POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Accelerating TSP Solving by Using Cost-Based Solution Densities of Relaxations

PIERRE COSTE
Département de mathématiques et de génie industriel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Mathématiques appliquées

Août 2019

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Accelerating TSP Solving by Using Cost-Based Solution Densities of Relaxations

présenté par **PIERRE COSTE**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Michel GENDREAU, président
Andrea LODI, membre et directeur de recherche
Gilles PESANT, membre et codirecteur de recherche
Louis-Martin ROUSSEAU, membre

ACKNOWLEDGEMENTS

I would like to show my appreciation to the individuals who helped me during the completion of this Master's thesis.

I would like to thank Dr. Andrea Lodi, my research director, and Dr. Gilles Pesant, my research co-director, for supervising my project throughout the duration of my Master's studies and for helping me overcome any trouble in my research or writing. I greatly appreciate their patience in taking the time to proof-read this thesis and their contribution to my work as a whole.

My appreciation also goes to Dr. Michel Gendreau and Dr. Louis-Martin Rousseau for forming the validation jury for this thesis.

Finally, I am extremely grateful to my parents, my brother, my grand-parents, the rest of my family and my close friends, for all their support and encouragement over the years.

My thanks go out to all the people mentioned here, without whom this would not have been possible.

RÉSUMÉ

Le problème du voyageur de commerce, ou problème du commis voyageur, est l'un des problèmes les plus importants dans le domaine de l'optimisation combinatoire. Il a fait l'objet d'inombrables travaux de recherche, à la fois théoriques et pratiques.

Parmi les aspects de ce problème, nous nous intéressons particulièrement, dans le cadre de notre sujet, à certaines de ses relaxations, qui ont aussi été étudiées pour apporter de nouvelles approches à la résolution du problème. Les structures combinatoires de ces relaxations peuvent être encapsulées dans des contraintes globales existantes en programmation par contraintes (PPC), ce qui nous motive à tester une approche basée sur des travaux récents sur les heuristiques de dénombrement en PPC.

L'objectif de ce projet est d'améliorer la résolution du problème du voyageur de commerce en appliquant les densités de solution aux relaxations du problème. On pose l'hypothèse qu'une arête a très peu de chance d'appartenir à la solution optimale du problème si plusieurs relaxations retournent de faibles densités de solution pour cette arête et qu'on peut donc l'éliminer pour nettoyer le graphe d'entrée du problème. On évalue donc chaque arête en fonction de leur densité de solution pour chaque relaxation et on élimine les arêtes évaluées comme "mauvaises" par toutes les relaxations.

Pour l'expérimentation, cet algorithme de pré-traitement sera appliqué à plusieurs exemplaires de TSPLIB, une bibliotheque d'exemplaires du problème de voyageur de commerce. On évaluera d'abord le temps de calcul de notre méthode. Enfin, on résoudra nos exemplaires élagués avec différents solveurs (CONCORDE, Gurobi et IBM CP Optimizer) et on comparera les résultats obtenus à la résolution des exemplaires originels. L'élagage est efficace si le temps de résolution gagné en pré-traitant les exemplaires compense le temps de pré-traitement.

ABSTRACT

The Traveling Salesman Problem is a combinatorial optimization problem, which, broadly speaking, consists of visiting a certain number n of cities, by passing through each city exactly once and by traveling the shortest possible distance. This problem is very prominent in research, as a representative of the NP-hard class of problems and as a problem with applications in various areas, including routing, networking and scheduling. Nowadays, integer programming methods dominate the landscape of TSP solvers, with the state-of-art solver CONCORDE.

As part of the efforts to solve the TSP, several of its relaxations have been studied, for computing lower bounds or domain filtering. Since these relaxations can provide insight on the combinatorial structure of the problem, we believe recent work in Constraint Programming concerning counting-based branching heuristics can bring new effective methods of using these relaxations.

In this Master's thesis, we present an approach to the traveling salesman problem which exploits cost-based solution densities from counting-based search. we propose a method for eliminating edges from the input graph of TSP instances in pre-processing, by using the solution densities from relaxations of the TSP to determine promising edges. Solution densities from different relaxations can also be combined for branching in a constraint programming solver. The efficiency and robustness of our pre-processing algorithm is evaluated by applying it to instances from TSPLIB and comparing the time to solve them with that of the original complete instances. We consider various solvers in our experimentation, namely the IBM CP Optimizer, CONCORDE and Gurobi.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

TSP	Traveling Salesman Problem
CP	Constraint Programming
CBS	Counting-Based Search
ILP	Integer Linear Programming
MIP	Mixed Integer Programming
SD(s)	Solution Density(ies)
STSP	Symmetric Traveling Salesman Problem
ATSP	Asymmetric Traveling Salesman Problem
DFJ	Dantzig Fulkerson Johnson
MTZ	Miller Tucker Zemlin

CHAPTER 1 INTRODUCTION

The project presented in this thesis belongs to the fields of Constraint Programming and Integer Programming, two distinct branches of mathematical optimization. The aim of this project is to use techniques from constraint programming to simplify the Traveling Salesman Problem, so it can be solved quickly in constraint programming or integer programming solvers. In this first chapter, we will introduce concepts necessary to understand the field of study and the context of our research project. We will then present our own objectives, hypotheses and give an outline of the rest of this thesis.

1.1 Constraint Programming

Constraint Programming (CP) is an optimization paradigm designed to solve constraint satisfaction problems, as well as constraint optimization problems. It differs from other methods of solving optimization problems, in that constraints in the model are declared in literal terms rather than being formatted in a specific way, such as constraints in Linear Programming which must take the form $Ax \leq b$. Global constraints, a common example of which is the `AllDifferent` constraint, are constraints found in CP that involve multiple variables and that are simply defined by name in CP models.

Since there is no strict formulation for constraints in CP models, methods for solving these models efficiently are not generalized, like the simplex algorithm for LP. Instead, algorithms specific to each global constraint must be implemented to help with solution search and are already available in CP solvers. These algorithms can involve domain filtering, i.e., removing unsupported values from the set of potential assignments of a variable, or guiding the search tree using heuristics.

The solving process in CP operates as a depth first search tree. But before beginning the search, the first step of the process is domain filtering. First, it is important to note that, in CP, variables have a finite domain of values. Constraint-specific algorithms are implemented to remove unsupported values, or values that are not part of any feasible solution of the problem, from the domains of each variable, to reduce the search space that the branching algorithm must traverse. Formally speaking, we say that General Arc Consistency is reached when, for every constraint c and every variable x that c applies to, all values in the domain of x are consistent with c, i.e., found in an assignment satisfying c.

Once filtering is done and consistency is reached, we can begin the search. At a node of the search tree, a branching heuristic is used to determine which variable x, and sometimes value v, to branch on. The tree splits into two sub-trees. How the tree splits varies from heuristic to heuristic: there could be a sub-tree where $x < v$ and one where $x \geq v$ or there could be a sub-tree where $x = v$ and one where $x \neq v$.

Before proceeding to the next node, constraint propagation takes place. This consists of reducing the feasible domains of other variables in the problem depending on which variable we branched on and which constraints these variables share. Essentially, by reducing a variable's domain during branching, some solutions may become infeasible. Thus, we need to check if variables sharing a global constraint with the branching variable have had values in their domain become unsupported. If so, we remove these values and we then check all variables sharing a constraint with this newly modified variable, hence the term "propagation". These propagation algorithms have unique implementations according to the global constraint they are applied to.

Once propagation is done, branching continues, until either a solution is found or until the sub-tree is infeasible. The sub-tree becomes infeasible when fixing a variable during branching causes another variable's domain to become empty during propagation. In this case, the algorithm backtracks and explores the other sub-tree.

When optimizing in CP, branching does not stop after finding a first solution and continues until it can prove that the best solution found is optimal. In order to avoid going through the entire search tree, partial assignments at each node are used to find lower bounds on the solutions of the corresponding sub-tree and if a sub-tree's lower bound is greater than the upper bound, in case of minimization, then the sub-tree is not explored.

There are various popular branching heuristics that perform well in practice : MIN-DOMAIN-SIZE selects the variable with the smallest number of values in its domain, as the name implies. As is the case with all heuristics that do not specify the value, the value to assign is chosen lexicographically, i.e., in numerical or alphabetical order. DOM/DDEG selects the variable with the smallest ratio of its domain size to its degree, the number of constraints it is found in. DOM/WDEG follows the same idea as DOM/DDEG but adds a weight to each constraint, starting at one, which is incremented each time this constraint causes infeasibility during branching. Counting-based search (CBS) heuristics [2] select the variable-value assignment with the largest solution density (SD), a value indicating how often the assignment appears in solutions satisfying a constraint.

1.2 Integer Programming

An integer programming problem is a mathematical optimization problem in which all variables must take integer values. Generally speaking, integer programming refers to Integer Linear Programming, where the objective function (value we aim to maximize or minimize) and the constraints are linear expressions:

$$\begin{aligned} \min_x \quad & c^T x && c \in \mathbb{R}^n \\ s.t. \quad & Ax \geq b && A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m \\ & x \geq 0 \\ & x \in \mathbb{Z}^n \end{aligned}$$

Different nomenclature exists for certain variants of ILP problems. Mixed integer programming (MIP) refers to problems where only some of the variables are under integrality constraints and others can be non-discrete. 0-1 programming problems are problems where all variables are binary.

There are many generic algorithms, either exact or heuristic, which can solve any given ILP problem.

One of the most relevant exact methods is using cutting planes, introduced by Ralph Gomory [3]. Cutting plane methods consist of solving the ILP's linear programming (LP) relaxation, which is obtained by relaxing the integrality constraints of the problem, with the simplex algorithm and adding linear constraints, known as cuts, to remove non-integer solutions without reducing the feasible integer solution space. Commonly used cuts include Chvátal-Gomory cuts [3], MIR inequalities [4], cover cuts, flow covers [5].

In practice, cutting plane methods prove to be impractical and inefficient on their own. However, a method called branch-and-cut [6] was introduced, combining these cutting planes with a branch-and-bound algorithm [7]. In branch-and-bound, we first find the optimal solution of the LP relaxation. If there is a variable x_i such that its optimal value x_i' is not integer, then the problem is split into two sub-problems, one with the constraint $x_i \leq \lfloor x_i' \rfloor$ and the other with $x_i \geq \lceil x_i' \rceil$. The process is repeated recursively on these two new problems. Sub-trees can be eliminated if the lower bound of the node (solution value of the LP relaxation) is greater than the upper bound of the problem (best found integer solution), when minimizing the objective function, since all solutions found in the sub-tree will be greater than its lower bound and thus worse than the best solution found so far. In branch-and-cut, cutting planes

can be applied at each node to tighten the LP relaxation.

