666	926.0	822.5	786.9	807.9	810.8	5 to 11	54.7
kroA200	7.3	7.2	7.0	6.8	6.7	0	0.3
kroB150	2.6	2.7	2.7	2.6	2.8	0	0.1
kroB200	6.6	5.6	4.9	6.2	6.7	0	0.8
lin318	3.4	3.3	4.0	3.2	3.5	0	0.3
pcb442	159.5	210.6	243.1	270.4	314.2	5 to 8	58.7
pr136	1.7	1.4	1.8	1.5	1.6	0 to 1	0.2
pr264	16.5	12.0	11.6	11.7	12.9	0 to 1	2.0
pr299	211.3	237.4	236.3	248.3	317.4	8 to 14	40.0
rat195	5.7	5.9	5.0	5.9	6.3	0	0.5
rd400	135.9	132.5	134.0	135.7	140.4	4 to 6	3.0
si175	5.0	5.2	5.5	6.2	4.9	0 to 1	0.5
tsp225	9.6	9.7	10.5	10.4	10.2	0	0.4
mean	159.2	153.1	160.2	159.2	169.7		6.0
geo mean	24.5	25.5	24.5	24.5	25.5		

Table 4.7: Impact of the alpha value on the solving times in seconds. We note s.d. the standard deviation of the solving times.

	EPS time(s)				EPSrd time(s)				
Instances	decomp.	all workers are active	not all workers are active	total	decomp.	before first re-decomp.	all workers are active	not all workers are active	total
a280	1.3	0.0	0.3	1.7	1.3	0.3	0.0	0.3	1.6
ali535	30.1	3.7	4,022.5	4,056.3	141.6	9.9	238.9	83.3	463.9
d198	9.3	1.0	4.2	14.5	9.4	3.4	2.4	1.5	13.3
d493	135.7	8.6	2,102.9	2,247.2	178.5	16.6	801.4	104.6	1,084.5
gil262	105.4	6.7	1.5	113.6	17.8	3.8	6.5	6.3	30.6
gr229	20.7	19.2	1.4	41.4	10.8	4.4	3.9	1.6	16.2
gr431	146.4	5.9	17.1	169.4	63.8	8.8	6.8	12.0	82.6
gr666	149.4	304.8	1,101.0	1,555.2	381.5	56.8	401.6	142.9	926.0
kroA200	5.6	0.3	0.1	6.1	6.7	0.6	0.5	0.1	7.3
kroB150	2.3	0.0	0.2	2.5	2.3	0.2	0.1	0.2	2.6
kroB200	4.2	0.2	0.1	4.4	6.1	0.5	0.2	0.3	6.6
lin318	3.4	0.0	0.3	3.6	3.3	0.1	0.0	0.1	3.4
pcb442	80.9	58.5	5,027.4	5,166.8	51.2	18.2	67.8	40.4	159.5
pr136	1.5	0.1	0.2	1.7	1.0	0.7	0.1	0.6	1.7
pr264	10.4	1.1	1.4	12.8	12.0	4.0	2.8	1.7	16.5
pr299	112.9	5.8	3,041.9	3,160.6	80.6	5.4	85.3	45.4	211.3
rat195	4.6	0.1	0.1	4.8	5.5	0.2	0.1	0.2	5.7
rd400	241.2	25.0	294.6	560.9	60.4	17.0	50.5	24.9	135.9
si175	12.7	0.7	1.7	15.0	3.3	1.6	0.9	0.8	5.0
tsp225	8.0	0.2	0.5	8.8	9.0	0.7	0.2	0.4	9.6

Table 4.8: Solving evolution of EPS and EPSrd.

Instances	Sequential(s)	EPS(s)	Seq./EPS	EPSrd(s)	Seq./EPSrd	EPS/EPSrd
a280	8.1	1.7	4.8	1.6	5.0	1.0
ali535	13,013.9	4,056.3	3.2	463.9	28.1	8.7
d198	7.0	14.5	0.5	13.3	0.5	1.1
d493	4,989.9	2,247.2	2.2	1,084.5	4.6	2.1
gil262	2,042.2	113.6	18.0	30.6	66.7	3.7
gr229	41.2	41.4	1.0	16.2	2.5	2.6
gr431	520.6	169.4	3.1	82.6	6.3	2.0
gr666	14,719.6	1,555.2	9.5	926.0	15.9	1.7
kroA200	63.7	6.1	10.5	7.3	8.7	0.8
kroB150	8.7	2.5	3.5	2.6	3.3	1.0
kroB200	15.1	4.4	3.4	6.6	2.3	0.7
lin318	6.4	3.6	1.8	3.4	1.9	1.1
pcb442	91,137.7	5,166.8	17.6	159.5	571.5	32.4
pr136	8.9	1.7	5.1	1.7	5.2	1.0
pr264	3.8	12.8	0.3	16.5	0.2	0.8
pr299	1,596.5	3,160.6	0.5	211.3	7.6	15.0
rat195	17.9	4.8	3.7	5.7	3.1	0.8
rd400	6,721.0	560.9	12.0	135.9	49.5	4.1
si175	100.7	15.0	6.7	5.0	20.2	3.0
tsp225	23.5	8.8	2.7	9.6	2.4	0.9
mean	6,752.3	857.4	5.5	159.2	40.3	4.2

Table 4.9: Comparison between EPS and EPSrd.

	EPS					EPSrd				
	min	max	mean	s.d.	max/min	min	max	mean	s.d.	max/min
a280	1.5	2.0	1.7	0.2	1.3	1.6	1.9	1.7	0.1	1.2
ali535	3,827.2	4,400.1	4,072.5	241.2	1.1	447.5	561.7	478.3	47.0	1.3
d198	13.4	40.0	19.2	11.6	3.0	13.1	14.7	13.8	0.6	1.1
d493	2,247.2	4,784.6	3,229.2	1,175.3	2.1	858.0	1,092.2	979.8	104.8	1.3
gil262	106.0	125.3	115.3	7.0	1.2	28.1	35.1	30.9	2.8	1.2
gr229	39.6	41.4	40.9	0.7	1.0	16.2	18.0	17.0	0.7	1.1
gr431	169.4	180.8	173.8	4.4	1.1	78.9	84.8	82.2	2.6	1.1
gr666	1,003.2	1,555.2	1,180.0	229.1	1.6	854.8	930.0	904.2	30.1	1.1
kroA200	6.1	8.1	6.6	0.9	1.3	5.9	7.3	6.6	0.7	1.2
kroB150	2.5	2.6	2.5	0.0	1.0	2.5	3.0	2.7	0.2	1.2
kroB200	4.2	4.6	4.4	0.1	1.1	5.4	6.6	6.0	0.4	1.2
lin318	3.4	3.6	3.5	0.1	1.1	3.3	4.2	3.5	0.4	1.3
pcb442	3,288.8	23,783.8	11,776.5	8,244.6	7.2	159.5	195.1	180.8	16.4	1.2
pr136	1.7	1.7	1.7	0.0	1.0	1.3	1.7	1.5	0.2	1.3
pr264	12.8	14.2	13.6	0.5	1.1	15.3	17.5	16.3	0.8	1.1
pr299	1,412.1	3,160.6	2,222.7	656.9	2.2	189.9	211.3	198.8	10.1	1.1
rat195	4.5	4.9	4.8	0.2	1.1	5.3	6.4	5.7	0.4	1.2
rd400	560.9	759.9	644.9	74.7	1.4	130.6	142.3	137.6	4.5	1.1

Table 4.10: Comparison of the robustness of EPS and EPSrd.

and EPSrd are used. For EPS, most instances have quite a large variation between min and max and therefore a huge standard deviation. For instance, with EPS the min and the max solving times of gr666 are respectively 1,003s and 1,555s. It leads to a standard deviation of 229.1. Moreover, the max solving time is 55% slower than the min solving time. With EPSrd, the min and the max solving times of gr666 are respectively 855s and 930s with a standard deviation of 30.1. Then, the max solving time is 9% slower than the min solving time. Finally, the ratio of max/min for EPS is between 1.0 and 7.2 whereas it is between 1.1 and 1.3 for EPSrd. Then, EPSrd bring more robustness in the solving times than EPS.

4.4.2.1 Increasing the number of sub-problems for the re-decompositions

At last, we tried to increase the number of sub-problems each time a new re-decomposition is performed. We expected to reduce the number of re-decompositions and therefore to reduce the wall clock solving times. Unfortunately, this is not the case. We obtained a reduction of the number of re-decompositions (reduction about a factor of 2) with a slight increase of the wall clock solving times. The advantage of the re-decomposition with a low #sppw is that the re-decompositions are fast and therefore a re-decomposition does not cost that much. Increasing #sppw each time a re-decomposition is performed lead to slower re-decompositions. In addition, it does not bring a better load balancing since in the TSP there are only few problems that are very slow to solve. Then, it is worthwhile to stick with a low #sppw and wait for the identification of the hardest sub-problems to solve in order to re-decompose them.

4.4.3 Conclusion

In this section, we focused on the parallel solving of difficult TSP problems with EPS and many cores. More precisely, we were interested in two issues: unstable decomposition and extremely heterogeneous sub-problems. We have shown that EPSrd, which uses a re-decomposition, improves EPS by solving these two issues. Indeed, a re-decomposition allows the use of a small first decomposition. In addition, it allows avoiding the case of too small proportion of workers working because of some extremely heterogeneous problems. Experimentally, we have shown that a small number of sub-problems per worker allows obtaining the best results when using EPSrd because the load balancing is managed by the re-decomposition. It leads to an overall improvement of the solving times for EPSrd and much more robustness.

CHAPTER 5

Shaving

In this chapter, we experiment with a shaving method. We show that it reduces the number of search nodes by several orders of magnitude. It consists in selecting some pairs (variable, value) and assigning them successively. If an immediate failure occurs, then the assigned pair cannot belong to any solution of the current sub-tree of the search tree. In practice, this method reduces the number of search nodes by several orders of magnitude. However, the direct application of this method is time consuming. Thus, we introduce some variations of this method in order to find a better trade-off between the search space pruning and the solving time. Although the use of shaving does not allow solving instances faster, it is nevertheless very interesting. Indeed, few methods allow obtaining a reduction of several orders of magnitude of the search space. Thus, this chapter allows us to share a very interesting way attacking a hard problem.

5.1 Introduction

In order to reduce the number of search nodes, we are interested in a higher level of consistency than the arc consistency: the singleton arc consistency (SAC) [Debruyne and Bessiere, 1997]. Given an arc consistent constraint network $\mathcal{N}$, for each pair of variable and value (x_i, a) of $\mathcal{N}$, if $\mathcal{N}$ is arc inconsistent when assigning a to x_i, then a can be safely removed from x_i. The advantage of this approach is that the filtering is much stronger than a simple arc consistency and thus it reduces the number of search nodes. However, testing all (x_i, a) pairs costs much more than a simple arc consistency. In this chapter, we will study this level of consistency for the TSP. More formally, we define SAC in Definition 5.1.1.

Definition 5.1.1 (Singleton arc consistency)**.** A constraint network $\mathcal{N} = (X, D, C)$ is singleton arc consistent iff $\forall x_i \in X, \forall a \in D(x_i)$ the constraint network $\mathcal{N}'$ such that $D(x_i) = \{a\}$ is arc consistent.

However, the CP model of the TSP is composed of a graph variable represented by two set-variables: one for the edges, one for the nodes. Note that the search strategy only considers the edges since all the nodes are mandatory for the TSP problem. By definition, an assignment of a value a to a set-variable x_i is equivalent to adding a to $lb(x_i)$. Then, we redefine Definition 5.1.1 in Definition 5.1.2.

Definition 5.1.2 (Singleton arc consistency on set-variables)**.** Given a constraint network $\mathcal{N} = (X, D, C)$ such that X is composed of set-variables only. Then, $\mathcal{N}$ is singleton arc consistent iff $\forall x_i \in X, \forall a \in ub(x_i) - lb(x_i)$ the constraint network $\mathcal{N}'$ such that $lb(x_i) = lb(x_i) \cup \{a\}$ is arc consistent.

In other words, we try to assign successively all the optional edges of the graph variable. If an assignment leads to an inconsistency, then we remove the optional edge from the graph.

Moreover, this idea has been used in scheduling problems [Carlier and Pinson, 1994, Martin and Shmoys, 1996] in order to find the position of some tasks (or at least generate constraints) and to prune the search tree. Usually, they do not enforce SAC but they try some elements. We will use this approach because testing all the values for all the variables is sometimes time consuming.

This process is sometimes named “shaving”. Since we will not necessarily enforce SAC, we will use this terminology.

5.2 Experiments

This chapter is mainly experimental since the general shaving method is already known. In practice, given a search node, the shaving is performed before the propagation over the search node. When an optional edge is removed by the shaving, we say that the edge is shaved. We define “candidates” as the set of optional edges to be assigned in the shaving. The results are represented in tables. For each table, we give the time in seconds, the number of search nodes, the number of assigned edges in the shaving (#check) and the number of its edges that have been shaved (#shaved). In addition, we give the arithmetic mean and the geometric mean for each parameter. We have chosen a subset of the instances considered in this thesis: the easy instances. Indeed, the hardest instances are very long to solve with shaving. Moreover, the conclusions are the same.

5.2.1 Check mode

There are two ways of considering assignments in the shaving. We can assign each edge of candidates only once per search node (check: once) or iterate on candidates until a fixed point over the shaved edges is reached (check: fixed point). Therefore, if candidates contains all the optional edges and the check mode fixed point is used, then SAC is achieved.