Some ILPs can prove very difficult to solve in which case heuristic algorithms, providing approximate solutions to the problem, can be used instead of exact algorithms. These methods include local search algorithms, such as hill climbing [8] and tabu search [9, 10], simulated annealing [11] and ant colony optimization [12] algorithms.

1.3 The Traveling Salesman Problem

Given a set of cities and the pair-wise distance between each city, the Traveling Salesman Problem (TSP) consists of finding the shortest cycle visiting each city exactly once. Formally speaking, this problem consists of finding an Hamiltonian cycle in a undirected graph $G = (V, E)$.

First called the "Messenger problem" by Karl Menger in 1930, then the "48 States problem" by Hassler Whitney in 1932, the name "Traveling Salesman Problem" was popularized by Merrill Flood in 1948. This problem is one of the most heavily studied problems in optimization, for various reasons.

One of these reasons is that the problem has practical applications in a wide range of fields. As the problem statement suggests, it has many applications in the fields of transportation, routing, as well as logistics or scheduling, for such applications as the routing of delivery trucks, the scheduling of warehouse cranes or the planning of telecommunication networks. Additionally, since its model is relatively simple, it is open to interpretation for use in other areas by modifying the definition of the city and the definition of a distance. For instance, we have seen this problem applied in such fields as genetics, for DNA sequencing, in astronomy and in X-ray crystallography.

Furthermore, what makes this problem interesting is that, despite its intuitive formulation, it is very difficult to solve it in practice. The TSP, like many other ILP problems, is part of the NP-hard class of optimization problems and no polynomial-time algorithm capable of solving it has been found as of yet. This gives the problem a lot of theoretical interest, as solving this problem in polynomial time, or proving it cannot be done, would answer the millennium question "P=NP?". The difficulty of the problem also makes it a good option for testing general methods.

1.4 Research Objectives

Our research aims to find a new way to exploit solution densities from constraints in CP and apply this method to the Traveling Salesman Problem. We believe that, by drawing out different CP constraints from various relaxations of the TSP and applying CBS (Counting-Based Search) to each of these relaxations individually, we can combine the information given by these constraints to identify the edges that are most likely to be found in an optimal tour of the TSP. This hypothesis comes from the fact that each constraint in Constraint Programming often represents a combinatorial substructure of the problem and different relaxations will reveal different substructures. Thus, we propose a method for removing undesirable edges, according to the counting-based search's computations, from the input graph of TSP instances as pre-processing and test its effectiveness by comparing the time necessary for solving the initial instance and our sparse instance. We will also use the SDs of different constraints for branching in CP and test the performance of various heuristics, combined with our pre-processing method.

1.5 Outline

In this document, we will first go over existing literature relevant to our project, in chapter 2, including breakthroughs in solving the TSP as well as work on counting-based search. We will then present our method for pre-processing graphs and the models we applied it to, in chapter 3, which will finally lead to the experimentation done to test our method's effectiveness, in chapter 4.

CHAPTER 2 LITERATURE REVIEW

In order to better understand our problem, we present this literature review taking a look at previous works in the context of our project. It will first cover relevant works on the Traveling Salesman Problem and its relaxations. It will then go over counting-based search heuristics in constraint programming. In particular, we will be interested in applications of constraint programming on the 2-matching relaxation, the 1-tree relaxation and the n-path relaxation of the TSP.

2.1 Traveling salesman problem

This section reviews relevant literature pertaining to the Traveling Salesman Problem. We will present common formulations of the TSP, as well as state-of-the-art methods of solving the problem, and, finally, various works on the problem's relaxations, which are of particular interest for our research.

2.1.1 Formulations

Karg and Thompson [13] state the TSP as : Given an $n \times n$ matrix of real numbers $C = (c_{ij})$, the traveling salesman problem consists of finding an acyclic permutation $(i_1, i_2, ..., i_n)$ of integers $1, 2, ..., n$ such that $c_{i_1 i_2} + c_{i_2 i_3} + ... + c_{i_{n-1} i_n}$ is minimized.

The TSP can be stated conceptually as finding the shortest Hamiltonian cycle in a graph. Considering a directed graph, a Hamiltonian cycle is a cycle, a set of arcs where each arc is adjacent to two other arcs, that passes exactly once through each node of the graph. The concept also applies to undirected graphs, where the arcs are replaced by unordered pairs of nodes, called edges, or links in the context of flow networks. Fulkerson [14] describes the traveling salesman problem using the concept of Hamiltonian cycles.

Integer programming formulations of the traveling salesman problem have been presented by many authors such as Dantzig et al. [15], Bellmore and Nemhauser [16], Miller and al. [17], and Golden [18].

Given a matrix $c = (c_{ij})$ of pairwise distances between nodes i and j for $i, j = 1, ..., n$:

$$\begin{aligned} \min & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \\ \text{s.t.} & \sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, ..., n \\ & \sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, ..., n \\ & \text{sub-tour elimination constraints} \\ & x_{ij} \in \{0, 1\}, \quad i, j = 1, ..., n \end{aligned}$$

where $x_{ij} = 1$ if arc (i, j) is in the tour, and 0 otherwise.

Or, given an undirected graph $G = (V, E)$ and costs $C = (c_{ij})$ associated with the edges of the graph :

$$\min \sum_{(i,j) \in E} c_{ij} x_{ij} \tag{2.1}$$

$$\text{s.t.} \sum_{(i,j) \in \delta(i)} x_{ij} = 2, \qquad i \in V \tag{2.2}$$

$$\text{sub-tour} \quad \text{elimination constraints} \tag{2.3}$$

$$x_{ij} \in \{0, 1\}, \qquad (i, j) \in E \tag{2.4}$$

where $\delta(i)$ denotes edges incident with node i.

Sub-tour elimination constraints are often formulated as:

$$\sum_{(i,j) \in A(S)} x_{ij} \leq |S| - 1, \quad S \subseteq V, 2 \leq |S| \leq |V| - 2$$

where $A(S)$ denotes the edges with both endpoints in set S. This also works with the undirected case, by specifying $i > j$, as to not count both x_{ij} and x_{ji}, and replacing $A(S)$ with $E(S)$. This particular integer programming formulation was introduced by Dantzig, Fulkerson and Johnson [15]. Despite the exponential number of sub-tour elimination constraints, this integer programming model of the TSP is often used since it offers the most effective linear programming relaxation.

For undirected graphs, there is another equivalent formulation for these constraints:

$$\sum_{i \in S, j \in \bar{S}} x_{ij} \geq 2, \quad S \subseteq V, 2 \leq |S| \leq |V| - 2$$

where $\bar{S} = V \backslash S$, meaning that subset S must be connected to the rest of the graph by at least two edges.

Miller, Tucker and Zemlin [17] propose an integer programming formulation which only requires a polynomial number of sub-tour elimination constraints. It should be noted that this formulation is for the asymmetric TSP and that there is no direct equivalent to this formulation for the symmetric TSP. However, any symmetric TSP (STSP, i.e., one in an undirected graph) is a special case of the asymmetric one (ATSP, in a directed graph) so the formulation can be used anyhow. The MTZ formulation introduced $n-1$ additional variables u_i $(i = 2, ..., n)$ and defines the sub-tour elimination constraints as follows :

$$u_i - u_j + (n-1)x_{ij} \leq n - 2 \quad i, j = 2, ..., n, i \neq j$$
$$1 \leq u_i \leq n - 1 \quad i = 2, ..., n$$

The u_i variables do no represent any concrete aspect of the problem, but are defined such that, for an assignment of x defining a valid tour, there is an assignment of u satisfying each individual constraint. The idea behind these sub-tour elimination constraints is that, if there was a sub-tour $(2, 3, ..., k+1)$, then taking the sum of these constraints for $(i, j) = ((2, 3), (3, 4), ..., (k, k+1), (k+1, 2))$ would yield $k(n-1) \leq k(n-2)$, which is contradictory.

Despite its compactness, this formulation has a weaker linear relaxation than the Dantzig, Fulkerson and Johnson formulation (DFJ) defined prior.

2.1.2 Approaches to solving the TSP

As this problem has been thoroughly studied, many algorithms have been developed in recent years to solve it. Among these methods are exact methods which guarantee that the solution found is optimal and heuristic methods which cannot guarantee optimality but find solutions reasonably close to the optimal.

Several algorithms have been implemented using tour-to-tour improvements, such as algorithms by Croes [19], Lin [20], Lin and Kernighan [21], Helsgaun [22] and Golden [23]. Tour-

to-tour improvement methods consist of finding an initial feasible tour and improving upon it until it can no longer be improved. The way these tours are improved varies from algorithm to algorithm. The Lin-Kernighan algorithm aims to improve by swapping two edges of the tour with two other unused edges, whereas LKH (Lin-Kernighan-Helsgaun) [22] can swap up to five. These methods are heuristic and do not always find the optimal TSP tour, since, like many neighborhood algorithms, they tend to fall into a local optimum, i.e., a solution better than all its neighboring solutions. Regardless, implementations of the Lin-Kernighan algorithm have been shown to find values 1% of optimality and LKH finds solution 0.1% short of optimality.

Though this may be the most prominent example, there are many other types of heuristic methods, such as tour merging [24, 25] and genetic algorithms [26].

One of the biggest breakthroughs in solving the TSP is Dantzig, Fulkerson and Johnson's cutting plane method [15]. This method is based on the observation that the assignment problem, which consists of finding a minimum-weight perfect matching in a weighted bipartite graph, is a relaxation of the TSP and that if the optimal solution of the assignment problem contains no sub-tours then it is an optimal solution of the TSP. The result is an example of a row generation algorithm. The idea is to relax the sub-tour elimination constraints from the TSP model and start by solving this relaxed problem. We then check if the solution violates any of the sub-tour elimination constraints and, if it does, these constraints are added to the problem and we solve again. This process is repeated until a solution that respects all sub-tour elimination constraints is found. This work not only influenced the TSP, but extends to solve many combinatorial problems, as well as many integer linear programming problem.

Today, the state-of-art solver for the TSP is CONCORDE, an Integer Programming solver developed by W. Cook et al. [27, 28], solving instances up to size 85,900. CONCORDE combines many of the methods mentioned previously. It computes an initial upper bound on the solution by using a Chained Lin-Kernighan [29] heuristic algorithm. This initial solution is used to bound the branch-and-cut search tree that follows.

There have been several approaches which rely on eliminating edges from the input graph of routing problems, in order to work on simpler, sparse graphs. Granular tabu search (GTS) [30] is a variant of the tabu search algorithm, proposed by Toth and Vigo for the Vehicle Routing Problem (VRP), though it can be adapted to the TSP. GTS works with

restricted neighborhoods, called granular neighborhoods, obtained from a sparse graph. This sparse graph includes edges whose costs are lower than a granularity threshold value $\nu = \beta\bar{z}$ (where β is a sparsification factor and $\bar{z}$ is the average cost of the edges) and edges belonging to the best feasible tour. More edges are considered for the VRP, but they are not relevant to the TSP.

GENIUS [31] is a tour construction algorithm, followed by a post-optimization step, consisting of removing a vertex from a feasible tour and re-inserting it. When constructing the tour, each vertex can only be inserted next to vertices belonging to its p-neighborhood, i.e., the p closest vertices in the tour. Limiting the search to a vertex's p-neighborhood is necessary to reduce the number of possible tours resulting from its insertion. The use of this nearest neighbor heuristic resembles sparsification, since the resulting tours will cover a sparse subset of the graph's edges, namely the edges between a vertex and its nearest neighbors.

Hougardy and Schroeder [32] also proposed an algorithm for removing edges form TSP graphs. Their algorithm consists of proving that an edge (p, q) is useless and can be removed, by identifying two points r and s such that, if (p, q) and any two edges incident to r and s respectively are part of a tour, there exists a valid 3-opt move that improves this tour by replacing (p, q). If such two points r and s exist, then (p, q) cannot belong to the optimal tour.

2.1.3 2-matching relaxation

The 2-matching relaxation is obtained for a TSP model by removing the sub-tour elimination constraints, while maintaining integrality constraints. As implied by the removal of sub-tour elimination constraints, solutions of this resulting 2-matching problem are collections of disjoint cycles over all vertices. Figure 2.2 shows an example of a 2-matching solution on the simple graph in Figure 2.1.

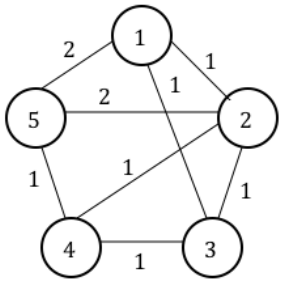

Figure 2.1 Full graph

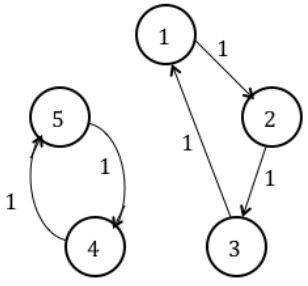

Figure 2.2 2-matching solution

Edmonds [33] provides an algorithm for solving this problem in polynomial time.

2.1.4 1-tree relaxation

Given a vertex of an undirected graph, say vertex 1, a 1-tree is a tree spanning the vertices in $V\backslash\{1\}$, with two additional edges incident with vertex 1. The relaxation can be obtained by removing degree constraints from the TSP.

We can see intuitively that a 1-tree has at most one cycle and that a 1-tree composed of a minimum-cost spanning tree and two edges of minimum cost provides a lower bound on the optimal cost of the TSP. A minimum-cost 1-tree can be computed in polynomial time and has been used in a breakthrough algorithm by Held and Karp [34, 35].

2.1.5 n-path relaxation

This relaxation was introduced by Christofides et al. [36] and is generally used with a path representation of the problem. A vertex, say vertex 1, is considered the starting point of the path and duplicated, as to transform a tour into a path from vertex 1 to itself. From this representation, the n-path relaxation is obtained by removing degree constraints on every vertex, while enforcing that $|V| = n$ edges need to be selected in the path. A shortest path can be found in polynomial time through dynamic programming, but the resulting path will not necessarily be elementary, meaning some vertices may be visited multiple times, and as a result some vertices will not be visited.

The n-path relaxation has been used in column generation approaches for complex routing problems, like the Capacitated Vehicle Routing Problem and its variants.

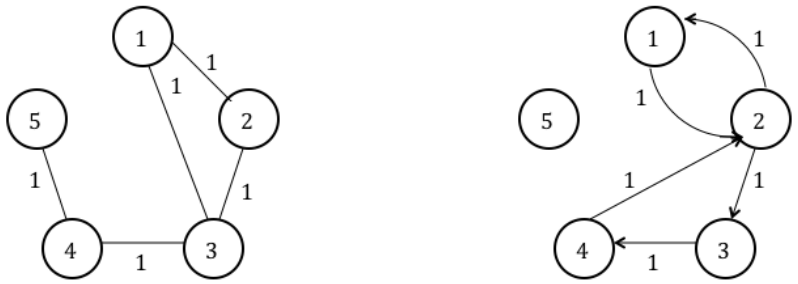

Figure 2.3 Solutions for the 1-tree and n-path

Figure 2.3 shows example solutions for the 1-tree and n-path relaxations for the graph in

Figure 2.1.

2.2 Counting-based search

In this section, we review the existing CP optimization constraints that can be used to capture the combinatorial structure of the TSP relaxations presented previously and the methods for computing solution densities on these constraints.

2.2.1 Cost-based solution densities

Counting-based search was originally introduced by Pesant et al. [2] for solving constraint satisfaction problems and was then extended for constraint optimization problems [37]. Cost-based solution densities are computed in counting-based search and serve as the criterion for selecting branching variables and values. Given a constraint $c(x_1, \ldots, x_k)$ on finite-domain variables $x_i \in \mathcal{D}_i$ $_{1 \leq i \leq k}$, let $f : \mathcal{D}_1 \times \cdots \times \mathcal{D}_k \to \mathbb{N}$ associate a cost to each k-tuple t of values for the variables appearing in that constraint and z be a finite-domain cost variable. An *optimization constraint* $c^\star(x_1, x_2, \ldots, x_k, z, f)$ holds if $c(x_1, x_2, \ldots, x_k)$ is satisfied and $z = f(x_1, x_2, \ldots, x_k)$. Let $\epsilon \geq 0$ be a small real number and $z^\star = \min_{t \,:\, c(t)} f(t)$ (without loss of generality consider that we are minimizing). We call

$$\sigma^\star(x_i, d, c^\star, \epsilon) = \frac{\sum_{t=(x_1,\ldots,x_{i-1},d,x_{i+1},\ldots,x_k) \,:\, c^\star(t,z,f) \wedge z \leq (1+\epsilon)z^\star} \omega(z, z^\star, \epsilon)}{\sum_{t=(x_1,\ldots,x_k) \,:\, c^\star(t,z,f) \wedge z \leq (1+\epsilon)z^\star} \omega(z, z^\star, \epsilon)}$$

the *cost-based solution density* of variable-value pair (x_i, d) in $c^\star$ given ϵ. This value is found between 0 and 1, and represents how often the assignment $x_i = d$ is found in assignments satisfying constraint $c^\star$. The number ϵ serves as a margin for including close-to-optimal solutions in the ratio. If $\epsilon = 0$, then the solution density is restricted to optimal solutions of the constraint, with respect to f. Otherwise, if ϵ is positive, then sub-optimal solutions can be included but are discounted proportionally to their distance from $z^\star$, the minimum value of f over solutions of c. This discount is given by the weight function $\omega(z, z^\star, \epsilon)$, which can vary depending on the constraint.

2.2.2 minWeightAllDifferent constraint

The collection of minimum-cost disjoint cycles for the integer programming model mentioned in Section 2.1 can be translated in constraint programming terms by a minimum-cost assignment, and can be implemented as a `minWeightAlldifferent` optimization constraint, as defined in Caseau and Laburthe [38].

Pesant [37] describes how to compute solution densities for this optimization constraint. We first compute a minimum-weight bipartite matching using the Hungarian algorithm. This computation also yields a reduced-cost matrix $R = (r_{ij})$ as a by-product. The non-negative r_{ij} indicate the expected increase in cost if the variable/value assignment (i, j) replaces an assignment of the computed matching. Given $z^\star$, the cost of the minimum-weight matching, we define the related matrix R' where :

$$r'_{ij} = \max(0, \frac{(\epsilon z^\star + 1) - r_{ij}}{\epsilon z^\star + 1})$$

Finally, to compute the solution densities of the constraint, we must compute the permanent of matrix R'. The permanent of an $n \times n$ matrix $A = (a_{ij})$ is defined as :

$$per(A) = \sum_{p \in P} \prod_{i=1}^{n} a_{ij}$$

where P is the set of permutations of $\{1, 2, ..., n\}$. In the case of a binary matrix where a "1" entry indicates that value j is in the domain of i, the permanent represents the number of solutions to the constraint, as we sum over every assignment and the product equals 1 only if each variable has the assigned value in their domain. With our matrix R', optimal assignments count as 1, assignments whose cost exceeds $(1 + \epsilon)z^\star$ count as 0 and other assignments are counted at a discount proportional to its distance from the optimum. Thus, we obtain a weighted counting of feasible assignments within our defined margin ϵ.

The cost-based solution density of a variable-value assignment $x_i = d$ is computed as the ratio of two permanents, the permanent of $R'_{x_i=d}$ over that of R', where $R'_{x_i=d}$ sets all values in column i except in row d to 0. Since computing permanents is $\#P$-complete, we will use an upper-bound defined in Soules [39], as :

$$U^1(A) = \prod_{i=1}^{n} t(A_i)\mu(m(A_i), s(A_i)/t(A_i))$$

where $s(A_i)$ denotes the sum of the entries in A_i, the i^{th} row of A, $t(A_i)$ denotes the maximum entry in A_i, $m(A_i)$ denotes the number of positive entries in A_i and $\mu(m, z)$ denotes the geometric mean of m numbers, equally-spaced from 1 to z.

2.2.3 minWeightSpanningtree constraint

The 1-tree structure can be represented in CP using a `minWeightSpanningtree` constraint. When implementing the 1-tree into our model, it can be represented by building a new graph with $n+1$ vertices, by duplicating the vertex that would be excluded, and finding a minimum spanning tree over the whole graph. To maintain the 1-tree structure with this representation, it is preferable to remove the vertex with the largest average distance from other vertices, to hopefully guarantee that said vertex and its duplicate each only have one incident edge. Theoretically, this does not guarantee a perfect 1-tree structure, but the relaxation works well in practice regardless. Figure 2.4 shows an example of a spanning tree on this modified graph, from the original graph on Figure 2.1.