In Table 5.1, we compare the check mode once and fixed point. First, in comparison of the check mode fixed point, the check mode once is faster by a factor of 2.2 in mean and by a factor 1.4 in geometric mean. However, the check mode fixed point leads to 1.3 times fewer search nodes on mean and 1.6 times fewer search nodes on geometric mean.

The difference is easily explained with the columns #check and #shaved. Indeed, with the check mode once, the mean of #check is 24,202.6 for #shaved=2,875.4 while with the check mode fixed point the mean of #check is 59,823.1 for #shaved=2,533. Thus, the check mode fixed point checks 2.4 more elements than the check mode once and shave a similar number of elements.

Then, the shaving with the check mode once loss a mean (resp. geo mean) solving time factor of 6.7 (resp. 3.6) compared to the state-of-the-art (*i.e.* without the shaving). Nevertheless, it allows us to reduce the number of search nodes by a factor of 100.3 in mean and by a factor of 60.3 in geometric mean.

Thus, the shaving is quite an interesting method for the TSP because a single level of higher consistency leads to a huge reduction of the search space (two orders of magnitude).

Without loss of generality, we will only consider the check mode once.

5.2.2 Candidates

In this section, we consider different sets of candidates (the set of optional edges to be assigned in the shaving). Indeed, considering all the optional edges of the graph for the shaving is not mandatory. Thus, we define three modes of selection:

- 1-mand: selects the optional edges having at least one end with an adjacent edge being mandatory.
- 2-mand: selects optional edges having both ends with an adjacent edge being mandatory.
- LCNode: selects the optional edges around the LCNode if a backtrack has just occurred or if the LCNode has just been changed.

In Table 5.2, we compare 1-mand, 2-mand and LCNode. For 1-mand and 2-mand, we notice that the mean solving time is quite close (32.8 and 34.3) while the number of search nodes is very different (261.1 and 2,357.8). Looking at the geometric mean, we notice that 1-mand does 53.8 search nodes and 1,328.5 checks (*i.e.* 1,382.3 propagations are triggered) and that 2-mand does 308.2 search nodes and 812.8 checks (*i.e.* 1,130 propagations are triggered). This represents 22% more propagations for 1-mand and 23% more in solving time. In addition, most instances are solved faster with 2-mand than with 1-mand while 2-mand makes 9 times (resp. 5.7) more search nodes on mean (resp. geometric mean). Next, we can see that the mode LCNode reduces the mean solving time but increases the number of search nodes. In the shaving, the more the number of checked edges is high, the more the number of search nodes is reduced and the solving time is increased. Finally, LCNode allows us to have solving times close to the state-of-the-art without the shaving method. Additionally, we obtain a gain of a factor of 1.7 on mean and geometric mean

	basic model		check: once				check: fixed point			
Instance	time	search nodes	time	search nodes	#check	#shaved	time	search nodes	#check	#shaved
a280	8.4	2,351	**12.0**	37	2,522	773	13.6	37	3,698	738
bier127	0.4	53	**3.4**	3	760	350	4.1	1	827	388
brg180	1.7	101	**369.6**	87	43,814	201	396.0	85	45,177	202
ch130	1.3	383	2.2	7	654	284	**2.1**	5	624	267
ch150	1.3	303	**2.1**	3	461	242	2.5	3	465	242
d198	8	3,093	**30.3**	57	12,826	2,237	61.7	29	28,298	1,841
eil101	0.3	65	0.9	15	190	34	**0.7**	15	225	34
gr120	1	279	1.4	1	243	118	**1.2**	1	243	118
gr137	1.5	513	**10.9**	29	6,129	777	16.5	7	11,267	694
gr202	1.8	383	**3.7**	5	821	371	**3.7**	1	839	364
gr229	41.3	26,565	**356.6**	269	126,568	17,109	1,102.5	181	412,219	16,379
gr96	0.8	227	**3.9**	19	2,399	324	7	15	5,055	322
kroA100	1.3	483	**6.7**	25	4,571	665	10.4	13	10,315	472
kroA150	2.9	1,269	**4.7**	5	1,765	479	5.8	1	2,423	443
kroA200	63.2	35,209	**158.3**	81	48,046	5,038	421.6	65	128,725	6,338
kroB100	1.7	1,045	**4.8**	15	2,695	527	7.6	7	5,600	474
kroB150	9.2	6,165	**136.8**	201	70,075	8,662	296.5	103	166,759	6,718
kroB200	14.7	9,381	**58.0**	51	22,487	3,360	159.9	39	72,867	3,090
kroC100	0.8	233	**1.6**	3	415	222	**1.6**	1	415	222
kroD100	0.6	149	**0.5**	1	78	45	**0.5**	1	78	45
kroE100	1.4	641	**3.2**	9	1,721	404	6.2	7	3,725	337
lin318	7.4	1,817	**19.1**	7	3,416	975	50.8	3	13,649	1,027
pr124	2	827	**3.4**	3	931	456	4.5	1	1,309	557
pr136	10	7,229	**72.7**	343	58,952	6,349	151.1	249	131,984	5,879
pr144	1.4	345	2.2	1	399	240	**2.1**	1	399	240
pr264	4.7	575	**7.2**	1	385	211	**7.2**	1	385	211
rat195	17.4	11,601	**213.3**	213	86,666	9,329	416.9	109	171,763	6,935
rat99	0.3	61	0.4	1	60	33	**0.3**	1	60	33
si175	100	92,573	**120.8**	303	79,904	9,223	324.9	365	243,729	10,065
st70	0.3	65	**0.3**	9	53	8	**0.3**	9	69	8
tsp225	23.5	11,557	**598.8**	339	194,266	22,860	1,372.2	225	450,928	16,907
u159	0.6	133	**1.4**	7	210	106	1.7	7	221	106
mean	10.3	6,739.8	**69.1**	67.2	24,202.6	2,875.4	151.7	49.6	59,823.1	2,553.0
geo mean	2.8	904.1	**9.4**	15.1	2,686.7	588.1	13.5	9.6	4,310.5	560.6

Table 5.1: Comparison with our CP model (WCC + k-cutset constraint + mandatory Hamiltonian path constraint + one-tree constraint, named "basic model") and the shaving method with the check mode once and fixed point.

on the number of search nodes. Unfortunately, this is still far from the factor 100 that we obtained by checking all the optional edges of the graph.

5.2.3 Calling mode

In this section, we will study the calling mode which is the type of search node for which the shaving is triggered. In a search tree we can assign a value to a variable (Apply) and we can remove a value from the domain of a variable (Refute). Thus, we consider three modes: Apply, Refute and ApplyAndRefute that run the shaving to both the search nodes Apply and Refute. In Table 5.3, we notice that the mean number of search nodes is very close for Apply and for Refute. However, it is reduced by about a factor of 2 for ApplyAndRefute. This can be simply explained by the fact that the combination of the two avoids missing important domain reductions. We note that the calling mode Refute is the fastest here. This is mainly due to brg180 which is solved 16 times faster. If we look more closely, very few edges are shaved for this instance and this instance has very few refute search node. Thus, we get a much better trade-off by using the shaving for Refute search nodes for this instance.

In Table 5.4, we observe similar results to those in Table 5.3. However, we notice that the callingMode Apply is clearly more efficient in terms of the number of search nodes. As for Table 5.3, ApplyAndRefute tends to be slower but to make fewer search nodes.

In Table 5.5, we notice that we check fewer edges than in Table 5.4 and Table 5.3. Thus, we observe very close solving times for the three different calling modes. However, we also notice that ApplyAndRefute allows us to reduce the number of search nodes.

Finally, if the priority is on the number of search nodes, then it is more interesting to use shaving on apply search nodes and on refute search nodes. If the priority is on the solving time, then we should either try to estimate the type of search tree we will obtain (*i.e.* if there is a similar number of apply and refute search nodes, or if there is a dominance) or naively trigger the shaving in either apply or refute mode.

5.2.4 Quick shaving

In order to obtain better practical results, we can also use the quick shaving method [Lhomme, 2005]. A large part of the overhead brought by shaving is due to the fact that we check many edges and we shave only a small proportion of them. The idea of the quick shaving is to reuse the information provided by the search tree. Given n a node of the search tree. The quick shaving exploits the fact that if a value checked in n is not shaved, then it will not be shaved by the parent of n. Thus, the quick shaving is triggered only when a backtrack occurs in the search tree. It considers only the shaved edges of its children by maintaining incrementally the set of candidates. Initially, the set is empty. When a fail occurs in the search tree, the decision is added to candidates and recursively this set will be modified.

In Table 5.6, we give the results of the quick shaving. Unfortunately, it does not work much better than checking all edges for the TSP. Compared with Table 5.4, quick shaving is only 1.3 times faster on mean and produces 3.6 times more search nodes on mean and 1.9 times more search nodes on geometric mean. It can be explained by several reasons. Fist, the quick shaving is based on the backtrack algorithm. If $x = a$ is a failing choice, the marked elements as "not shaved" will be backtracked and therefore they will not be tried in $x \neq a$. However, we observed that some not shaved elements of $x = a$ could have been shaved in $x \neq a$. Then, the reduction of the number

	candidates: 1-mand				candidates: 2-mand				candidates: LCNode			
Instance	time	search nodes	#check	#shaved	time	search nodes	#check	#shaved	time	search nodes	#check	#shaved
a280	30.8	405	6,648	3,329	11.5	1,659	1,640	1,091	**9.3**	1,411	1,025	653
bier127	0.6	3	65	44	**0.5**	7	43	32	**0.5**	41	19	14
brg180	369.6	87	43,814	201	367.9	87	43,814	201	**1.3**	87	8	6
ch130	1.8	9	342	187	1.7	71	346	157	**1.5**	235	179	112
ch150	3.5	35	1,062	303	**2.4**	217	496	294	2.8	659	319	180
d198	12.5	161	3,222	1,421	**9.2**	973	1,489	964	12.2	2,561	1,485	804
eil101	0.6	15	146	22	0.4	19	62	27	**0.3**	59	20	12
gr120	1.2	5	230	142	**1.0**	51	136	98	1.2	165	93	59
gr137	4.8	51	1,402	613	2.5	267	440	319	**2.3**	373	313	195
gr202	2.0	17	320	203	2.7	209	319	177	**1.7**	199	85	58
gr229	122.5	1,179	37,874	16,384	49.4	4,971	13,492	6,099	**47.4**	13,691	8,420	5,306
gr96	2.6	67	1,034	447	1.3	143	255	173	**0.9**	113	88	46
kroA100	1.7	31	425	244	**1.3**	157	171	110	**1.3**	177	161	98
kroA150	**3.2**	23	769	339	4.5	415	1,106	671	3.9	821	731	472
kroA200	51.9	433	12,727	6,121	78.8	10,829	23,906	13,962	**44.4**	9,081	8,454	5,579
kroB100	4.4	69	1,735	720	3.3	677	933	709	**2.5**	477	458	290
kroB150	36.7	573	14,226	5,822	13.7	2,107	4,318	2,700	**13.0**	3,749	3,070	1,906
kroB200	23.1	227	6,989	3,160	20.7	2,727	5,788	4,501	**13.8**	3,859	2,336	1,412
kroC100	1.6	35	492	272	1.1	139	133	111	**1.0**	149	129	83
kroD100	0.7	5	138	100	**0.4**	11	52	39	**0.4**	79	65	43
kroE100	3.5	57	1,410	638	**2.2**	217	691	464	2.3	495	426	231
lin318	14.8	19	2,316	988	18.6	147	2,511	798	**5.6**	563	401	284
pr124	7.1	129	1,937	804	2.4	471	323	269	**2.2**	249	276	199
pr136	19.0	475	8,753	4,240	18.8	2,731	7,630	5,107	**12.6**	4,497	2,786	1,551
pr144	1.6	13	188	112	1.6	153	199	178	**1.5**	157	129	85
pr264	**5.3**	13	210	128	5.4	279	122	85	5.7	185	218	151
rat195	116.2	2,249	44,778	25,025	24.4	4,403	8,192	5,906	**12.3**	3,611	2,211	1,387
rat99	0.4	2	45	29	0.6	25	60	27	**0.3**	59	23	13
si175	**56.1**	259	25,285	5,002	369.6	29,845	191,537	45,648	164.7	72,579	47,256	26,771
st70	**0.2**	9	40	6	0.4	13	62	28	**0.2**	51	26	13
tsp225	148.7	1,725	46,593	20,210	77.1	11,439	20,144	11,320	**33.1**	6,243	4,901	2,981
u159	**0.8**	9	67	38	0.9	21	46	24	**0.8**	69	40	26
mean	32.8	262.2	8,290.1	3,040.4	34.3	2,358.8	10,326.8	3,196.5	**12.6**	3,960.8	2,692.2	1,594.4
geo mean	5.9	53.8	1,328.5	515.9	4.8	308.2	812.8	418.9	**3.2**	518.3	334.4	208.3

Table 5.2: Comparison of the shaving method with check: once and the candidates mode 1-mand, 2-mand and LCNode.

	calling: Apply				calling: Apply and Refute				calling: Refute			
		search				search				search		
Instance	time	nodes	#check	#shaved	time	nodes	#check	#shaved	time	nodes	#check	#shaved
a280	34.1	1,045	8,040	4,711	30.8	405	6,648	3,329	**14.3**	387	2,929	1,617
bier127	0.6	9	61	41	0.6	3	65	44	**0.5**	7	68	47
brg180	369.4	87	43,814	201	369.6	87	43,814	201	**23.0**	87	3,786	1
ch130	**1.6**	19	279	165	1.8	9	342	187	1.9	21	285	168
ch150	**3.2**	77	901	326	3.5	35	1,062	303	3.6	111	989	388
d198	**12.2**	297	2,721	1,506	12.5	161	3,222	1,421	12.6	375	2,938	1,478
eil101	0.6	15	146	22	0.6	15	146	22	**0.4**	17	71	19
gr120	1.3	15	232	147	**1.2**	5	230	142	1.3	17	228	141
gr137	5.8	305	1,706	716	4.8	51	1,402	613	**4.4**	141	986	439
gr202	2.1	29	270	173	**2.0**	17	320	203	2.5	39	318	191
gr229	91.9	2,147	28,255	15,328	122.5	1,179	37,874	16,384	**62.7**	1,319	18,688	9,484
gr96	2.5	173	1,030	481	2.6	67	1,034	447	**2.1**	99	581	330
kroA100	**1.6**	69	414	283	1.7	31	425	244	2.3	77	564	343
kroA150	4.3	87	1,242	609	**3.2**	23	769	339	3.6	53	827	419
kroA200	**38.8**	827	9,106	5,498	51.9	433	12,727	6,121	56.4	1,017	14,884	8,176
kroB100	**4.1**	175	1,518	789	4.4	69	1,735	720	5.3	179	1,755	943
kroB150	28.7	1,083	10,591	5,835	36.7	573	14,226	5,822	**20.2**	709	6,805	3,265
kroB200	24.6	591	7,704	4,025	23.1	227	6,989	3,160	**22.6**	537	7,044	3,790
kroC100	**1.4**	63	260	171	1.6	35	492	272	1.4	67	410	237
kroD100	**0.7**	11	122	87	0.7	5	138	100	0.9	19	152	115
kroE100	**3.3**	111	1,253	655	3.5	57	1,410	638	3.4	125	1,137	551
lin318	**12.6**	41	1,760	834	14.8	19	2,316	988	13.8	79	2,101	985
pr124	**5.7**	237	1,469	875	7.1	129	1,937	804	6.8	303	1,961	1,248
pr136	**15.2**	1,093	6,424	3,641	19.0	475	8,753	4,240	17.2	1,079	7,705	3,754
pr144	1.6	27	186	113	1.6	13	188	112	**1.5**	15	149	113
pr264	5.4	23	165	100	**5.3**	13	210	128	6.1	29	296	192
rat195	72.8	2,941	27,516	17,138	116.2	2,249	44,778	25,025	**71.6**	2,849	23,090	11,912
rat99	**0.4**	1	45	29	0.4	1	45	29	0.5	1	45	29
si175	106.0	1,889	59,646	14,254	**56.1**	259	25,285	5,002	122.3	1,999	62,536	15,895
st70	0.3	9	40	6	0.2	9	40	6	**0.1**	9	15	5
tsp225	123.3	3,607	38,903	20,535	148.7	1,725	46,593	20,210	**119.8**	2,901	33,152	16,035
u159	**0.7**	9	67	38	0.8	9	67	38	0.8	11	60	40
mean	30.5	534.8	7,996.4	3,104.1	32.8	262.1	8,290.1	3,040.4	**18.9**	458.7	6,142.3	2,573.4
geo mean	5.8	108.1	1,221.5	530.9	5.9	52.7	1,328.5	515.9	**5.1**	104.0	1,043.2	430.0

Table 5.3: Comparison of the shaving method with check: once, candidates: 1-mand and the calling mode Apply, ApplyAndRefute and Refute.

	calling mode: Apply				calling mode: Apply and Refute				calling mode: Refute			
Instance	time	search nodes	#check	#shaved	time	search nodes	#check	#shaved	time	search nodes	#check	#shaved
a280	11.6	53	2,537	783	12.0	37	2,522	773	**8.5**	43	1,748	851
bier127	3.8	3	760	350	**3.4**	3	760	350	3.7	9	764	353
brg180	369.8	87	43,814	201	369.6	87	43,814	201	**23.1**	87	3,786	1
ch130	2.3	11	646	280	**2.2**	7	654	284	2.3	17	594	272
ch150	2.4	3	461	242	**2.1**	3	461	242	2.2	3	461	242
d198	33.8	231	15,009	3,515	30.3	57	12,826	2,237	**28.7**	143	10,668	2,825
eil101	0.7	15	190	34	0.9	15	190	34	**0.4**	15	83	34
gr120	1.4	1	243	118	1.4	1	243	118	**1.2**	1	243	118
gr137	9.4	57	4,564	830	10.9	29	6,129	777	**7.7**	47	3,260	835
gr202	**3.3**	7	784	347	3.7	5	821	371	3.6	11	757	345
gr229	**327.3**	635	106,280	21,342	356.6	269	126,568	17,109	442.5	765	131,807	23,481
gr96	4.1	59	2,515	328	3.9	19	2,399	324	**2.5**	47	1,315	344
kroA100	5.9	65	3,452	702	6.7	25	4,571	665	**4.6**	27	2,205	518
kroA150	4.8	17	1,589	432	**4.7**	5	1,765	479	4.9	11	1,696	466
kroA200	**127.0**	187	37,927	5,459	158.3	81	48,046	5,038	133.8	255	40,168	5,040
kroB100	**4.7**	71	2,466	535	4.8	15	2,695	527	5.2	49	2,854	611
kroB150	**105.5**	499	52,632	8,816	136.8	201	70,075	8,662	109.3	593	55,566	10,648
kroB200	**45.8**	115	16,984	3,655	58.0	51	22,487	3,360	56.0	153	20,828	4,015
kroC100	1.7	3	415	222	**1.6**	3	415	222	1.7	9	407	216
kroD100	0.7	1	78	45	**0.5**	1	78	45	0.5	1	78	45
kroE100	3.1	21	1,524	364	3.2	9	1,721	404	**2.6**	17	1,076	305
lin318	18.8	11	3,413	972	19.1	7	3,416	975	**16.3**	27	2,639	1,068
pr124	**3.2**	11	898	432	3.4	3	931	456	3.8	5	937	461
pr136	49.3	675	36,695	5,919	72.7	343	58,952	6,349	**36.3**	531	26,177	4,689
pr144	2.1	1	399	240	2.2	1	399	240	**1.9**	1	399	240
pr264	**7.0**	1	385	211	7.2	1	385	211	7.3	1	385	211
rat195	174.3	619	73,328	11,258	213.3	213	86,666	9,329	**140.6**	607	58,324	10,477
rat99	0.4	1	60	33	0.4	1	60	33	**0.3**	1	60	33
si175	**60.9**	339	34,749	7,257	120.8	303	79,904	9,223	200.2	2,403	143,588	24,746
st70	**0.2**	9	53	8	0.3	9	53	8	0.2	9	29	8
tsp225	**395.6**	815	131,048	21,153	598.8	339	194,266	22,860	466.6	1,097	156,403	24,433
u159	**1.3**	7	210	106	1.4	7	210	106	1.3	7	198	106
mean	55.7	144.7	18,003.4	3,005.9	69.1	67.2	24,202.6	2,875.4	**53.8**	218.5	20,922.0	3,688.7
geo mean	8.6	25.6	2,399.2	596.8	9.4	15.1	2,686.7	588.1	**7.6**	28.2	2,051.5	523.6

Table 5.4: Comparison of the shaving method with check: once, candidates: full and the calling mode Apply, ApplyAndRefute and Refute.

	calling mode: Apply				calling mode: Apply and Refute				calling mode: Refute			
		search				search				search		
Instance	time	nodes	#check	#shaved	time	nodes	#check	#shaved	time	nodes	#check	#shaved
a280	**5.2**	1,107	186	138	9.3	1,411	1,025	653	5.7	821	466	278
bier127	**0.5**	53	7	5	0.5	41	19	14	0.5	39	4	4
brg180	1.6	101	1	1	**1.3**	87	8	6	1.6	87	8	6
ch130	**1.4**	343	71	53	1.5	235	179	112	1.4	217	123	71
ch150	**1.8**	433	106	73	2.8	659	319	180	2.0	435	195	110
d198	**8.8**	2,431	363	200	12.2	2,561	1,485	804	10.4	2,171	1,160	694
eil101	**0.3**	77	5	3	0.3	59	20	12	0.4	61	14	9
gr120	1.2	259	47	26	1.2	165	93	59	**1.1**	145	64	42
gr137	**1.7**	353	116	82	2.3	373	313	195	2.2	445	248	139
gr202	**1.7**	297	45	30	1.7	199	85	58	2.0	339	94	59
gr229	**45.7**	22,763	3,753	2,719	47.4	13,691	8,420	5,306	47.4	15,313	8,229	5,160
gr96	1.2	267	49	33	**0.9**	113	88	46	0.9	149	94	48
kroA100	**1.2**	389	104	78	1.3	177	161	98	1.4	211	166	90
kroA150	3.2	1,293	204	146	3.9	821	731	472	**2.8**	555	412	244
kroA200	**37.1**	13,427	3,705	2,858	44.4	9,081	8,454	5,579	44.8	9,993	7,528	4,710
kroB100	**1.8**	633	178	128	2.5	477	458	290	2.3	549	391	257
kroB150	13.6	7,271	1,600	1,190	**13.0**	3,749	3,070	1,906	15.3	4,793	3,117	1,835
kroB200	**12.5**	5,411	943	625	13.8	3,859	2,336	1,412	14.1	4,443	2,352	1,426
kroC100	1.1	207	54	37	**1.0**	149	129	83	1.1	145	105	68
kroD100	0.6	111	32	26	**0.4**	79	65	43	0.6	91	40	26
kroE100	**1.6**	559	156	111	2.3	495	426	231	2.5	591	424	214
lin318	**5.4**	1,197	76	50	5.6	563	401	284	6.7	937	490	350
pr124	**2.0**	507	193	153	2.2	249	276	199	3.0	531	426	285
pr136	12.7	7,521	1,206	713	12.6	4,497	2,786	1,551	**11.0**	3,977	2,223	1,274
pr144	1.6	249	41	29	1.5	157	129	85	**1.3**	155	87	60
pr264	**5.1**	229	122	98	5.7	185	218	151	5.7	153	179	80
rat195	12.7	5,823	1,146	814	**12.3**	3,611	2,211	1,387	15.1	4,727	2,427	1,461
rat99	**0.3**	51	9	4	0.3	59	23	13	0.3	53	6	3
si175	211.2	158,801	24,062	16,332	**164.7**	72,579	47,256	26,771	180.3	86,883	47,803	26,414
st70	0.3	63	12	5	**0.2**	51	26	13	0.4	41	12	7
tsp225	**32.4**	12,001	2,744	1,953	33.1	6,243	4,901	2,981	39.5	9,657	5,800	3,447
u159	0.8	125	20	12	0.8	69	40	26	**0.6**	89	33	24
mean	13.4	7,636.0	1,292.4	897.7	**12.6**	3,960.8	2,692.2	1,594.4	13.3	4,649.9	2,647.5	1,528.0
geo mean	**3.0**	741.0	131.6	89.6	3.2	518.3	334.4	208.3	3.2	555.3	260.6	159.4

Table 5.5: Comparison of the shaving method with check: once, candidates: LCNode and the calling mode Apply, ApplyAndRefute and Refute.

of search nodes obtained on the TSP is not due to particular edges, but by a set of particular edges having a strong structural impact. In addition, the Lagrangian relaxation makes the growth of the lower bound of the TSP non-monotone. Thus, if an edge is not shaved for a search node n, then it can be shaved in the parent of n and therefore it violates a fundamental rule of the quick shaving.

5.2.5 Model

In this section, we will study shaving with different models in order to better understand the impact of filtering algorithms on the shaving method. We then define 4 models: (1) WCC, (2) WCC + k-cutset constraint, (3) WCC + k-cutset constraint + mandatory Hamiltonian path constraint and (4) WCC + k-cutset constraint + mandatory Hamiltonian path constraint + one-tree constraint. Thus, there is more and more filtering going from models (1) to (4). For the sake of readability, we split the results in two tables (Table 5.7 and Table 5.8).

First, we see that adding filtering in the shaving is very efficient. For instance, the average solving time of (1) is 2.994s while the solving time of (2) is 200.8s, that of (3) is 83s and finally that of (4) is 69.1s. Compared to (1), we have a gain of a factor of 15 for (2), 36 for (3) and 43 for (4). Regarding the search nodes, we observe a gain factor of 23 for (2), 55 for (3) and 109 for (4). Nevertheless, one instance (brg180) works much better with less filtering. Here, this is explained by the transition from the model (2) to the model (3). The mandatory Hamiltonian path algorithm uses the 3-opt which has a complexity in $O(n^3)$. In the case of this instance solved with the shaving method, this filtering algorithm does not filter anything and therefore add an overhead. Finally, strengthening the filtering generally implies a reduction of the search space. Since the shaving method is very time consuming, avoiding shaving rounds by reducing the search space allows us to obtain huge factors improvement.