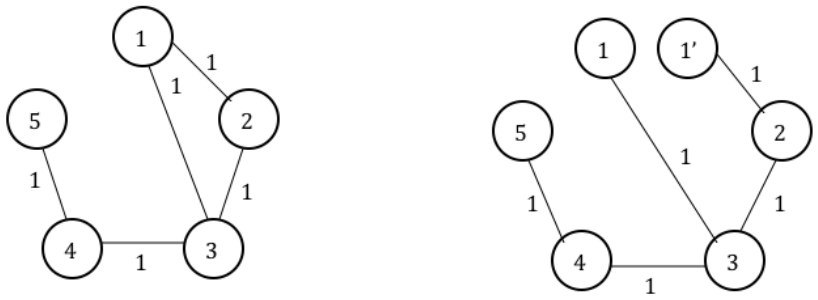

Figure 2.4 Comparing 1-tree with spanning-tree on modified graph

We will now present work done on counting weighted spanning trees in CP. Delaite and Pesant [40] propose a method for computing the solution densities of an edge in a weighted spanning tree.

They start by defining a matrix $M = (m_{ij})$ where m_{ij} are univariate polynomials defined from edge set E and weight function w defined over E :

$$m_{ij} = \begin{cases} -x^{w(e)}, & i \neq j \quad e = (v_i, v_j) \in E \\ 0, & i \neq j \quad (v_i, v_j) \notin E \\ \sum_{e=(v_i,v_k)\in E} x^{w(e)}, & i = j \end{cases}$$

Broder and Mayr [41] proved that any minor of M (i.e., the determinant of the submatrix obtained by removing any row and column i from M) yields a polynomial $\sum_k a_k x^k$ where the monomial $a_k x^k$ indicates that there are a_k spanning trees of cost k in the graph. Rather than computing this minor on a matrix of polynomials, which can be time-consuming, Delaite

and Pesant chose to instantiate x to real value between 0 and 1, to work with a matrix of scalars. This value also serves to apply an exponential decay to the number a_k according to the difference between the weight k and the weight of an optimal spanning tree, benefiting close-to-optimal trees over trees of higher weight. An x value of 1 would not apply any decay and count every tree equally and the decay becomes more aggressive as it approaches 0.

To compute the solution density of an edge, Brockbank, Pesant and Rousseau [42] describe an efficient method, exploiting the fact that the matrix M of a graph without that edge is nearly identical to the original. In order to compute the solution density of a given edge $(i, j) \in E$, we can take the ratio of the number of spanning trees without that edge over the total number of spanning trees. Consider the matrix M' defined similarly to the matrix M above, but over the edge set $E' = E \backslash (i, j)$. The solution density of edge (i, j) not being in the spanning tree can be calculated as a ratio of any minor of M' over a minor of M. Since M' is identical to M except for entries m_{ii}, m_{ij}, m_{ji} and m_{jj}, removing row and column i from M and M' leaves us with two sub-matrices, say N and N', that are nearly identical except for entry m_{jj}. The Sherman-Morrison formula states that if matrix N' is obtained from N by replacing column $(N)_j$ by column vector u, then :

$$\det(N') = (1 + e_j^T N^{-1}(u - (N)_j)) \det(N)$$

By using this formula, the solution densities of every edge can be computed with only a linear (in terms of vertices) number of determinant computations, rather than quadratic.

2.2.4 cost-regular constraint

The n-path relaxation can be represented in CP with a `cost-regular` constraint, a weighted variant of the `regular` constraint. The constraint is applied on variables $\langle x_1, x_2, ..., x_n \rangle$, where x_i is the vertex in ith position on the path and the TSP instance's input graph serves as the automaton.

Pesant [37] proposes an algorithm for computing cost-based solution densities for this constraint. The domain filtering algorithm for the `regular` constraint builds a layered digraph by unfolding the automaton over the sequence of variables. A path from the first layer to the last layer of the digraph corresponds to a solution, which in our case is an n-path. The cost-based solution density algorithm needs to find paths whose costs are at most $(1+\epsilon)z^\star$, where $z^\star$ is the cost of the shortest n-path. At every node of the digraph, it computes the number of incoming and outgoing partial paths of costs up to $(1+\epsilon)z^\star$. The number of relevant paths

containing a given variable-value pair is computed as the sum over all corresponding arcs of the products of number of partial incoming and outgoing paths at their endpoints, provided their composition makes an n-path of cost at most $(1 + \epsilon)z^\star$.

CHAPTER 3 IMPLEMENTATION

This chapter will describe our contribution in this field. We will mention the CP model implemented to use and test our method, the different algorithmic options explored for our pre-processing method, and lastly we will go over methods to improve the computing time of cost-based solution densities.

3.1 Description of our CP model

In order to apply our method, we need a functioning CP model which will be used for computing the cost-based solution densities of the global constraints drawn from TSP relaxations as mentioned previously. This model is implemented in IBM ILOG CP Solver 1.6.

3.1.1 Initial model with minWeightAllDifferent constraint

We start off with a basic CP model:

$$
\begin{array}{ll}
\min z = \sum_{i=1}^{n} \gamma_{is_i} \quad \text{s.t.} & \\
\texttt{minWeightAlldifferent}(\{s_1, \ldots, s_n\}, z, \Gamma) & \\
\texttt{noCycle}(\{s_1, \ldots, s_n\}) & \\
s_i \in \{2, 3, \ldots, n+1\} & 1 \leq i \leq n \\
z \in \mathbb{N} &
\end{array}
$$

Table 3.1 A basic CP model for the TSP

where Γ is a cost matrix and s_i are our variables. These are successor variables, i.e., the value of s_i is the city following city i in the tour. We implement this model as follows:

```
next = IloIntVarArray(env, n);
IloIntArray domain = IloIntArray(env)
for(int i=0; i<n; i++){
    domain.clear();
    for(int j=1; j<n+1; j++){
        if(instance[i][j] != 0)
            domain.add(j);
    }
```

```
    next[i] = IloIntVar(env, domain);
}
```

We start by defining the successor variables s_i as the array of n integer variables `next` in environment `env`. We define an empty domain (`domain.clear()`) for each variable s_i and if there is an edge between vertex i and j (`instance[i][j] != 0`) we add j to the domain. We end the iteration by defining the individual variable `next[i]` with the appropriate `domain` parameter. It should be noted that the values in the domain of `next` range from 1 to n, vertex n being a duplicate of vertex 0, rather than 0 to $n-1$.

```
costTotal = IloIntVar(env, 0, IlcIntMax);
IloIntVar cx(env, 0, IlcIntMax);
for(int i=0; i<n; i++){
    model.add(cx[i] == cost[i][next[i]-1]);
}
model.add(costTotal == IloSum(cx));
```

Then, we define the cost function of our model, as the integer variable `costTotal` which can take any value from 0 to the maximum integer value. It is defined as the sum of variables `cx`, themselves defined by the constraint `cx[i] == cost[i][next[i]-1]`, where `cost` is the cost matrix of the instance. Overall the code represents the function $z = \sum_i c_{is_i}$.

```
addCountableConstraint(IloCostAllDiffCounting(env, next, getRandomGenerator,
                                                cost, costTotal, 0.1));
model.add(IloNoCycle(env, next));
```

Finally, we add our global constraints to the model. The `minWeightAllDifferent` constraint is implemented in CPO as the function `IloCostAllDiffCounting`, taking as parameters the environment, the `next` variables, a random generator, the instance's cost matrix, the cost function and a `lowCostGap` parameter, whose value is 0.1 here and which represents ϵ from the definition of SDs, as seen in Section 2.2.1. The sub-tour elimination constraint `IloNoCycle` is implemented as follows:

```
int i = _next.getIndexValue();
int j = _next[i].getValue();

end[start[i].getValue()].setValue(solver,end[j].getValue());
length[start[i].getValue()].setValue(solver,
```

```
    length[start[i].getValue()].getValue()+
    length[start[j].getValue()].getValue()+1);
start[end[j].getValue()].setValue(solver,start[i].getValue());

if (end[j].getValue() < n) {
    _next[end[j].getValue()].removeValue(start[i].getValue());
}
else {
    if (length[0].getValue()+length[start[i].getValue()].getValue()+1 < n)
       _next[end[0].getValue()].removeValue(start[i].getValue());
    }
if ((start[i].getValue() == 0) &&
    (length[0].getValue()+length[start[n].getValue()].getValue()+1) < n)) {
    _next[end[0].getValue()].removeValue(start[n].getValue());
}
```

When `next[i]` is fixed to `j`, the partial path ending at i is merged with the partial path starting at j. This merging is done in the first three lines, where the end of the path that started at j also becomes the end of the path that ended at i and the start of the path that ended at i also becomes the start of the path that started at j. A valid tour would be defined as a path from 0 to n, its duplicate. The three `if` conditions serve to prevent sub-tours by removing values from certain domains if those conditions are met. If a partial path does not end at n, then its end vertex cannot be followed by its start (see Figure 3.1). If a partial path does end at n, but the sum of its length and the length of the path starting at 0 is less than $n-1$, then the end vertex of the path starting at 0 cannot be followed by the start of the path ending at n. The last `if` checks the same thing as the second. In the second, the index i is in the partial path ending in n, whereas, in the third, the index i is in the partial path starting in 0.

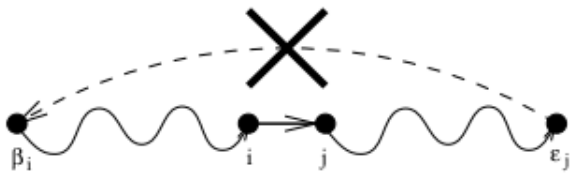

Figure 3.1 Eliminating subtours. Reprinted from [1]

By branching on the variables `next` and using counting-based search, the program yields the cost-based solution densities corresponding to the `minWeightAlldifferent` constraint. For pre-processing, we are only interested in the first iteration of cost-based solution densities, so we can stop the execution of the solver as soon as that first iteration is over.