5.2.6 Search strategy

In Table 5.9, we compare the search strategy minDeltaDeg with and without LCFirst [Fages et al., 2016]. Similar results are obtained (geometric mean of the solving times equal to 10 with and without LCFirst) whereas in the initial paper of Fages et al. [Fages et al., 2016], adding LCFirst clearly improves the results. Thus, with the shaving and minDeltaDeg, we are able to remove LCFirst. In practice, we had some issues with this search strategy because of its high dynamicity. For instance, the parallelization of the TSP solving with EPS which decomposes the search tree in breadth whereas LCFirst needs to learn in depth [Isoart and Régin, 2020b]. In Table 5.10, we observe similar results but LCFirst still improves the solving times by a factor of 1.5. Thus, the shaving reduces the impact of LCFirst. However, the choice of the search strategy still has a strong interest in the TSP solving even with the shaving method (there is a 4.4 factor between LCFirst minDeltaDeg and LCFirst maxCost).

Instance	time	search nodes	#check	#shaved
a280	9.3	51	2,162	839
bier127	3.5	9	764	353
brg180	22.2	87	3,786	1
ch130	2.1	17	594	272
ch150	2.4	3	461	242
d198	30.3	179	11,482	2,943
eil101	0.5	15	83	34
gr120	1.3	1	243	118
gr137	8.1	53	3,359	684
gr202	3.8	11	757	345
gr229	272.9	733	91,204	18,930
gr96	2.5	47	1,302	321
kroA100	3.9	27	1,953	452
kroA150	4.7	11	1,671	455
kroA200	97.0	173	28,725	3,593
kroB100	4.1	35	2,286	585
kroB150	110.5	651	55,755	10,040
kroB200	43.3	141	15,137	3,386
kroC100	1.7	9	407	216
kroD100	0.6	1	78	45
kroE100	2.3	13	1,065	298
lin318	17.1	21	2,755	1,058
pr124	3.5	5	937	461
pr136	39.9	737	28,933	4,860
pr144	2.1	1	399	240
pr264	7.2	1	385	211
rat195	190.5	859	81,958	10,583
rat99	0.4	1	60	33
si175	146.9	2,151	112,241	18,364
st70	0.2	9	29	8
tsp225	601.6	1,687	205,431	27,437
u159	1.2	7	198	106
mean	51.2	242.1	20,518.8	3,359.8
geo mean	7.4	28.4	2,015.2	501.2

Table 5.6: Results for the quick shaving method.

	(1)				(2)			
Instance	time	search nodes	#check	#shaved	time	search nodes	#check	#shaved
a280	208.6	393	81,684	4,622	**9.5**	39	2,696	819
bier127	**3.3**	7	1,115	237	3.6	3	768	363
brg180	**39.9**	87	43,823	210	47.1	87	43,823	210
ch130	21.6	119	19,040	1,508	2.3	7	738	252
ch150	8.9	29	5,342	633	2.3	3	458	211
d198	149.6	253	67,254	6,248	116.3	171	51,385	6,115
eil101	**0.5**	17	313	42	0.6	15	199	36
gr120	1.6	5	849	193	**1.1**	1	258	113
gr137	40.7	161	37,069	3,736	12.3	31	7,672	1,124
gr202	42.2	157	28,084	2,458	3.6	5	946	372
gr229	7,572.6	8,771	3,113,639	262,299	1,146.6	1,085	477,521	63,763
gr96	20.2	139	22,348	1,552	10.1	61	9,254	914
kroA100	18.5	91	18,422	2,034	13.5	67	13,444	1,697
kroA150	118.6	371	84,425	7,591	8.7	11	4,053	803
kroA200	59,401.7	125,449	32,686,090	2,501,415	307	155	98,130	9,668
kroB100	29.7	181	33,263	2,957	8	29	5,610	845
kroB150	8,297.0	21,771	5,475,450	455,628	1,048.7	1,889	629,892	72,606
kroB200	1,868.7	3,359	946,174	72,337	514.8	745	261,581	25,620
kroC100	10.5	57	8,977	801	2	5	768	273
kroD100	1.1	3	373	119	0.7	1	130	69
kroE100	32	217	36,388	3,234	9	47	7,264	606
lin318	480.6	311	142,167	13,170	153	45	34,745	4,742
pr124	31.1	73	21,738	1,746	6.2	7	2,069	624
pr136	426.1	3,879	463,644	38,748	144.1	1,059	149,278	12,897
pr144	**1.6**	3	376	194	1.9	1	357	196
pr264	7.6	3	710	383	7.1	1	370	203
rat195	746.1	1,233	335,858	31,170	308.6	367	130,379	12,435
rat99	0.4	5	206	62	**0.3**	1	81	41
si175	11,380.5	64,049	10,619,938	804,255	1,515.2	3,547	1,058,812	121,425
st70	**0.2**	9	62	10	**0.2**	9	53	8
tsp225	4,843.40	3,645	1,660,106	137,000	1,019.1	677	351,832	35,684
u159	2	13	842	147	1.6	7	252	128
mean	2,994.00	7,339.4	1,748,617.8	136,148.1	200.8	318.1	104,525.6	11,714.4
geo mean	55.9	188.0	31,355.3	3,305.3	16.3	31.8	6,076.2	1,093.6

Table 5.7: Comparison of the shaving method with check: once, candidates: full, calling: ApplyAndRefute and the models (1) and (2)

	(3)				(4)			
Instance	time	search nodes	#check	#shaved	time	search nodes	#check	#shaved
a280	11.1	35	2,479	770	12	37	2,522	773
bier127	3.6	3	765	362	3.4	3	760	350
brg180	366.6	87	43,814	201	369.6	87	43,814	201
ch130	**2.1**	7	626	275	2.2	7	654	284
ch150	2.2	3	415	206	**2.1**	3	461	242
d198	**23.7**	47	9,373	1,899	30.3	57	12,826	2,237
eil101	0.6	15	199	36	0.9	15	190	34
gr120	1.3	1	265	124	1.4	1	243	118
gr137	**10.5**	25	5,886	849	10.9	29	6,129	777
gr202	**3.5**	5	880	384	3.7	5	821	371
gr229	415.4	229	137,450	21,096	**356.6**	269	126,568	17,109
gr96	5.2	23	3,243	468	**3.9**	19	2,399	324
kroA100	7.7	35	5,997	809	**6.7**	25	4,571	665
kroA150	5.5	7	2,180	471	**4.7**	5	1,765	479
kroA200	**135.5**	65	41,010	4,493	158.3	81	48,046	5,038
kroB100	5.5	19	3,287	585	**4.8**	15	2,695	527
kroB150	**118.1**	179	63,187	8,094	136.8	201	70,075	8,662
kroB200	**54.7**	51	21,861	3,233	58	51	22,487	3,360
kroC100	**1.6**	3	436	232	**1.6**	3	415	222
kroD100	0.6	1	103	60	**0.5**	1	78	45
kroE100	**3.0**	7	1,502	322	3.2	9	1,721	404
lin318	20.5	9	4,228	991	**19.1**	7	3,416	975
pr124	6.2	7	1,969	528	**3.4**	3	931	456
pr136	72.8	415	63,454	6,405	**72.7**	343	58,952	6,349
pr144	1.9	1	366	210	2.2	1	399	240
pr264	**6.7**	1	366	201	7.2	1	385	211
rat195	282.5	327	122,762	12,322	**213.3**	213	86,666	9,329
rat99	**0.3**	1	78	40	0.4	1	60	33
si175	512.9	2,315	436,475	38,520	**120.8**	303	79,904	9,223
st70	0.3	9	53	8	0.3	9	53	8
tsp225	**572.5**	345	199,901	23,192	598.8	339	194,266	22,860
u159	**1.1**	7	226	116	1.4	7	210	106
mean	83.0	133.9	36,713.6	3,984.40	**69.1**	67.2	24,202.6	2,875.4
geo mean	10.4	18.1	3,279.5	675.1	**10.0**	15.8	2,871.8	617.1

Table 5.8: Comparison of the shaving method with check: once, candidates: full, calling: ApplyAndRefute and the models (3) and (4)

	minDeltaDeg				LCFirst minDeltaDeg			
Instance	time	search nodes	#check	#shaved	time	search nodes	#check	#shaved
a280	**11.4**	37	2,522	773	12.0	37	2,522	773
bier127	3.7	3	760	350	**3.4**	3	760	350
brg180	**368.0**	87	43,814	201	369.6	87	43,814	201
ch130	**2.1**	7	654	284	2.2	7	654	284
ch150	2.4	3	461	242	**2.1**	3	461	242
d198	**22.1**	45	8,374	1,605	30.3	57	12,826	2,237
eil101	**0.8**	15	190	34	0.9	15	190	34
gr120	**1.4**	1	243	118	**1.4**	1	243	118
gr137	**10.4**	25	5,729	765	10.9	29	6,129	777
gr202	**3.6**	5	821	371	3.7	5	821	371
gr229	**349.0**	263	124,430	16,725	356.6	269	126,568	17,109
gr96	4.0	19	2,399	324	**3.9**	19	2,399	324
kroA100	**6.2**	23	3,978	568	6.7	25	4,571	665
kroA150	4.9	5	1,765	479	**4.7**	5	1,765	479
kroA200	**143.0**	69	44,354	4,569	158.3	81	48,046	5,038
kroB100	4.9	15	2,563	485	**4.8**	15	2,695	527
kroB150	145.7	213	73,327	9,122	**136.8**	201	70,075	8,662
kroB200	71.7	85	31,225	3,257	**58.0**	51	22,487	3,360
kroC100	1.7	3	415	222	**1.6**	3	415	222
kroD100	**0.5**	1	78	45	**0.5**	1	78	45
kroE100	3.4	9	1,721	404	**3.2**	9	1,721	404
lin318	**18.8**	7	3,416	975	19.1	7	3,416	975
pr124	3.6	3	931	456	**3.4**	3	931	456
pr136	**63.2**	365	53,378	4,825	72.7	343	58,952	6,349
pr144	**2.1**	1	399	240	2.2	1	399	240
pr264	**7.2**	1	385	211	**7.2**	1	385	211
rat195	**126.0**	129	50,319	6,684	213.3	213	86,666	9,329
rat99	**0.4**	1	60	33	**0.4**	1	60	33
si175	277.2	1,075	213,034	19,689	**120.8**	303	79,904	9,223
st70	**0.3**	9	53	8	**0.3**	9	53	8
tsp225	620.4	327	186,980	20,762	**598.8**	339	194,266	22,860
u159	**1.3**	7	210	106	1.4	7	210	106
mean	71.3	89.3	26,843.4	2,966.6	**69.1**	67.2	24,202.6	2,875.4
geo mean	**10.0**	16.3	2,871.5	605.9	**10.0**	15.8	2,871.8	617.1

Table 5.9: Comparison of the shaving method with check: once, candidates: full, calling: ApplyAndRefute and the search strategy minDeltaDeg and LCFirst minDeltaDeg.

	maxCost				LCFirst maxCost			
Instance	time	search nodes	#check	#shaved	time	search nodes	#check	#shaved
a280	13.4	45	2,672	720	**13.0**	47	2,915	727
bier127	4.1	3	760	350	**4.0**	3	760	350
brg180	5,323.8	7363	1108199	113708	**390.2**	407	100,168	10,420
ch130	**2.2**	7	596	264	**2.2**	7	596	264
ch150	**2.2**	3	461	242	2.3	3	461	242
d198	44.4	99	21,513	2,756	**41.3**	85	19,435	2,562
eil101	**0.8**	13	184	34	**0.8**	13	184	34
gr120	**1.3**	1	243	118	**1.3**	1	243	118
gr137	14.1	51	8,773	970	**12.9**	41	7,642	1,043
gr202	3.7	5	812	366	**3.3**	5	812	366
gr229	**456.9**	387	169,173	21,883	909.6	471	288,549	40,534
gr96	**2.7**	7	1,155	275	2.8	7	1,155	275
kroA100	7.6	31	5,537	588	**4.8**	11	2,770	587
kroA150	6.0	9	2,609	479	**5.7**	9	2,501	513
kroA200	**1,181.8**	819	425,935	24,290	1,289.9	863	447,906	39,340
kroB100	7.2	37	5,148	564	**5.7**	19	3,364	480
kroB150	581.7	857	313,399	17,569	**551.8**	835	302,339	22,796
kroB200	526.7	613	238,893	13,719	**288.6**	323	126,222	12,443
kroC100	**1.7**	3	408	217	**1.7**	3	408	217
kroD100	0.6	1	78	45	**0.5**	1	78	45
kroE100	**3.5**	15	2,038	302	4.1	15	2,261	357
lin318	25.4	9	5,392	1,454	**25.2**	9	5,024	1,178
pr124	4.9	5	1,277	565	**4.7**	5	1,277	565
pr136	123.0	627	103,289	3,970	**63.2**	251	47,101	4,518
pr144	2.1	1	399	240	**2.0**	1	399	240
pr264	7.4	1	385	211	**7.3**	1	385	211
rat195	1,090.8	1,149	475,689	15,659	**921.8**	1,099	398,060	31,491
rat99	0.5	1	60	33	**0.4**	1	60	33
si175	195.1	621	135,660	11,260	**169.8**	435	110,215	10,497
st70	**0.3**	9	53	8	**0.3**	9	53	8
tsp225	**4,553.0**	3,739	1,754,720	113,747	4,935.8	4,435	1,961,234	126,487
u159	**1.3**	7	210	106	1.4	7	210	106
mean	443.4	516.8	149,553.8	10,834.8	**302.1**	294.4	119,837.1	9,657.7
geo mean	16.8	30.5	5,045.8	937.6	**14.4**	24.6	4,329.3	923.2

Table 5.10: Comparison of the shaving method with check: once, candidates: full, calling: ApplyAndRefute and the search strategy maxCost and LCFirst maxCost.