3.1.2 Additional minWeightSpanningTree and cost-regular constraints

Next, we want to add to this model the `minWeightSpanningtree` constraint corresponding to the 1-tree relaxation. This constraint is defined in such a way that the `next` variables defined above are not compatible, so a new set of variables must also be added to the model. These variables will be binary variables e_{ij}, whose value is 1 if the edge (i, j) is in the solution tour and 0 otherwise. It is implemented as follows:

```
edgevars = IloIntVarArray(env, nbEdges, 0, 1);
model.add(edgevars);
model.add(IloSum(edgevars) == n );
```

The variables e_{ij} are implemented as an array `nbEdges` (number of edges in the graph computed when reading the instance) variables, with lower bound 0 and upper bound 1. We add the constraint $\sum_{i,j} e_{ij} = n$ to strengthen the model.

```
treeCost = IloFloatVar(env, 0, IlcFloatMax);
model.add(treeCost);
```

`treeCost` will be the objective value of the spanning tree constraint, taking any floating-point value from 0 to the max value.

```
addCountableConstraint(IloSpanningT(env, edgevars, treeCost, graph,
                                    getConfiguration()->getHeuristic()));
```

The minWeightSpanningTree constraint, implemented as `IloSpanningT`, takes as parameters the environment `env`, the decision variables `edgevars`, the objective value `treeCost`, the input graph `graph` and a `Heuristic` parameter, which helps determine the value of variable x (2.2.3) depending on whether or not the heuristic is cost-aware or not.
In case we want to solve the CP model using both sets of variables (4.2.1), another set of constraints is necessary to link the two sets of variables:

```
for(int i=0; i<n; i++){
    int n1 = edges[i].n1;
    int n2 = edges[i].n2;
    model.add(edgevars[i] == (next[n1%n]==n2)||(next[n2%n]==n1))
}
```

`edges[i].n1` and `edges[i].n2` are the extremities of edge `edges[i]`. For each edge, we add a constraint which states that if that edge is in the solution (`edgevars[i]==1`) then either `next[n1%n]==n2` or `next[n2%n]=n1`. The reason for the n modulo is that the graph for the spanning tree has a vertex n, a duplicate of vertex 0.

Lastly, we want to implement the `cost-regular` constraint that corresponds to the n-path relaxation. Again, this constraint also requires its own set of variables. The value of these variables p_i represents which vertex is found in the i^{th} position of the tour. We implement the constraint as:

```
state = IloIntVarArray(env, n, 0, n-1);
model.add(state);
pathcost = IloFloatVar(env, 0, IlcFloatMax);
model.add(pathcost);
addCountableConstraint(IloRegular(env, state, automaton,
                                  auto_cost, pathcost, 0.1))
```

The n integer `state` variables represent the variables p_i and can take a value from 0 to $n-1$, representing any of the vertices. The variable `pathcost` is the objective value of the n-path. The constraint `IloRegular` takes as parameters an environment variable `env`, the decisions variables `state`, an `automaton` which represents the input graph, the cost function `cost_auto` of said automaton, the objective value `pathcost` and a `lowCostGap` parameter, arbitrarily set at 0.1.
We also link the variables with the successor variables like this:

```
model.add(next[0] == state[0]);
for(int i=1; i<n-1; i++){
    model.add(next[state[i-1]] == state[i])
}
model.add(next[state[n-2]] == n);
model.add(state[n-1] == 0);
```

The starting point of the automaton is vertex 0, so the first decision `state[0]` corresponds to `next[0]`. `state[i]` follows `state[i-1]` in the path, hence `next[state[i-1]] == state[i]`. Finally, for the path to close, we must have `state[n-1] == 0` and `next[state[n-2]] == n`, keeping in mind that, in the domains of the `next` variables, n represents vertex 0.

Finally, we may want to solve the CP model using a counting-based search branching heuristic which takes into account several of these constraints. However, because of the differing structures of these constraints and their variables (linear number of integer successor variables v.s. quadratic number of binary edge variables for example), it is necessary to add a function homogenizing the cost-based SDs of these constraints. For this, we use the functions `IloEdge2Next` and `IloPos2Next` which associate the SDs of `edgevars` from `SpanningT` and `state` from `Regular` to their corresponding `next` variable.

`IloEdge2Next` is somewhat intuitive since `next[i]=j` and `edgevars[i,j]=1` both represent the same edge being present in the tour. The only modification needed is normalizing the solution densities to guarantee $\sum_j(\texttt{next[i]=j}) = 1$. The function `IloEdge2Next` is essentially an implementation of the formula:

$$SD(\texttt{next[i]=j}) = \frac{SD(\texttt{edgevars[i,j]=1})}{\sum_{k \in dom(\texttt{next[i]})} \texttt{edgevars[i,k]=1}}$$

Unlike the `next` and `edgevars` variables, the `state` variables do not represent an edge of the tree but rather the position of a vertex in the tour. The idea behind the algorithm of `IloPos2Next` is that, if k is a likely position for vertex i and $k+1$ is a likely position for vertex j, then (i, j) is likely to be an edge in the tour. We get the initial result:

$$SD'(\texttt{next[i]=j}) = \sum_{k \,:\, i \in dom(\texttt{state[k]})} \frac{SD(\texttt{state[k+1]=j})}{|dom(\texttt{state[k]})|}$$

Essentially, we consider the solution densities of vertex j being in position $k+1$ of the tour for every possible position k of vertex i. The reason for dividing by the size of the domain of `state[k]` is that, if `state[k+1]=j`, any vertex that can be found in position k could be a predecessor of j, not necessarily i. Therefore, the SD of `state[k+1]=j` is divided equally among all possible predecessors.

Finally, we must normalize the SDs:

$$SD(\texttt{next[i]=j}) = \frac{SD'(\texttt{next[i]=j})}{\sum_k SD'(\texttt{next[i]=k})}$$

3.2 Using SDs for pre-processing

Thanks to the model described previously, we can extract solution densities for all three of the constraints/relaxations observed. We will now explain how we exploit these solution densities to clean up input graphs of TSP instances.

By ordering the set of solution densities of one constraint, we effectively get a ranking of the most promising edges according to that relaxation. We call a promising edge an edge that is found in many good solutions satisfying the constraint. Our objective is to remove edges from the graphs of TSP instances, according to the edges that each constraint finds promising. Lastly, we would like our method to remove as many edges as possible from the graph, making it as sparse as possible, while minimizing the risk of removing edges that would be in the optimal tour of the instance. We must remember that the solution densities we obtained come from relaxations of the TSP and not the problem itself. Therefore, an edge found in the optimal solution of the TSP is not necessarily found in many good solutions of one of its relaxations.

3.2.1 Selecting good edges

The first step to the pre-processing is defining a threshold as to what a promising edge is, in other words, how many edges should we consider from each set of cost-based solution densities. The most intuitive option is taking a percentage of all edges. For example, we could take the top half or the top quartile of edges with regards to their solution densities. One advantage of this method is that, by taking the same percentage for each relaxation, we can guarantee that each relaxation is considered equally, but conversely, we could also modify the percentages to give a relaxation more representation than another, in case a relaxation makes better decisions than another.

Another criterion for defining a promising edge is setting a threshold on the value of their solution densities, meaning we would only consider edges above a certain solution density. This method poses some problems however. First of all, the average solution density can vary according to the size of the instance but also according to the constraint itself. For

instance, SDs for the `Spanningtree` constraint are very heterogeneous since the variables are binary, and so, we will often see assignments $e_{ij} = 0$ with very large SDs (> 0.9) whereas their respective $e_{ij} = 1$ assignment will be very low (< 0.1). This method will also yield an inconsistent number of edges for a given threshold, which will make it more difficult to control the dimension of the sparse graph. As you can see on Figures 3.2 and 3.3, the larger pre-processed graph ends up being considerably sparser despite a more lenient threshold on the solution densities. Indeed, the average SD tends to drop as the size of the instance rises. Considering that the edge density of the pre-processed graph affects how likely we are to preserve the optimal tour, finding a threshold that can be relatively aggressive while maintaining optimality for every instance seems unlikely, and recalculating an appropriate threshold per instance is not an interesting alternative.

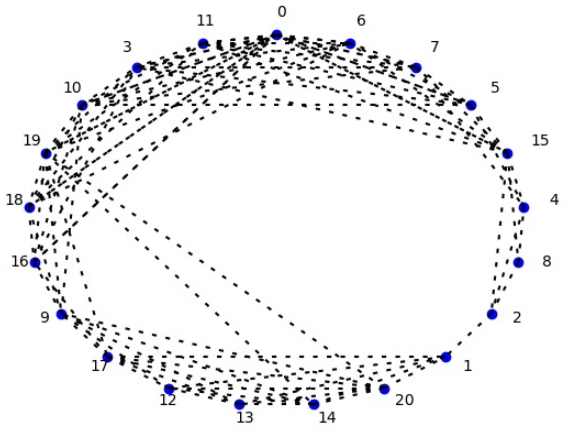

Figure 3.2 Removing edges of SD below 0.1 in *gr21*

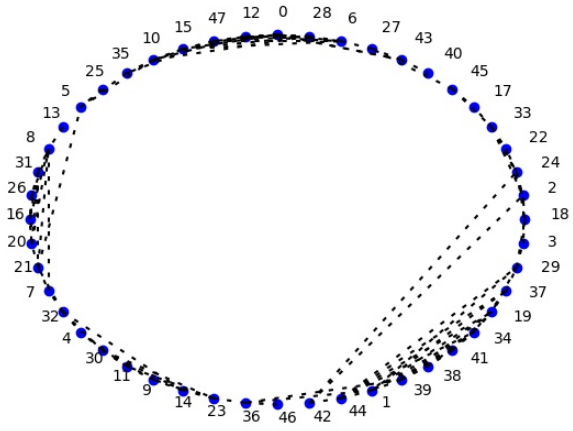

Figure 3.3 Removing edges of SD below 0.075 in *gr48*

Now, we propose a more robust criterion for selecting promising edges which tends to reduce the risk of removing edges that would be part of the optimal tour. Rather than taking a percentage of edges overall, we will consider a percentage of the edges incident with each vertex. The idea is that in many cases, the distribution of solution densities over each vertex of the graph may be uneven and a certain vertex may have twice as many incident edges in the top quartile of solution densities than another. Since an optimal tour must have two edges incident to each vertex, having certain vertices with fewer incident edges in our sparse graph increases the chance that we lose the optimal tour around these vertices. Therefore, if we take the top quartile of the edges incident to each node rather than a quartile of the

edges overall, our pre-processing method will prove to be much more robust and will have more chance of keeping the optimal tour intact.

On Figures 3.4 and 3.5, the optimal tour of the instance is represented as the perimeter of

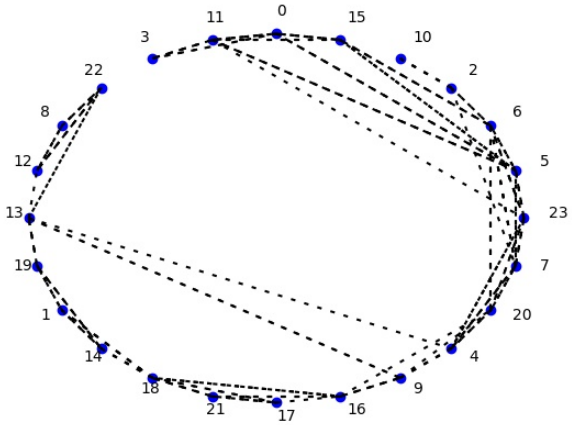

Figure 3.4 15% best SDs overall for the TSP instance *gr24*

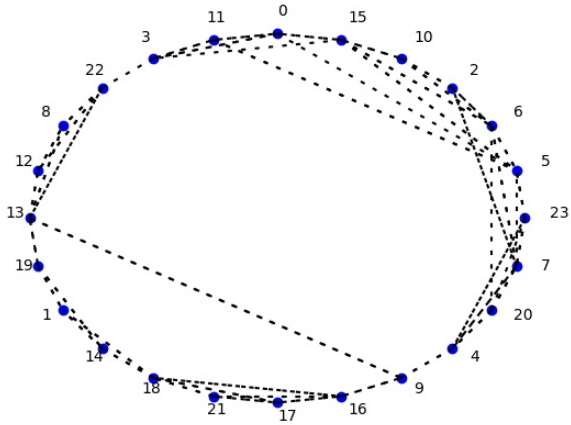

Figure 3.5 15% best SDs per vertex for the TSP instance *gr24*

the circle. As we can see, when considering the same number of edges (15% of edges) for each method, we lose two edges, $(22, 3)$ and $(15, 10)$, from the optimal tour in the former method while keeping this tour intact in the latter. Looking closely, we can see that vertex 10 only has one incident edge, so not only would we not get an optimal solution, but the problem would be infeasible. Though this is only one example, it clearly shows the merit of distributing the edges evenly among the vertices of the graph. One slight inconvenience of the later method is that it may be slightly more time consuming than the others.

Now that we have determined which edges are promising for each relaxation, we combine this information to decide which edges to keep in our final sparse graph and which to remove. We explore two options for combining the sets of promising edges from each relaxation.

The first option is to keep the edges in the intersection of these sets. In other words, we would only keep edges considered promising by each relaxation. The second option is to keep the edges in the union of the sets, which means we would only remove edges considered unpromising by each relaxation.

The second option proves to be much better than the first. As mentioned previously, the

solution densities we are working with are taken from relaxations of the problem and not the problem itself, therefore a relaxation could consider poor an edge that is actually part of the main problem's optimal solution. If any of the relaxations does make such a mistake, then, by only keeping edges considered promising by all relaxations, we run a high risk of losing the optimal tour and the only way to prevent that would be to increase the threshold until those missing edges are recovered. Thus, in some cases, using the first option could require a much denser graph than we would like to have. On the other hand, if we opt for the second option then there is a chance that if one relaxation does not find an optimal edge promising, another relaxation will make up for it and, in practice, we find that it is rare for edges in the optimal tour of the TSP to be severely misjudged by two different relaxations.

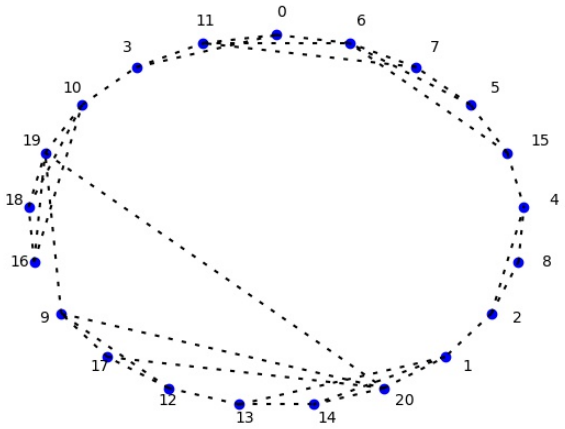

Figure 3.6 Top 15% of edges according to `Alldifferent`

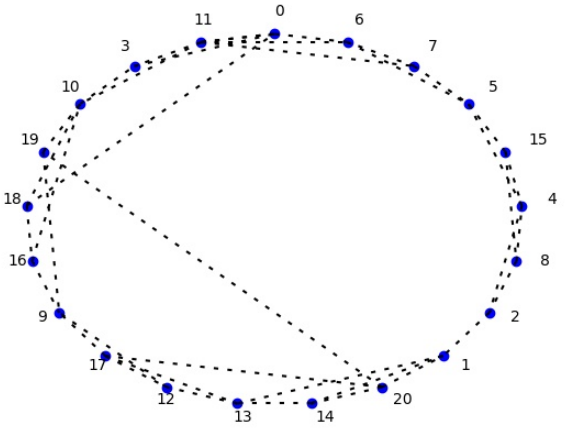

Figure 3.7 Top 15% of edges according to `Spanningtree`

In the graphs in Figures 3.6 and 3.7 , we can see that, in this case, the `Alldifferent` and `Spanningtree` constraints each consider an edge in the optimal tour ((9,16) for `Alldifferent` and (18,19) for `Spanningtree`) not to be in the top 15% of edges. If we were to pre-process by taking the intersection of both, we would lose two edges and, in order to restore the optimal tour, we would need to increase the number of edges taken from each relaxation. However, by taking the union, each relaxation covers the other's mistake and we can maintain the optimal tour without increasing the number of edges from each relaxation.

3.2.2 Choice of relaxations

As stated in Section 3.1, the CP model we exploit has three global constraints that we can extract cost-based solution densities from, but we will only be using the ones derived from the 2-matching and 1-tree relaxations, and ignore the n-path relaxation. A solution density is linked to a variable-value assignment and, in the case of the `Alldifferent` and `Spanningtree` these assignments define edges very intuitively: $s_i = j$ and $e_{ij} = 1$ represent edge (i, j) directly. Secondly, the structures of these relaxations' solutions still hold many of the characteristics of the TSP's structure and thus edges in the TSP's solution could be common in the solutions of these relaxations. Specifically, in both relaxations, all vertices are "visited" by the solution, i.e., there are edges incident to each vertex. Additionally, the 2-matching conserves the structure of tours, with exactly 2 incident edges per vertex and the 1-tree's structure contains exactly one tour.

On the other hand, the n-path relaxation poses various problems. First of all, the variables for the `Regular` constraint representing the n-path indicate the vertex in a certain position of the tour and thus do not provide direct information on edges of the tour. There is a way to go around this problem to gain information on edges. If vertex i is a promising value for variable pos_k and vertex j is a promising value for variable pos_{k+1}, then edge (i, j) is promising. We can define a recursive algorithm where we consider vertex 0 as a starting point and we add edges from 0 to good values of position 1, from these vertices to good values of position 2, and so on and so forth.

However one finds that even with this workaround, the structure of the relaxation itself yields undesirable results with regards to which edges it finds promising. The best n-path solutions will often be non-elementary paths, since we allow the paths to ignore vertices and pass multiple times through other vertices. As a result, there will often be a subset of vertices, with small distances from one another relative to the rest of the graph, and all the best solutions of the n-path will be various ways of looping around those same vertices. Therefore all the best assignments, in terms of solution density, will be assignments where, for any i, pos_i is in this subset of vertices. This leads to very poor edge selection from this relaxation.

The graph in Figure 3.8 shows a primitive method of pre-processing using the n-path relaxation. We considered the top 25% of position assignments and built the edges recursively over these positions as defined previously. We can see many of the vertices are excluded and we are left with a dense graph over a small subset of vertices.

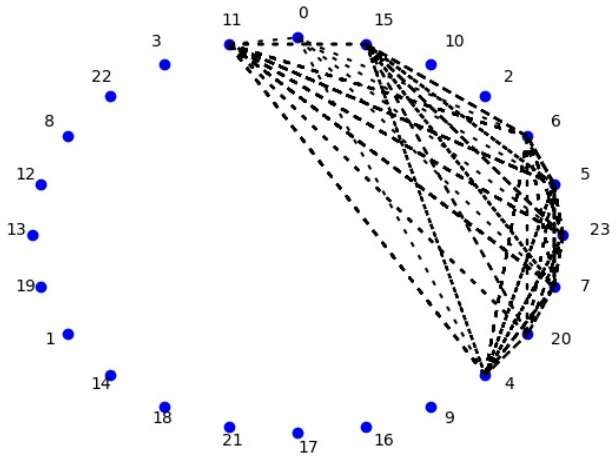

Figure 3.8 Simple pre-processing of *gr24* from `Regular` constraint's solution densities

There is a better way of exploiting the `Regular` constraint, though its results remain difficult to use. Using the function `IloPos2Next` mentioned in Section 3.1, we can translate the SDs of the `Regular` constraint into SDs of successor variables. Figures 3.9 and 3.10 show what we get on the same instance as before by pre-processing taking the best SDs from this conversion.

As we can see, despite these modifications the edges considered promising remain uninteresting and ignore much of the optimal tour. `IloPos2Next` computes the SD of edge (i, j) as the sum over k of the product of SDs for $pos_k = i$ and $pos_{k+1} = j$. Since, the best SDs of pos_k for every k have the same small subset of vertices as their value, then every edge incident to those values is heavily favored. Even by distributing edges more evenly over the vertices (figure 3.10), the overall result is the same. We therefore concluded that the `Regular` constraint's solution densities provide no viable information for pre-processing.

Therefore, in our tests, we will only pre-process the instance by combining SDs from `Alldifferent` and `Spanningtree`. Regardless, these two relaxations prove to be sufficient for a relatively aggressive yet robust pre-processing. By taking into account our best options, i.e., taking $k\%$ of edges incident to each vertex and considering the union of SDs from `Alldifferent` and `Spanningtree`, we are able to eliminate a large number of edges, while having relatively low chances of losing the optimal tour.