5.3 Conclusion

In this section, we studied the shaving method. It consists in trying some pairs (variable, value), to successively assign them. If a failure was triggered by a pair, then that pair cannot belong to any solution. This method allows a reduction of the search space by several orders of magnitude when trying each pair (variable,value) for each search node of the search tree. Unfortunately, it leads to a huge overhead for the solving times. Therefore, we tried a lot of variations of the shaving. For example, trying only the optional edges having at least one end with an adjacent edge being mandatory, performing the shaving only for a specific type of search nodes, changing the model while the shaving is performed, *etc*. All these results lead to the same conclusions: the more we check edges, the more the search space is reduced and the more we use strong filtering algorithms, the more the shaving will be efficient. Thus, a good trade-off is hard to find. If we relax the filtering algorithm, then we will perform much more search node with a lighter shaving. In addition, reducing the number of iterations of the Lagrangian relaxation led to weaker bound, leading to a lighter filtering, leading to an augmentation of the search nodes. It should be noted that this method is hard to implement: the shaving must be performed between the running of the next search node and the end of the current search node therefore some data should be backtracked whereas others do not. Finally, the shaving is a very interesting method obtaining a similar number of search nodes to Concorde. Therefore, it shows that stronger filtering algorithms could lead to equivalent performance to Concorde.

CHAPTER 6

Efficient implementation

In this chapter, we present various code optimizations and we show how we implemented the TSP model in CP.

6.1 Data structures and algorithms

All the introduced methods of this thesis have been implemented in a locally developed CP solver in Java 11. First, we had to implement a graph variable. We needed to have access to: the mandatory edges, the optional edges, the mandatory edges adjacent to a node, the optional edges adjacent to a node and a delta over the modified edges. In addition, we have a direct access to the number of neighbors both optional and mandatory for each node. For the various algorithms we implemented, it was the right granularity. One disadvantage of this implementation is that sometimes the codes must be duplicated if we want to apply the same things to both mandatory and optional edges. Nevertheless, it can be hidden with syntactic sugaring.

When an instance is loaded, we construct both the graph variable represented by an adjacency list and a matrix. Incrementally, we only maintain the adjacency list. The matrix allows us to retrieve the id of an edge given a pair of nodes. For instance, the Mandatory Hamiltonian path constraint is much easier to implement if we have the matrix.

Next, we made the decision to convert all the floating-point numbers (*i.e.* the edge costs modified by the Lagrangian relaxation) in long integer by multiplying them by a huge number. In practice, we take the initial cost and we multiply it by 100,000,000. Then, we are dealing only with long integers. It is much easier to deal with long than with float or double for most of the algorithms we needed. In addition, it allows us to easily handle the rounding in the multipliers optimization of the Lagrangian relaxation.

On the other hand, most of the memory we need is allocated at the initialization step. Only some dynamic arrays allocate memory while solving, but it is quite rare and we carefully control that. Our experiments are running without the garbage collector so dear to Java.

6.1.1 1-tree computation

Profiling our code, we observed that half of the solving time is spent in the computation of the 1-trees, the other half is for the structural constraints with a little part for the solver internal methods. Therefore, it is very important to carefully optimize the 1-tree computation. For the MST computation we can use Kruskal's or Prim's algorithm [Cormen et al., 2009]. The filtering algorithm (particularly, the marginal cost filtering) starts by removing a lot of edges in the input graph. Indeed, it is very common on the graphs of the TSPLib to have at least 80% of the edges removed at the root of the search tree (see Figure 6.1, 86% of the edges are removed). Note that all the graphs from the TSPLib are initially complete graphs.

In practice, Prim's algorithm is usually faster than Kruskal's algorithm if the graph is dense, which is not the case here. Therefore, we chose to use Kruskal's algorithm every time we need to compute a 1-tree. In addition, the marginal cost filtering algorithm needs to compute a specialized data structure that is obtained from a Kruskal's computation. Therefore, there is no interest to consider Prim's algorithm here. We recall that Kruskal's starts by sorting the edges and successively add an edge in the MST if that edge does not create a cycle in the MST. In order to detect if an edge creates a cycle, we can use a DFS but it leads to poor results. Otherwise, we can use a disjoint set data structure [Cormen et al., 2009] in order to reduce the time complexity of the algorithm. It works by creating sets of elements that are connected. Therefore, if an edge has its two endpoints in the same set, then adding that edge in the MST would create a cycle. In practice, we used a non-recursive implementation of each algorithm we implemented.

Since we use Kruskal's algorithm, we can divide the 1-tree computation in three parts:

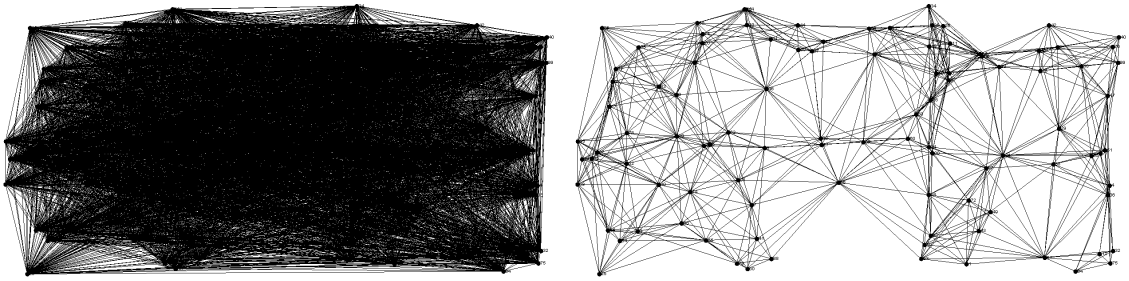


Figure 6.1: Graph kroA100 from TSPLib [Reinelt, 1991] with 100 nodes. The left graph is the input graph (a complete graph with 4950 edges). The right graph is obtained after propagation round at the root of the search tree (687 edges).

1. Updating the costs modified by the Lagrangian relaxation
2. Sorting the edges
3. Computing the 1-tree

There is not much to optimize in the point 3. Thus, we will discuss the points 1 and 2.

6.1.1.1 Updating the edge costs modified by the Lagrangian relaxation

In practice, it is easier (and faster) to deal with an array than with an adjacency list. Our experiments shown that it is worth spending times copying the edges in an array and sort the edges within the array rather than using sorting algorithms dealing with adjacency lists. Note that arrays allow a better use of the cache of the CPUs and we often call several times the sorting algorithms on the same array not necessarily modify.

Therefore, we copy the id of each edge of the graph variable in two stacks represented by arrays. One for the optional edges $optEdges$, the other for the mandatory edges $mandEdges$. We can see $optEdges$ as a "shrinking set" and $mandEdges$ as a "growing set".

Considering the problem of two successive updating of the costs in the same search node, there is no need to copy every time all the edges in $optEdges$ and $mandEdges$. Therefore, we can split the problem in three cases:

1. Costs have been modified
2. Domains have been modified
3. Costs and domains have been modified

Our experiments shown that saving and restoring $optEdges$ and $mandEdges$ does not improve the solving times. Therefore, at the beginning of each search nodes we copy all the edges in $optEdges$ and $mandEdges$. Then, in the search node we maintain the arrays. Since the costs are modified in the Lagrangian relaxation only, we can initialize $optEdges$ and $mandEdges$ in the beginning of the first propagation of the LR for each search node. Then, we must maintain them and update the costs incrementally.

The difference between two propagations of the LR is that some elements could be removed or detected mandatory by the other constraints. Therefore, if it is not the first propagation of the search node of the LR, then we parse the elements of $optEdges$ and we remove the edges that are no more in the optional edges of the graph. If an edge is removed but is newly a mandatory edge, then we add it to $mandEdges$. In order to remove an element of $optEdges$ we can either shift the elements (we can parse and maintain an offset) or swap the element to remove with the last element of $optEdges$. The difference between the two approaches is that swapping lost the order. Here, there is no need to update the costs since only the LR modifies the costs.

If we are in the optimization step of the LR, then the costs are modified. Then, we simply parse the edges of $optEdges$ and $mandEdges$ and to update the costs with the cost update function of the LR. If a pruning algorithm was triggered between two optimization steps, then we must update the elements of $optEdges$ and $mandEdges$ and update the costs.

These optimizations allow us to reduce by 20% to 30% the overall solving times.

6.1.1.2 Sorting the edges

In order to sort the edges for Kruskal's algorithm, we tried a lot of sorting algorithms. Experimentally, we defined that two of them are useful in the TSP purpose: the insertion sort in $O(n^2)$ and the radix sort in $O(wn)$ [Cormen et al., 2009] where n is the number of elements and w is the length of the largest element.

The insertion sort is used when there are very few elements or very few modifications of the costs. In practice, the insertion sort outperforms a lot of sorting algorithms when there are very few elements or when the list is almost sorted even if it has a bad worst-case complexity. Otherwise, we used the radix sort. In practice, we observed that the radix sort is faster than most of the candidates we considered (quick sort, merge sort, ...). However, the radix sort must be implemented carefully. First, we can have negative costs and a naive implementation of the radix sort did not handle that correctly. Indeed, in the binary representation the last bit is the sign bit. Therefore, if the algorithm sort elements with both negative and positive elements in non-decreasing order without taking care of the sign bit, then the negative elements will be sorted between them, but they will be after the positive elements (a negative number have its bit sign equal to 1). There are two ways to get over this, we can parse the array after we sort the elements and put the elements at the beginning of the array, or we can identify the negative elements before we sort them and perform the sorting step over the negative elements then over the positive elements. We chose to use the second method since we must update the costs, then we identify the negative elements and update the costs at the same time.

6.2 Conclusion

In this chapter, we presented various optimizations that we implemented in order to obtain an efficient code. First, we have shown our needs in terms of modeling. Then, we identified that most of the solving time is used to compute 1-trees. Thus, we focused our attention on this algorithm. We introduced an efficient way to parse the edges and update the costs modified by the Lagrangian relaxation. Finally, we have shown the radix sort and the insertion sort perfectly suit for the 1-tree computations.

CHAPTER 7

Conclusion and Perspectives

7.1 Conclusion

In this thesis, we considered the TSP in CP. Starting from the state of the art, that is the WCC in conjunction with the search strategy LCFirst, we defined and implemented three new constraints and filtering algorithms:

- The k-cutset constraint imposes that any solution contains a strictly positive and even number of elements in each cutset. We defined a linear time filtering algorithm based on Tarjan's DFS for the k-cutset constraint. Given a k-cutset with $k-1$ mandatory edges, if k is odd, then it removes the non-mandatory edge. Otherwise, it makes mandatory the non-mandatory edge. It also makes mandatory each edge belonging to a 2-cutset of the graph. Moreover, if there is a k-cutset with k mandatory edges such that k is odd, then it raises a failure and the current branch of the search tree can be aborted. Integrating this constraint in the WCC allows obtaining a reduction of the solving times of more than a factor of 2.

- The mandatory Hamiltonian path constraint is based on the local search k-opt algorithm. If a path of mandatory edges is not itself optimal (*i.e.* there exists a k-opt), then it cannot belong to any optimal solution and the current branch of the search tree can be aborted. Then, we defined a filtering algorithm removing edges: if a path can be improved when an edge is appended to it, then it cannot exist an optimal solution simultaneously containing that path and that edge. For this constraint, we defined an incremental version of the filtering algorithm greatly improving the overall solving times. The integration of this constraint into the WCC and the k-cutset constraint allows obtaining a reduction of at least a factor of 3.

- The one-tree constraint is based on the idea that if the problem can be decomposed in two independent sub-problems, then a part of the 1-tree can be optimal in a sub-problem. This constraint leads to a slight improvement of the results.

In order to improve the practical performances, we introduced SSSA which is an adaptive algorithm for the CP-based Lagrangian relaxation. It stops local multipliers optimization when the objective value no longer varies or oscillates and trigger the cost-based FAs at the end of the optimization phase. We experimentally measured that this part of the optimization step did not bring anything in terms of filtering. Therefore, using SSSA, the solving times are improved by a factor 9.1 in mean and 3 in geometric mean.

We then use EPS in order to parallelize the search for solutions in parallel. First, we have shown that the application of EPS to the TSP is not trivial. Indeed, EPS decomposition is breadth-based whereas TSP embeds LCFirst, a depth-based search strategy. Then, we introduced *Bound-Backtrack-and-Dive*, a diving algorithm, which consists in a first step of performing a sequential execution with a bounded number of backtrack in order to study the behavior of LCFirst. Then, run EPS, simulate LCFirst during the decomposition using our preliminary study and finally solve with a classical LCFirst the generated sub-problems in parallel. We have experimentally shown that the use of *Bound-Backtrack-and-Dive* improves the results compared to a sequential solving: the mean solving time is reduced by a factor of 4.9 and the number of search nodes is reduced by a factor of 1.9. Note that we used 8 workers. Then, we were interested in increasing the number of workers.

In addition, we identified three major issues in EPS arising while solving the TSP:

- Non-monotonic decomposition.
- Unstable decomposition.
- Extremely heterogeneous sub-problems.