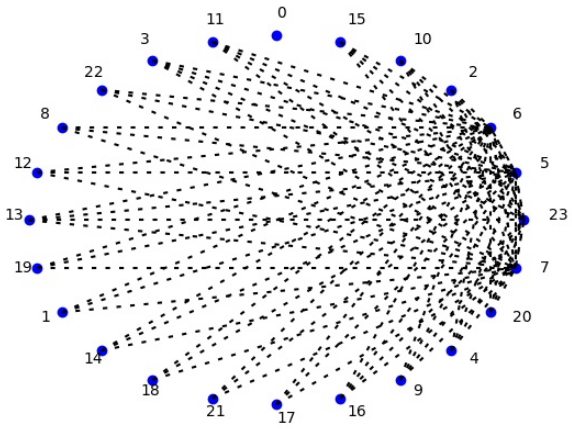

Figure 3.9 15% best SDs overall from `IloPos2Next`

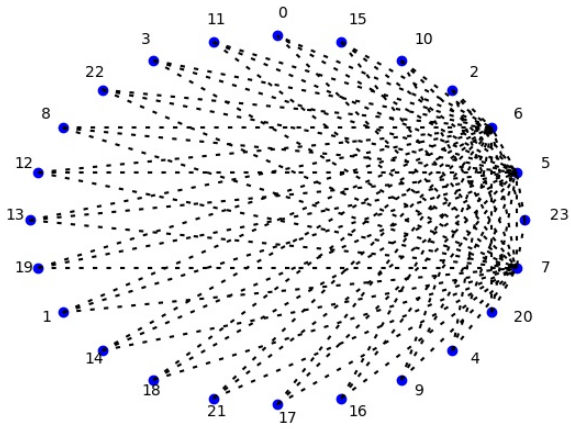

Figure 3.10 15% best SDs per vertex from `IloPos2Next`

3.2.3 Full algorithm

To summarize our algorithm, we begin by computing solution densities for the 2-matching and 1-tree relaxations, order the edges of the graph from largest to smallest SD, in 2 different lists depending on the relaxation and decide on the percentage of edges we want to keep per vertex. Then, for each relaxation, we go through their list of edges, edge by edge, and if either of its extremities have less selected edges than the minimum percentage we defined then the edge is added to the final list of selected edges. We can stop looking through the list once all vertices have met their quota.

```
list1 := list of edges sorted by SD from AllDifferent
list2 := list of edges sorted by SD from SpanningTree
limit := number of edges to keep per vertex
count := [limit for each vertex]
edge_list := []
loop :
    (i,j) := next in list1
    if count[i]>0 or count[j]>0 :
        add (i,j) to edge_list
        if count[i]>0 : count[i]--
```

```
        if count[j]>0 : count[j]--
    if sum(count) = 0 :
        exit loop
count := [limit for each vertex]
loop :
    (i,j) := next in list2
    if count[i]>0 or count[j]>0 :
        add (i,j) to edge_list
        if count[i]>0 : count[i]--
        if count[j]>0 : count[j]--
    if sum(count) = 0 :
        exit loop
return edge_list
```

To show our algorithm's effectiveness for pre-processing, we compare our method (Figure 3.11) to a much simpler method of eliminating edges (Figure 3.12) which only considers the cost of the edges. As we can see for this particular instance, our approach yields a much sparser graph without losing the optimal tour, whereas a simpler method loses edge $(0, 24)$ despite keeping twice as many edges.

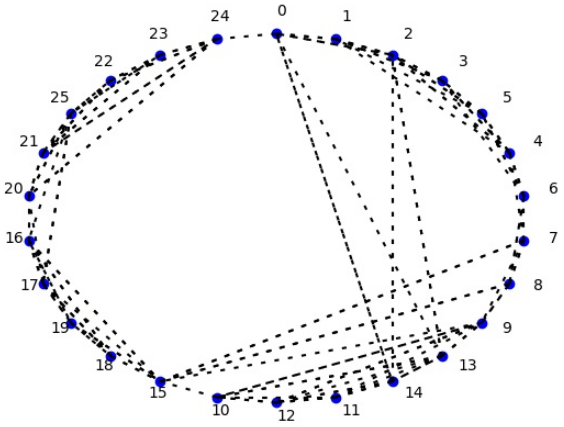

Figure 3.11 Pre-processing taking the top 15% of SDs

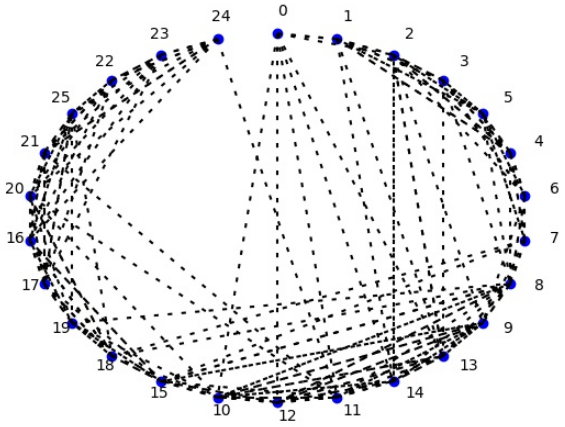

Figure 3.12 Pre-processing taking the top 30% cheapest edges

However, there is, for now, no way of proving whether or not the optimal tour will be preserved

when keeping $k\%$ of edges for a given k. That being said, despite some rare outliers which require keeping up to 50% of edges, most instances of TSPLIB that we experimented with maintain optimality when keeping a fourth of the edges and many of the larger instances can go as low or lower than 10%. Unfortunately, the appropriate percentage can still only be determined empirically. That being said, in the event that too few edges are kept and optimality is lost as a result, this approach may still provide a good solution relatively close to optimal tour.

3.3 Faster alternatives to the original SDs

The first methods for computing cost-based solution densities proved to be quite time consuming when tackling larger instances of our problem, starting at sizes around 500. To test our pre-processing on even larger instances, we required faster alternatives to computing SDs.

First of all, for the `minWeightAlldifferent`, it was found that there was a strong correlation between the solution density of assignment $s_i = j$ and the reduced cost r_{ij} yielded by the Hungarian algorithm (see 2.2.2).

Table 3.2 Comparison of reduced costs and solution densities

instance	m: reduced costs of 0	common edges in top lists	percentage
eil101	205	201	98.05
lin105	170	167	98.24
gr120	201	191	95.02
ch150	264	260	98.48
kroA150	262	257	98.09
kroB150	238	238	100.00
si175	377	377	100.00
brg180	180	180	100.00
rat195	367	367	100.00
ts225	608	608	100.00
pr226	480	480	100.00
a280	688	688	100.00
lin318	518	509	98.26
rd400	676	676	100.00

To compare these two criteria, we count the number m of edges whose reduced costs are equal to 0, i.e., the best m edges according to this criterion, and then count how many of these edges are also found in the top m edges according to solution densities. There are several edges with the same reduced costs, making the ordering of these edges irrelevant, which is why we chose to limit the comparison to edges of reduced cost 0 and consider only which edges are in these top lists, rather than how they are ordered. As we can see in Table 3.2, these two top lists are identical for several instances and the percentage of edges in common between them rarely goes below 98%.
Thus, we concluded that we could rank edges for this constraint according to their reduced cost, rather than the corresponding solution density, which in practice is considerably faster.

Speeding up the computation of cost-based SDs for the `Spanningtree` constraint requires a bit more work, but the new method remains closely related to the algorithm described in Section 2.2.3. That section explains that by applying the Sherman-Morrison formula for each SD computation, the number of determinant computations necessary to compute solution densities can be reduced from a quadratic number to a linear number. This led us to believe that by using it twice for each SD computation we could compute all solution densities with only one determinant computation.

Consider the same matrices M and M' defined in Section 2.2.3. We consider N the minor of M respectively obtained by removing row and column 1. For edges incident with vertex 1, nothing changes from the previous method. The determinant of N is computed traditionally and, for each edge, the determinant of N', obtained by removing row and column 1 from M', is computed by applying the Sherman-Morrison formula once, since m_{jj} is the only difference between the two. And the solution density is computed from the ratio of $\det(N')$ and $\det(N)$.

Unlike the previous method, which would create a new matrix N for every vertex i by removing row and column i from M, we want to keep the same N (without row and column 1). This allows us to continue without any more determinant computations. For every other edge (i, j), the only differences between N and N' are entries n_{ii}, n_{ij}, n_{ji} and n_{jj}, so we can obtain the determinant of N' from the determinant of N from only two applications of the Sherman-Morrison formula since only two columns need to be replaced. We can first consider a matrix L obtained from N by replacing the i^{th} column of N by the i^{th} column of N' and we obtain its determinant by applying Sherman-Morrison:

$$\det(L) = (1 + e_i^T N^{-1}((N')_i - (N)_i)) \det(N)$$

From there, we reach the determinant of N' by replacing the j^{th} column of L (equivalent to

the j^{th} column of N) with the j^{th} column of N':

$$\det(N') = (1 + e_j^T N^{-1}((N')_j - (L)_j)) \det(L)$$

Now every SD has been computed with only one determinant computation and multiple applications of the Sherman-Morrison formula.

Using this method another slight time gain can be implemented. The ratio $\frac{\det(N')}{\det(N)}$ for a given edge (i, j) defines the SD for the edge not being in the solution (assignment $e_{ij} = 0$), since M' is defined as the matrix representing the graph without edge (i, j). Therefore, ranking edges from most to least promising is equivalent to ranking this ratio from lowest to highest. However, when applying this faster method, N is always the same since we only ever remove row and column 1 from M. Since all these ratios have the same denominator, we can rank edges by ranking their determinant of N' from lowest to highest.

CHAPTER 4 EXPERIMENTATION AND RESULTS

In this section, we will present the empirical evaluation of our methods and our implemented algorithms. We tested this on a variety of solvers, from generic CP and MIP solvers to the specialized CONCORDE solver. We also evaluate the potential effect of certain parameters.

4.1 Computation time for SDs and pre-processing

We begin by evaluating the effectiveness of our pre-processing algorithm by observing the time necessary to compute solution densities and the computation time of the pre-processing algorithm itself.

4.1.1 Computing SDs for `minWeightAllDifferent`

First we will evaluate how long it takes to compute the solution densities for the 2-matching relaxation and we will compare the original algorithm, as seen in Section 2.2.2, and the faster alternative, as seen in Section 3.3.

Table 4.1 Computation time of `AllDifferent` SDs and their alternative one for instances of size 150 to 280

instance	time(s) for original SD	time(s) for alternative
ch150	32.414	0.682
kroA150	21.932	0.717
kroB150	20.436	0.635
si175	142.448	0.828
rat195	117.048	0.744
d198	76.584	0.957
kroA200	76.885	0.758
kroB200	84.446	0.961
ts225	158.823	1.302
pr264	102.241	0.995
a280	562.098	1.124

As we can see from Table 4.1, computing the SDs completely is considerably slower than only computing the reduced cost of the bipartite matching. We can also see in Table 4.2 that computing these reduced costs is still very fast on much larger instances.

Table 4.2 Computation time of faster SDs for the 2-matching on larger instances

instance	fl417	pcb442	d493	rat575	p654	d657	u724	rat783	u1060	vm1084	pcb1173
time (s)	2.046	2.387	2.818	3.385	4.541	4.733	5.585	5.821	14.118	15.744	14.203

4.1.2 Computing SDs for `minWeightSpanningTree`

This time, we will evaluate how quickly we can compute SDs for the 1-tree relaxation, using the original algorithm (Section 2.2.3) and the faster algorithm (Section 3.3).