The non-monotonic decomposition issue is solved by using a stopping criterion of the decomposition other than reaching the number of required sub-problems. Then, the two other issues are solved using EPSrd. That is EPS with a re-decomposition policy of hard sub-problems. It detects if there is a load balancing issue for a given problem. Then, it re-decomposes the sub-problems causing that issue in order to better distribute the work to do. Thus, EPSrd use smaller decomposition and re-decomposition steps than EPS and each time there is a huge load balancing issue it re-decomposes the sub-problems. Therefore, it avoids the cases of both an insufficient proportion of workers working because of some extremely heterogeneous problems and an unstable decomposition. Experimentally, we showed that an overall improvement of the solving times for EPSrd is obtained. In addition, with multiple runs of EPSrd we have shown that it is much more robust than EPS.

We also studied the shaving method for the TSP. It consists in selecting some pair (variable, value) and to successively assign them. If running a propagation step on an assignment leads to a failure, then the value can be safely removed from the domain of the variable. Trying all the pairs, we obtained a reduction over several orders of magnitude of the search space. However, this method is quite slow and trying all the pairs leads to an increase of the solving times. Then, we defined numerous variations of the shaving assigning particular pairs. Unfortunately, a good trade-off is hard to find. The experiments shown two things. First, the more the number of checked pairs is high, the more the search space is reduced. In addition, the more we use strong filtering algorithms, the more the shaving will be efficient. Finally, a method reducing by several orders of magnitude the search space is quite interesting especially when dealing with a problem that is so difficult to solve such as the TSP. Thus, it makes us believe that stronger filtering algorithms could lead to equivalent performance to Concorde.

In short, in this thesis we attacked the TSP with CP by introducing new constraints, engineering the state of the art constraint, introducing a method improving the parallelization of the TSP and introducing a method reducing by several order of magnitude the number of search nodes.

In Table 7.1, we give an experiment showing the solving times before and after this thesis. The column “Choco WCC” represents the solving times obtained with Choco and the WCC

Instances	Choco WCC time(s)	EPSrd time(s)
a280	806.6	1.6
d198	76.1	13.3
gil262	30,000.0	30.6
gr229	14,025.4	16.2
kroA200	30,000.0	7.3
kroB150	1,609.0	2.6
kroB200	2,218.9	6.6
pr264	21.6	16.5
rat195	330.5	5.7
si175	4,544.1	5.0
tsp225	4,171.4	9.6
mean	7,982.1	10.5

Table 7.1: Comparison of Choco WCC and our best configuration with parallelism.

[Fages et al., 2016], and the column "EPSrd" reprents the solving times we obtained with our best model and EPSrd on 192 workers. For this table, we selected the common solved instances of Table 3.17 and Table 4.8. We observe that a huge improvement factor is obtained. Indeed, the mean solving time for Choco WCC is 7,982.1s whereas we obtain a mean solving time of 10.5s. That is an improvement factor of 760. In order to obtain the best possible results, the model, the implementation and making the best possible use of our machine are the three most important things to deal with and this table showed that.

7.2 Perspectives

7.2.1 Continuation

In future works, the shaving method should be studied. Indeed, gain factors of two orders of magnitude on the search space show that more filtering could lead to a comparable search space size of Concorde. Therefore, it could be very interesting to search for a way to transfer the reduction of the search space to the solving times.

A programming work should be done in order to consider larger graphs. Indeed, most of the algorithms used throughout this thesis are systematic and consider all the edges of the graph. Thus, using incremental algorithms, like the one we used for the mandatory Hamiltonian path constraint, could greatly improve the solving time, especially for large graphs. For example, in the Lagrangian relaxation we compute a succession of minimum spanning trees with Kruskal's algorithm. Then, one can ask the following question: is it possible to define an incremental algorithm for Kruskal's algorithm when the costs are modified? Or, is it possible to use an incremental algorithm in some cases? It would save a lot of computation if we could do this incrementally.

Then, structural constraints such as the k-cutset constraint has shown a great interest. Therefore, continuing in this way seems interesting. For instance, we could extend this idea to graph nodes in order to deduce some filtering on edges.

Next, it could be interesting to split the problem of solving the TSP into some subproblems consisting in solving a minimum Hamiltonian path problem. In some cases, it is possible. For

instance, if there is a 2-cutset in the graph, then the graph can be cut in two parts with exactly one entry point and one exit point for each part. Then, each of these part can be solved independently by solving the minimum Hamiltonian path problem. However, this requires to obtain an upper bound on each part. This can be done with a modified version of LKH. In addition, we should be able to split the problem into two independent sub-problems at any time in the search tree. Clearly, this part is tricky and requires to carefully implement such a mechanism in a solver.

The work done on the mandatory Hamiltonian path constraint should be generalized. Indeed, it is an original way to integrate local search method within CP solvers. Therefore, trying to tackle other problems with this idea could be an original and efficient way to solve optimization problems.

All the experiments of this thesis are performed on the instances of the TSPLib [Reinelt, 1991]. However, almost all the instances are constructed to be hard with pathological cases. Note that in Section A we displayed all the 2D instances that we considered along this thesis. The Lagrangian relaxation could be used in order to avoid it. For instance, if we detect that there is a 3-cutset of optional edges in the graph, then we can add a constraint in the Lagrangian relaxation stating that the optimal solution with the current assignment contains exactly 2 edges from this cutset.

7.2.2 Extension

The work we have done could be extended to variations of the TSP. For example, a first extension case could be the extension to asymmetric graphs. In this case, the k-cutset constraint should be modified to handle asymmetric graphs. For example, if we have a 2-cutset for the cut (S, T), then there must exist an edge going from S to T and the other going from T to S. For the mandatory Hamiltonian path constraint, since we check only valid permutations (*i.e.* we swap the edges only with existing ones in the initial graph), there is no need to modify the algorithm.

Next, this work could be extended to VRP [Dantzig and Ramser, 1959], in which case the k-cutset constraint and the mandatory Hamiltonian path constraint could be used without any modification.

Finally, we could consider the TSPTW, that is a TSP with the constraint specifying that some cities must be visited within a given time window. For this problem, the k-cutset constraint does not need any modification, since it enforces the existence of a Hamiltonian path. However, the mandatory Hamiltonian path should be carefully handled. Indeed, if a k-opt is found on a mandatory Hamiltonian path, then the k-opt must be consistent with the time windows constraint.

Bibliography

[Applegate et al., 2006] Applegate, D. L., Bixby, R. E., Chvatal, V., and Cook, W. J. (2006). *The traveling salesman problem: a computational study*. Princeton university press.

[Barbagallo et al., 1996] Barbagallo, S., Bodoni, M. L., Medina, D., Corno, F., Prinetto, P., and Reorda, M. S. (1996). Scan insertion criteria for low design impact. In *Proceedings of 14th VLSI Test Symposium*, pages 26–31. IEEE.

[Beasley, 1993] Beasley, J. E. (1993). *Lagrangian Relaxation*, chapter 6, pages 243–303. John Wiley & Sons, Inc., New York,NY, USA.

[Bellman, 1958] Bellman, R. (1958). Combinatorial processes and dynamic programming. Technical report, RAND CORP SANTA MONICA CA.

[Bellman et al., 1957] Bellman, R., Bellman, R., and Corporation, R. (1957). *Dynamic Programming*. Rand Corporation research study. Princeton University Press.

[Benchimol et al., 2012] Benchimol, P., Van Hoeve, W.-J., Régin, J.-C., Rousseau, L.-M., and Rueher, M. (2012). Improved filtering for weighted circuit constraints. *Constraints*, 17(3):205–233.

[Bentley, 1992] Bentley, J. J. (1992). Fast algorithms for geometric traveling salesman problems. *ORSA Journal on computing*, 4(4):387–411.

[Bergman et al., 2015] Bergman, D., Cire, A. A., and van Hoeve, W.-J. (2015). Improved constraint propagation via lagrangian decomposition. In *International Conference on Principles and Practice of Constraint Programming*, pages 30–38. Springer.

[Borůvka, 1926] Borůvka, O. (1926). O jistém problému minimálním.

[Boudreault and Quimper, 2021] Boudreault, R. and Quimper, C.-G. (2021). Improved CP-Based Lagrangian Relaxation Approach with an Application to the TSP. In Zhou, Z.-H., editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 1374–1380. International Joint Conferences on Artificial Intelligence Organization. Main Track.

[Burton and Sleep, 1981] Burton, F. W. and Sleep, M. R. (1981). Executing Functional Programs on a Virtual Tree of Processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194, New York, NY, USA. ACM.

[Cambazard and Fages, 2015] Cambazard, H. and Fages, J.-G. (2015). New filtering for AtMostNValue and its weighted variant: A Lagrangian approach. *Constraints*, 20(3):362–380.

[Carlier and Pinson, 1994] Carlier, J. and Pinson, E. (1994). Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78(2):146–161. Project Management and Scheduling.

[Carlson, 1997] Carlson, S. (1997). Algorithm of the gods. *Scientific American*, 276(3):121–123.

[Chazelle, 2000] Chazelle, B. (2000). A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity. *J. ACM*, 47(6):1028–1047.

[Christofides, 1976] Christofides, N. (1976). Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem.

[Chvátal, 1973] Chvátal, V. (1973). Edmonds polytopes and weakly hamiltonian graphs. *Math. Program.*, 5(1):29–40.

[Cook et al., 2011] Cook, W., Cunningham, W., Pulleyblank, W., and Schrijver, A. (2011). *Combinatorial Optimization*. Wiley Series in Discrete Mathematics and Optimization. Wiley.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

[Dantzig et al., 1954] Dantzig, G., Fulkerson, R., and Johnson, S. (1954). Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410.

[Dantzig and Ramser, 1959] Dantzig, G. B. and Ramser, J. H. (1959). The truck dispatching problem. *Management science*, 6(1):80–91.

[Debruyne and Bessiere, 1997] Debruyne, R. and Bessiere, C. (1997). Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *IJCAI*.

[Demassey, 2017] Demassey, S. (2017). *Compositions and hybridizations for applied combinatorial optimization*.

[Ducomman et al., 2016] Ducomman, S., Cambazard, H., and Penz, B. (2016). Alternative Filtering for the Weighted Circuit Constraint: Comparing Lower Bounds for the TSP and Solving TSPTW. In *AAAI*.

[Edmonds, 1965] Edmonds, J. (1965). Paths, Trees, and Flowers. *Canadian Journal of Mathematics*, 17:449–467.

[Edmonds and Karp, 1972] Edmonds, J. and Karp, R. M. (1972). Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*, 19(2):248–264.

[ein alter Commis-Voyageur, 1832] ein alter Commis-Voyageur (1832). *"Der Handlungsreisende – wie er sein soll und was er zu tun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein – von einem alten Commis-Voyageur"*.

[Fages et al., 2016] Fages, J.-G., Lorca, X., and Rousseau, L.-M. (2016). The salesman and the tree: the importance of search in CP. *Constraints*, 21(2):145–162.

[Flood, 1956] Flood, M. M. (1956). The Traveling-Salesman Problem. *Operations Research*, 4(1):61–75.

[Fontaine et al., 2014] Fontaine, D., Michel, L. D., and Van Hentenryck, P. (2014). Constraint-based lagrangian relaxation. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 324–339.

[Ford, 1956] Ford, L. R. (1956). *Network Flow Theory.* RAND Corporation, Santa Monica, CA.

[Frangioni, 2002] Frangioni, A. (2002). Generalized Bundle Methods. *SIAM J. on Optimization*, 13(1):117–156.

[Galea and Le Cun, 2007] Galea, Fran c. and Le Cun, B. (2007). Bob++ : a Framework for Exact Combinatorial Optimization Methods on Parallel Machines. In *International Conference High Performance Computing & Simulation 2007 (HPCS'07) and in conjunction with The 21st European Conference on Modeling and Simulation (ECMS 2007)*, pages 779–785.

[Gomory, 1958] Gomory, R. (1958). Outline of an Algorithm for Integer Solutions to Linear Programs. *Bulletin of the American Mathematical Society*, 64:275–278.

[Gonzales, 1962] Gonzales, R. (1962). *Solution of the Traveling Salesman Problem by Dynamic Programming on the Hypercube*. Interim technical report. Massachusetts Institute of Technology.

[Grötschel et al., 1991] Grötschel, M., Jünger, M., and Reinelt, G. (1991). Optimal control of plotting and drilling machines: a case study. *Zeitschrift für Operations Research*, 35(1):61–84.

[Grötschel and Padberg, 1979] Grötschel, M. and Padberg, M. (1979). On the symmetric travelling salesman problem I: Inequalities. *Mathematical Programming*, 16:265–280.

[Gutin et al., 2001] Gutin, G., Punnen, A., Barvinok, A., Gimadi, E., and Serdyukov, A. (2001). The Traveling Salesman Problem and Its Variations.

[Gutwenger and Mutzel, 2000] Gutwenger, C. and Mutzel, P. (2000). A linear time implementation of SPQR-trees. volume 1984.

[Halstead, 1984] Halstead, R. (1984). Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 9–17, New York, NY, USA. ACM.

[Haralick and Elliott, 1979] Haralick, R. and Elliott, G. (1979). Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313.

[Held and Karp, 1962] Held, M. and Karp, R. M. (1962). A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied mathematics*, 10(1):196–210.

[Held and Karp, 1970] Held, M. and Karp, R. M. (1970). The Traveling-Salesman Problem and Minimum Spanning Trees. *Operations Research*, 18(6):1138–1162.

[Held and Karp, 1971] Held, M. and Karp, R. M. (1971). The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1(1):6–25.

[Helsgaun, 2000] Helsgaun, K. (2000). An effective implementation of the Lin–Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130.

[Hopcroft and Tarjan, 1973] Hopcroft, J. E. and Tarjan, R. E. (1973). Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158.

[Isoart and Régin, 2019] Isoart, N. and Régin, J.-C. (2019). Integration of Structural Constraints into TSP Models. In Schiex, T. and de Givry, S., editors, *Principles and Practice of Constraint Programming*, pages 284–299, Cham. Springer International Publishing.

[Isoart and Régin, 2020a] Isoart, N. and Régin, J.-C. (2020a). Adaptive CP-Based Lagrangian Relaxation for TSP Solving. In Hebrard, E. and Musliu, N., editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 300–316, Cham. Springer International Publishing.

[Isoart and Régin, 2020b] Isoart, N. and Régin, J.-C. (2020b). Parallelization of TSP Solving in CP. In Simonis, H., editor, *Principles and Practice of Constraint Programming*, pages 410–426, Cham. Springer International Publishing.

[Isoart and Régin, 2021a] Isoart, N. and Régin, J.-C. (2021a). A k-Opt Based Constraint for the TSP. In Michel, L. D., editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:16, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Isoart and Régin, 2021b] Isoart, N. and Régin, J.-C. (2021b). A Linear Time Algorithm for the k-Cutset Constraint. In Michel, L. D., editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:16, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Isoart and Régin, 2020] Isoart, N. and Régin, J.-C. (EasyChair, 2020). Relationship Between K-Cutsets and Comb Inequalities. EasyChair Preprint no. 2715.

[Johnson and McGeoch, 2008] Johnson, D. and McGeoch, L. A. (2008). The Traveling Salesman Problem: A Case Study in Local Optimization.

[Jünger et al., 1995] Jünger, M., Reinelt, G., and Rinaldi, G. (1995). Chapter 4 the traveling salesman problem. In *Network Models*, volume 7 of *Handbooks in Operations Research and Management Science*, pages 225–330. Elsevier.

[Karger et al., 1995] Karger, D. R., Klein, P. N., and Tarjan, R. E. (1995). A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328.

[Karp and L. Ruzzo, 1996] Karp, R. and L. Ruzzo, M. T. (1996). Algorithms in molecular biology (course notes). department of computer science and engineering, university of washington, seattle, waishington, usa.

[Khemmoudj et al., 2005] Khemmoudj, M. O. I., Bennaceur, H., and Nagih, A. (2005). Combining arc-consistency and dual Lagrangean relaxation for filtering CSPs. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 258–272. Springer.

[Korf, 1985] Korf, R. E. (1985). Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artif. Intell.*, 27(1):97–109.

[Kruskal, 1956] Kruskal, J. B. (1956). On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1):48–50.

[Kuhn, 1955] Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97.

[Le Cun et al., 2007] Le Cun, B., Menouer, T., and Vander-Swalmen, P. (2007). Bobpp. http://forge.prism.uvsq.fr/projects/bobpp.

[Lecoutre et al., 2009] Lecoutre, C., Saïs, L., Tabary, S., and Vidal, V. (2009). Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592 – 1614.

[Lenstra, 1974] Lenstra, J. K. (1974). Clustering a data array and the traveling-salesman problem. *Operations Research*, 22(2):413–414.

[Letchford and Lodi, 2002] Letchford, A. N. and Lodi, A. (2002). Polynomial-Time Separation of Simple Comb Inequalities. In Cook, W. J. and Schulz, A. S., editors, *Integer Programming and Combinatorial Optimization*, pages 93–108, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Lhomme, 2005] Lhomme, O. (2005). Quick Shaving. In *AAAI*.

[Lin, 1965] Lin, S. (1965). Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44:2245–2269.

[Lin and Kernighan, 1973] Lin, S. and Kernighan, B. (1973). An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Oper. Res.*, 21:498–516.

[Malapert et al., 2016] Malapert, A., Régin, J., and Rezgui, M. (2016). Embarrassingly Parallel Search in Constraint Programming. *J. Artif. Intell. Res. (JAIR)*, 57:421–464.

[Martin and Shmoys, 1996] Martin, P. and Shmoys, D. (1996). A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem.

[Menana, 2011] Menana, J. (2011). *Automates et programmation par contraintes pour la planification de personnel*. PhD thesis, Université de Nantes.

[Or, 1977] Or, I. (1977). Traveling salesman type combinatorial problems and their relation to the logistics of regional blood banking.

[Perron, 1999] Perron, L. (1999). Search Procedures and Parallelism in Constraint Programming. In *Principles and Practice of Constraint Programming – CP'99: 5th International Conference, CP'99, Alexandria, VA, USA, October 11-14, 1999. Proceedings*, pages 346–360, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Prim, 1957] Prim, R. C. (1957). Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401.

[Régin, 1994] Régin, J.-C. (1994). A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94, the 12th (US) National Conference on Artificial Intelligence*. Citeseer.

[Régin, 2008] Régin, J.-C. (2008). Simpler and Incremental Consistency Checking and Arc Consistency Filtering Algorithms for the Weighted Spanning Tree Constraint. In Perron, L. and Trick, M. A., editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 233–247, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Régin and Malapert., 2017] Régin, J.-C. and Malapert., A. (2017). Parallel Constraint Programming. In *Handbook of Parallel Constraint Reasoning,*. Springer, y. hamadi and l. sais edition.

[Régin et al., 2013] Régin, J.-C., Rezgui, M., and Malapert, A. (2013). Embarrassingly Parallel Search. In *19th International Conference CP 2013 Uppsala Sweden*.

[Régin et al., 2014] Régin, J.-C., Rezgui, M., and Malapert, A. (2014). Improvement of the Embarrassingly Parallel Search for Data Centers. In O'Sullivan, B., editor, *Principles and Practice of Constraint Programming*, volume 8656 of *Lecture Notes in Computer Science*, pages 622–635. Springer International Publishing.

[Régin et al., 2010] Régin, J.-C., Rousseau, L.-M., Rueher, M., and van Hoeve, W.-J. (2010). The Weighted Spanning Tree Constraint Revisited. In Lodi, A., Milano, M., and Toth, P., editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 287–291, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Reinelt, 1991] Reinelt, G. (1991). TSPLIB—A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4):376–384.

[Rosenkrantz et al., 1997] Rosenkrantz, D. E., Stearns, R. E., and Lewis II, P. M. (1997). *An analysis of several heuristics for the traveling salesman problem*, volume 6, pages 563–581.

[Rosenkrantz et al., 1974] Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M. (1974). Approximate algorithms for the traveling salesperson problem. In *15th Annual Symposium on Switching and Automata Theory (swat 1974)*, pages 33–42.

[Rossi et al., 2006] Rossi, F., van Beek, P., and Walsh, T., editors (2006). *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier.

[Sellmann, 2004] Sellmann, M. (2004). Theoretical Foundations of CP-Based Lagrangian Relaxation. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, pages 634–647.

[Sellmann and Fahle, 2003] Sellmann, M. and Fahle, T. (2003). Constraint Programming Based Lagrangian Relaxation for the Automatic Recording Problem. *Annals OR*, 118(1-4):17–33.

[Tarjan, 1974] Tarjan, R. E. (1974). A Note on Finding the Bridges of a Graph. *Inf. Process. Lett.*, 2:160–161.

[Tarjan, 1983] Tarjan, R. E. (1983). *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics.

[Tsin, 2009] Tsin, Y. H. (2009). Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms*, 7(1):130 – 146. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP).

[Vidal et al., 2010] Vidal, V., Bordeaux, L., and Hamadi, Y. (2010). Adaptive K-Parallel Best-First Search: A Simple but Efficient Algorithm for Multi-Core Domain-Independent Planning. In *Proceedings of the Third International Symposium on Combinatorial Search*.

[Yeh et al., 2010] Yeh, L.-P., Wang, B.-F., and Su, H.-H. (2010). Efficient algorithms for the problems of enumerating cuts by non-decreasing weights. *Algorithmica*, 56(3):297–312.

List of Figures

List of Tables

List of definitions

List of Algorithms

Appendix

A Representation of the instance set

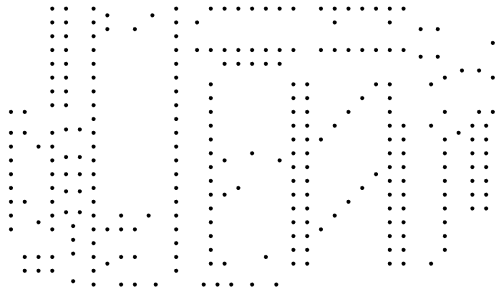


Figure A.1: a280

Figure A.2: bier127

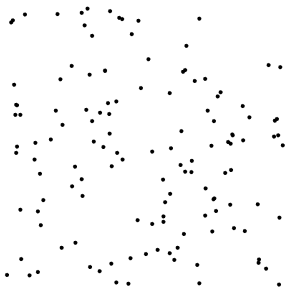


Figure A.3: ch130

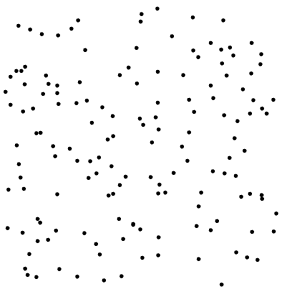


Figure A.4: ch150

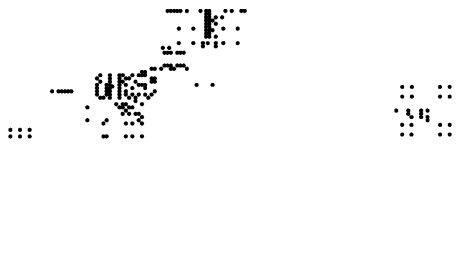


Figure A.5: d198

Figure A.6: d493

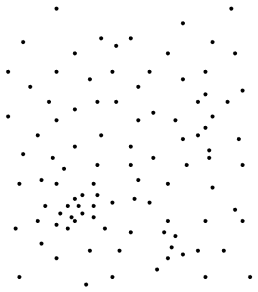


Figure A.7: eil101

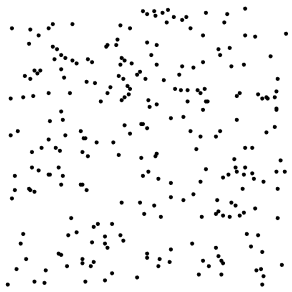


Figure A.8: gil262

Figure A.9: kroA100

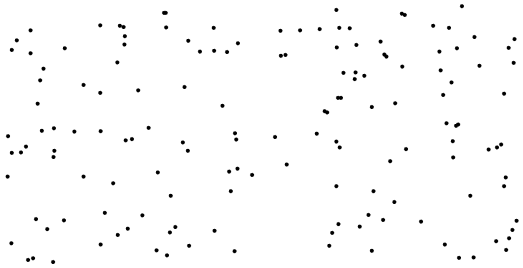


Figure A.10: kroA150

Figure A.11: kroA200

Figure A.12: kroB100

B Imposing edges in an MST

In this section, we are interested in the consequences of imposing edges in T a minimum spanning tree. We prove that the sum of the marginal costs in T of the imposed edges is a lower bound of the additional costs. More precisely, given $\overline{w}(T, e)$ the marginal cost of the edge e in T, we prove that if we impose a set I of non-tree edges of T, then $w(T_I) \geq \sum_{e \in I} \overline{w}(T, e) + cost(T)$ such that T_I a minimum spanning tree containing all the edges of I.

We recall the Optimality Conditions of an MST:

Property B.1.

Figure A.13: kroB150

Figure A.14: kroB200

Figure A.15: kroC100

Figure A.16: kroD100

Figure A.17: kroE100

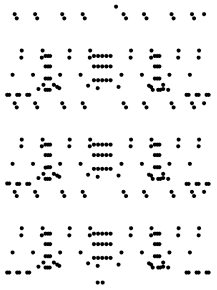


Figure A.18: lin318

- **[Path Optimality Condition]** A spanning tree T is a minimum spanning tree if and only if it satisfies the following path optimality conditions: for every non-tree edge $\{i, j\}$ of G, cost($\{i, j\}$) $\geq$ cost($\{u, v\}$) for every edge $\{u, v\}$ contained in the path in T connecting nodes i and j.
- **[Cut Optimality Condition]** A spanning tree T is a minimum spanning tree if and only if it satisfies the following cut optimality conditions: for every tree edge $\{i, j\}$ of G, cost($\{i, j\}$) $\leq$ cost($\{u, v\}$) for every edge $\{u, v\}$ contained in the cut formed by deleting edge $\{i, j\}$ from T.

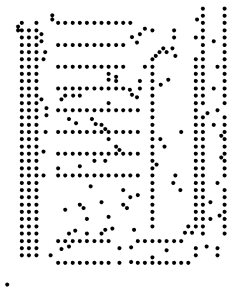


Figure A.19: pcb442

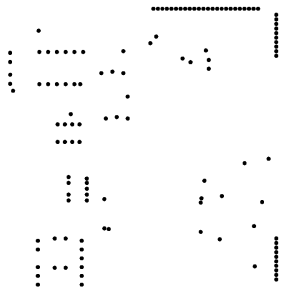


Figure A.20: pr124

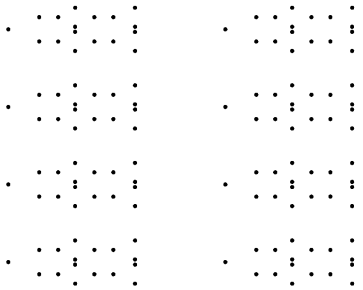


Figure A.21: pr136

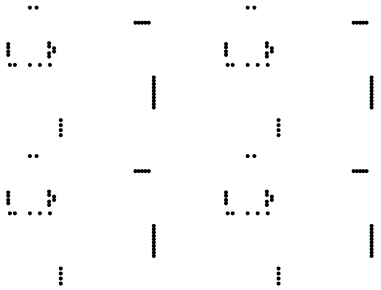


Figure A.22: pr144

Figure A.23: pr264

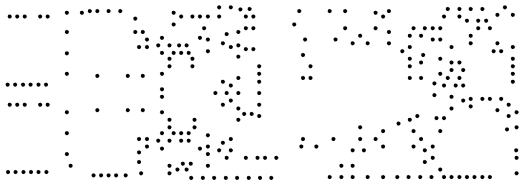


Figure A.24: pr299

For the sake of clarity, we will consider that T is a minimum spanning tree of G and $T_{\{i,j\}}$ is the minimum spanning tree containing the edge $\{i, j\}$.

Property B.2. Let $G = (X, E)$ be a graph, $\{i, j\} \in E$ be an edge of G. A minimum spanning $T_{\{i,j\}}$ in G can be computed by first merging the nodes i and j and then by computing a MST.

The support edge of a non-tree edge is defined as follows:

Property B.3. The support edge s_e of a non-tree edge $\{i, j\}$ is the non-mandatory edge with the maximum cost of the simple path going from i to j in T.

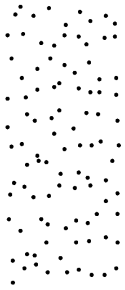


Figure A.25: rat99

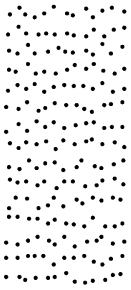


Figure A.26: rat195

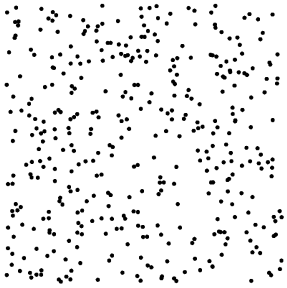


Figure A.27: rd400

Figure A.28: st70

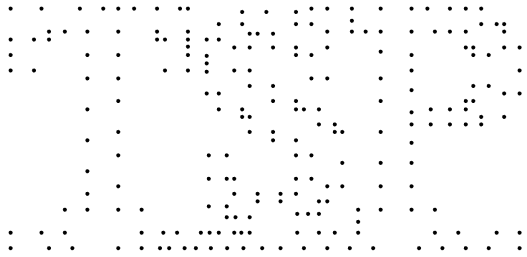


Figure A.29: tsp225

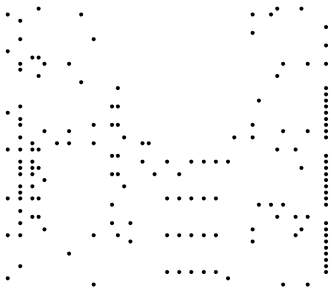


Figure A.30: u159

Proof.

If the edge $\{i, j\}$ is added to the tree, then a cycle is created and the Path Optimality Condition implies that the edge of the cycle having the largest cost must be removed. Therefore, $T_{\{i,j\}}$ is obtained by removing the support edge of $\{i, j\}$ and adding $\{i, j\}$ in T. □

Note that it is possible that an edge has no support edge by closing a path of mandatory edges. In this case, we will consider that the marginal cost of this edge is infinite. Note that a filtering algorithm will remove that edge.

Notation B.1.

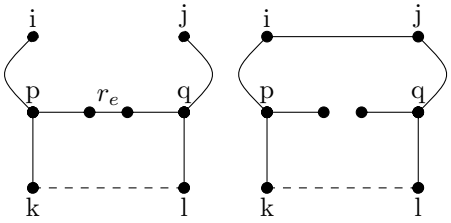


Figure B.31: Imposition of the non-tree edge $\{i, j\}$ and $\{k, l\}$ in T. T is the left graph and $T_{\{i,j\}}$ the right graph. The edge r_e is the support edge of $\{i, j\}$ and $\{k, l\}$ in T.

- $P(T, \{i, j\})$ the edges of the simple path from i to j in the minimum spanning tree T.
- support$(T, \{i, j\})$ is the support edge of the edge $\{i, j\}$ in the minimum spanning tree T.
- $\overline{w}(T, \{i, j\})$ is the marginal cost of the edge $\{i, j\}$. It is defined by $w(\{i, j\}) - w(support(T, \{i, j\}))$.

In Property B.4, we will show that imposing an edge $\{k, l\}$ in T always leads to an increase smaller or equal than that imposing $\{k, l\}$ in $T_\{i, j\}$ such that $\{i, j\}$ is another edge that we previously imposed.

Property B.4. $\forall \{k, l\} \notin T$: $\overline{w}(T_{\{i,j\}}, \{k, l\}) \geq \overline{w}(T, \{k, l\})$

Proof.

There are two cases: $support(T, \{i, j\})$ belongs to $P(T, \{k, l\})$ or not.

If $support(T, \{i, j\})$ does not belong to $P(T, \{k, l\})$, then $support(T_{\{i,j\}}, \{k, l\}) = support(T, \{k, l\})$. Thus, the marginal cost is not changed.

Now, consider that $support(T, \{i, j\}) \in P(T, \{k, l\})$. $T_{\{i,j\}}$ is computed by applying the replacement operation from T: the edge $support(T, \{i, j\})$ is removed and $\{i, j\}$ is added. Since $support(T, \{i, j\}) \in P(T, \{k, l\})$, then the path from k to l in $T_{\{i,j\}}$ is different from $P(T, k, l)$ because $support(T, \{i, j\}) \notin T_{\{i,j\}}$. Without loss of generality, we assume that k can reach i in T when $support(T, \{i, j\})$ is removed from T. The path $P(T_{\{i,j\}}, \{k, l\})$ can be split into three parts: $P(T_{\{i,j\}}, \{k, i\})$, $\{i, j\}$ and $P(T_{\{i,j\}}, \{j, l\})$. The edge $\{i, j\}$ cannot be a support edge because it is imposed in the spanning tree. Thus, the support edge of $\{k, l\}$ is either in $P(T_{\{i,j\}}, \{k, i\})$ or in $P(T_{\{i,j\}}, \{j, l\})$.

$P(T_{\{i,j\}}, \{k, i\})$ can also be split into two parts (that can be empty): $P(T_{\{i,j\}}, \{k, p\})$ and $P(T_{\{i,j\}}, \{p, i\})$ where p is the node in $P(T, \{i, j\})$ and in $P(T, \{k, l\})$ whose removal in $T_{\{i,j\}}$ disconnects k and i (See Figure B.31)*. Clearly, we have $\forall \{u, v\} \in P(T_{\{i,j\}}, \{p, i\}), cost(\{u, v\}) \leq cost(support(T, \{i, j\}))$, because these edges belong to $P(T, \{i, j\})$ and the support edges have the largest cost. Similarly we have $\forall \{u, v\} \in P(T_{\{i,j\}}, k, p), cost(\{u, v\}) \leq cost(support(T, \{k, l\}))$. In addition $cost(support(T, \{i, j\}) \leq cost(support(T, \{k, l\}))$ because $support(T, \{i, j\}) \in$

*In fact, there are three possibilities: either there is a path from i to j through k, or a path from k to j though i, or a fork having i and k as extremities with p in the center and path from p to j. We consider only the latter case which is more general.

$P(T, \{k, l\})$. So, every edge in $P(T_{\{i,j\}}, \{k, i\})$ has a cost that is less than or equal to $cost(support(T, \{k, l\}))$.

A similar reasoning can be applied to $P(T_{\{i,j\}}, \{j, l\})$. $P(T_{\{i,j\}}, \{j, l\})$ can also be split into two parts (that can be empty): $P(T_{\{i,j\}}, \{j, q\})$ and $P(T_{\{i,j\}}, \{q, l\})$ where q is the node in $P(T, \{i, j\})$ and in $P(T, \{k, l\})$ whose removal in $T_{\{i,j\}}$ disconnects j and l (See Figure B.31). We have $\forall \{u, v\} \in P(T_{\{i,j\}}, q, j), cost(\{u, v\}) \leq cost(support(T, \{i, j\}))$, because these edges belong to $P(T, \{i, j\})$ and the support edges have the largest cost. We also have $\forall \{u, v\} \in P(T_{\{i,j\}}, \{l, q\}), cost(\{u, v\}) \leq cost(support(T, \{k, l\}))$. In addition $cost(support(T, \{i, j\})) \leq cost(support(T, k, l))$ because $support(T, \{i, j\}) \in P(T, \{k, l\})$. Thus, every edge in $P(T_{\{i,j\}}, \{j, l\})$ has a cost that is less than or equal to $cost(support(T, \{k, l\}))$.

Hence, $cost(support(T_{\{i,j\}}, \{k, l\}) \leq cost(support(T, \{k, l\})$ so the marginal cost of $\{k, l\}$ in T is less than or equal to the marginal cost of $\{k, l\}$ in $T_{\{i,j\}}$. □

We can now define the wanted proposition:

Property B.5. Let T be an MST and $I = \{e_1, e_2, ...e_n\}$ a set of non-tree edges of T. Then, $\overline{w}(T_I, \{k, l\}) \geq \overline{w}(T, \{k, l\})$.

Proof.

We assume it is true for $n - 1$ edges. From Property B.4 we have $\overline{w}(T_{e_1,e_2,...e_n}, \{k, l\}) \geq \overline{w}(T_{e_1,e_2,...e_{n-1}}, \{k, l\})$. In addition we have $\overline{w}(T_{e_1,e_2,...e_{n-1}}, \{k, l\}) \geq \overline{w}(T, \{k, l\})$. So, the proposition holds. □

This means that we have the final proposition:

Property B.6. Let T be an MST and $I = \{e_1, e_2, ...e_n\}$ a set of non-tree edges of T. Then, $w(T_I) \geq \sum_{e \in I} \overline{w}(T, e) + cost(T)$.

Le problème du voyageur de commerce en programmation par contraintes

Nicolas ISOART

Résumé

Plusieurs modèles de programmation par contraintes, basés sur la méthode de relaxation lagrangienne (LR), ont été introduits pour résoudre le problème du voyageur de commerce (TSP). Dans cette thèse, nous définissons trois nouvelles contraintes et algorithmes de filtrage considérant la structure du graphe. La contrainte k-cutset impose que toute solution contienne un nombre strictement positif et pair d'éléments dans chaque cutset. La contrainte mandatory Hamiltonian path est basée sur l'algorithme de recherche locale k-opt. Si un chemin composé d'arêtes obligatoires n'est pas lui-même optimal (c.-à-d., il existe un k-opt), alors ce chemin n'appartient à aucune solution optimale. Enfin, la contrainte du 1-tree est basée sur l'idée que si le problème peut être décomposé en deux sous-problèmes indépendants, alors une partie du 1-tree peut être optimale dans un des sous-problèmes. De plus, nous introduisons l'algorithme SSSA afin d'améliorer les temps de résolution. SSSA évite les oscillations et les non-variations de la fonction objective de la LR. Ensuite, nous parallélisons la recherche de solutions avec Embarrassingly Parallel Search (EPS). Malheureusement, le mécanisme de décomposition d'EPS est un processus à profondeur borné, contrairement à la stratégie de recherche utilisée pour résoudre la TSP qui est en profondeur d'abord. Cela rend difficile l'obtention de bons résultats en appliquant directement EPS. Afin de diminuer ce défaut, nous introduisons un algorithme de précalcul. Cependant, des sous-problèmes avec des temps de résolution extrêmement différents peuvent apparaître. Pour remédier à cela, nous introduisons une méthode procédant à des redécompositions dans EPS. Finalement, nous expérimentons sur la TSPLib. Nous montrons que les contraintes structurelles permettent de réduire les temps de résolution d'un ordre de grandeur, et que la parallélisation permet d'obtenir de très bons résultats liés au nombre de coeurs.

Mots-clés : PPC, TSP, algorithmes, optimisation, parallélisme.

Abstract

Several constraint programming (CP) models, based on Lagrangian relaxation (LR), have been introduced to solve the traveling salesman problem (TSP). In this thesis, we define three new constraints and filtering algorithms based on the structure of the graph. First, the k-cutset constraint imposes that any solution contains a strictly positive and even number of elements in each cutset. Then, the mandatory Hamiltonian path constraint is based on the local search k-opt algorithm. If a path of mandatory edges is not optimal (i.e. it exists a k-opt), then it cannot belong to any optimal solution. Finally, the 1-tree constraint is based on the idea that if the problem can be decomposed in two independent sub-problems, then a part of the 1-tree can be optimal in a sub-problem. In addition, to speed-up the practical performances, we introduce an algorithm named SSSA to avoid oscillations and non-variations of the objective function of LR, saving useless solving times. We also parallelize the search for solutions with Embarrassingly Parallel Search (EPS). Unfortunately, a direct application of EPS does not lead to good results for the TSP. Indeed, the decomposition mechanism of EPS is a depth-bounded process whereas the search strategy used to solve the TSP is depth-first. Therefore, we define a diving algorithm fixing this issue. However, sub-problems with extremely different solving times may appear. Thus, we introduce a re-decomposition policy in EPS. Finally, our experiments on the TSPLib showed that the structural constraints reduce the solving times by an order of magnitude. Moreover, we show that our version of EPS leads to a huge improvement related to the number of cores.

Keywords: CP, TSP, algorithms, optimization, parallelism.