Table 4.3 Computation time of `SpanningTree` SDs and their alternative one for instances of size 150 to 280

instance	time(s) for original SD	time(s) for alternative
ch150	3.085	1.082
kroA150	2.822	1.054
kroB150	2.695	1.106
si175	4.166	1.452
rat195	5.559	1.721
d198	5.884	1.929
kroA200	5.945	1.825
kroB200	5.996	1.941
ts225	9.143	2.355
pr264	14.941	3.655
a280	17.806	4.159

We can see from Table 4.3 that the original computation of SDs for the spanning-tree is quite a bit faster than it was for the `AllDifferent`. That being, although our alternative for computing these SDs does scale better than the original algorithm, there is significantly less improvement than the alternative for the `AllDifferent` and, for larger instances (see Table 4.4), these computations still take a significant amount of time (over 10 minutes for instances of size 1000 and above).

Table 4.4 Computation time of faster SDs for the 1-tree on larger instances

instance	fl417	pcb442	d493	rat575	p654	d657	u724	rat783	u1060
time (s)	12.265	15.684	21.760	52.266	76.069	80.670	124.570	186.409	629.787

4.1.3 Time for pre-processing

Finally we will observe the performance of our pre-processing algorithm described in Section 3.2.3, once the SDs for each relaxation have already been computed.

Looking at Table 4.5, we can see that the pre-processing algorithm itself is relatively efficient, compared to the time required to compute the solution densities beforehand. Although its scaling is worse than the "faster SDs" of the 2-matching, it remains significantly faster than computing SDs for the 1-tree.

Table 4.5 Time (in seconds) necessary to pre-process instances of dimension 150 to 1173

instance	time(s)	instance	time(s)
ch150	0.232	lin318	1.059
kroA150	0.264	rd400	2.130
kroB150	0.312	fl417	3.636
si175	0.359	d493	6.413
rat195	0.389	rat575	6.039
d198	0.526	p654	8.090
kroA200	0.447	d657	10.876
kroB200	0.487	u724	10.805
ts225	1.172	rat783	14.172
pr264	0.767	u1060	34.386
a280	0.789	vm1084	35.576
pr299	0.928	pcb1173	50.268

4.2 CP solver for the TSP

In this part, we evaluate two methods of exploiting solution densities within a CP model. Taking small symmetrical instances from TSPLIB, we first discard edges from the initial graph, then use solution densities for variable branching while using the IBM ILOG CP solver.
It should be noted that the CP model is not competitive with the state-of-the-art IP solvers when it comes to the TSP, but remains useful for evaluation purposes.

4.2.1 Choice of branching heuristics

We first compare different branching heuristics on both complete instances and instances that have been preprocessed using $k = 15\%$. The heuristics observed are maximum regret (a); maxSD* on 2-matching (b), on 2-matching and 1-tree (c), and on 2-matching, 1-tree and n-path (d); arithmetic mean of SDs on 2-matching and 1-tree (e) and of 2-matching, 1-tree and n-path (f). Table 4.7 compares these heuristics on the complete graphs, whereas Table 4.6 compares them on the pre-processed instances. Both tables indicate, for each instance and heuristic, the best found solution value within a 30 minute time limit, the time it took to find that solution and the number of fail nodes in the search tree.

Comparing both tables, we can see that every heuristic performed much worse on the complete instances, proving that removing edges from the graph in this way is beneficial. This is especially true with the maximum regret heuristic which generally finds poor solutions on

complete instances. For the larger instances, the best found solutions on complete instances are over twice the value of the instances' optimal solutions.

Secondly, our proposed heuristics generally perform better than the maximum regret heuristic. They consistently find better solutions when the optimal solution cannot be reached and find the optimal solution faster when it can be found. However, heuristics involving the n-path relaxation behave poorly, as they are too computationally expensive.

Table 4.6 Performance of IBM ILOG CP 1.6 on 9 small sparsified instances (30 min timeout).

	opt		best	time(s)	fails		opt		best	time (s)	fails		opt		best	time(s)	fails
gr21	2707	a	2707	0.11	173	gr24	1272	a	1272	0.26	359	fri26	937	a	937	0.43	475
		b	2707	0.04	48			b	1272	0.03	25			b	937	1.71	2163
		c	2707	0.14	27			c	1272	0.07	2			c	937	16.54	3156
		d	2707	34.90	60			d	1272	76.67	182			d	937	485.96	1367
		e	2707	0.15	31			e	1272	0.31	47			e	937	12.37	2393
		f	2707	35.23	70			f	1272	30.39	51			f	937	1684.26	3254
bays29	2020	a	2020	125.36	118869	dantzig42	699	a	699	0.26	0	swiss42	1273	a	1464	1537.48	1349413
		b	2020	7.57	9341			b	722	114.73	40888			b	1298	127.31	42503
		c	2020	8.37	1179			c	722	1147.09	73431			c	1397	24.85	1316
		e	2020	9.03	1287			e	718	72.78	3505			e	1410	25.30	1266
gr48	5046	a	5898	1097.14	423271	hk48	11461	a	14734	1486.84	970394	berlin52	7542	a	10434	1162.14	702850
		b	5055	548.57	195503			b	12466	591.61	277333			b	8224	104.56	123179
		c	5174	860.07	31897			c	12039	765.05	29543			c	8193	621.60	18781
		e	-	-	-			e	12032	1379.09	54948			e	8016	558.90	15548

Table 4.7 Performance of IBM ILOG CP 1.6 on 9 small complete instances (30 min timeout).

	opt		best	time(s)	fails		opt		best	time (s)	fails		opt		best	time(s)	fails
gr21	2707	a	2707	68.01	36326	gr24	1272	a	1861	1679.27	865198	fri26	937	a	937	1003.97	336475
		b	2707	4.85	1648			b	1272	0.20	199			b	937	736.95	161226
		c	2707	11.98	1640			c	1272	1.04	200			c	954	194.34	21214
		e	2707	13.25	1829			e	1272	0.96	171			e	937	1615.59	161633
bays29	2020	a	3743	1774.69	1312433	dantzig42	699	a	699	1.03	0	swiss42	1273	a	2228	1538.42	1030452
		b	2093	1610.44	527637			b	783	71.97	20390			b	1368	3.79	817
		c	2123	1556.13	153615			c	783	400.75	21259			c	1368	16.44	711
		e	2020	64.36	7342			e	769	1002.58	48854			e	1381	100.00	3787
gr48	5046	a	14247	850.31	150656	hk48	11461	a	31822	1547.52	973401	berlin52	7542	a	18904	395.59	342182
		b	5075	1152.88	126075			b	12061	245.78	55344			b	8446	408.00	909200
		c	5169	1235.57	32022			c	12078	17.52	491			c	8449	1585.79	57455
		e	5278	12.55	285			e	12078	434.98	18616			e	8122	1652.49	49203

4.2.2 Effect of parameters

lowCostGap

Table 4.8 reports the results of ILOG CP on three different branching heuristics (costMaxSD on 2-matching (b), costMaxSD on 2-matching and 1-tree (c), and arithmetic mean of SDs on 2-matching and 1-tree (e)) and for different values (0.01, 0.05, 0.1, 0.2, 0.5) of the lowCostGap parameter for the weighted allDifferent constraint (section 3.1.1).

Table 4.8 Performance of IBM ILOG CP 1.6 on 9 small sparsified instances with varying lowCostGap values (30 min timeout) (the values in the top row are the lowCostGap values).

			0.01			0.05			0.1			0.2			0.5		
instance	opt		best	time(s)	fails	best	time(s)	fails	best	time (s)	fails	best	time(s)	fails	best	time(s)	fails
gr21	2707	b	2707	0.03	52	2707	0.04	48	2707	0.04	48	2707	0.03	43	2707	0.02	27
		c	2707	0.05	38	2707	0.18	38	2707	0.14	27	2707	0.10	19	2707	0.10	20
		e	2707	0.14	23	2707	0.17	45	2707	0.15	31	2707	0.10	15	2707	0.10	15
gr24	1272	b	1272	0.01	4	1272	0.02	14	1272	0.03	25	1272	0.03	32	1272	0.03	32
		c	1272	0.07	4	1272	0.07	0	1272	0.07	2	1272	0.09	4	1272	0.09	4
		e	1272	0.08	7	1272	0.13	15	1272	0.31	47	1272	0.06	2	1272	0.08	3
fri26	937	b	937	5.62	14920	937	0.06	25	937	1.71	2163	937	0.95	2510	937	1.25	2245
		c	937	45.23	12371	937	0.22	21	937	16.54	3156	937	9.88	2743	937	8.15	2498
		e	937	11.08	3321	-	-	-	937	12.37	2393	937	8.19	1563	937	6.14	1853
bays29	2020	b	2020	38.41	84781	2020	11.70	15896	2020	7.57	9341	2020	6.73	15052	2020	9.88	15394
		c	2020	325.79	68458	2020	74.50	14653	2020	8.37	1179	2020	11.11	2463	2020	11.05	2472
		e	2020	69.32	14895	2020	52.18	10652	2020	9.03	1287	2020	18.10	4085	2020	18.13	4095
dantzig42	699	b	722	1217.60	2153592	717	1624.40	1912419	722	114.73	40888	758	837.78	935309	812	1421.77	1217315
		c	727	525.24	60972	725	1325.24	97320	722	1147.09	73431	734	86.74	6435	735	780.91	63355
		e	-	-	-	-	-	-	718	72.78	3505	-	-	-	710	1690.36	132285
swiss42	1273	b	1281	181.08	327922	1304	1233.83	567634	1298	127.31	42503	1344	1500.15	726983	1386	1079.35	385777
		c	1294	1488.11	140039	1342	99.29	5779	1397	24.85	1316	1347	1051.76	92650	1347	1020.59	89211
		e	1286	717.17	67975	1342	81.99	2907	1410	25.30	1266	1327	60.34	3007	1354	76.74	5939
gr48	5046	b	5123	1121.25	1539460	5083	546.51	334262	5055	548.57	195503	5297	2.88	2731	5416	391.64	168015
		c	5262	66.13	4460	5083	680.09	24947	5174	860.07	31897	5395	78.91	4907	5336	626.81	36237
		e	5270	1633.62	98785	5201	1670.83	84736	-	-	-	5340	1146.98	42093	5338	638.91	35868
hk48	11461	b	12221	306.46	1138587	12437	747.98	282211	12466	591.61	277333	11952	1261.66	503770	-	-	-
		c	-	-	-	12522	108.30	4149	12039	765.05	29543	11805	1383.38	85816	11788	1326.51	77572
		e	11894	1380.76	94225	11828	694.46	27448	12032	1379.09	54948	12718	224.89	8689	12718	140.79	8722
berlin52	7542	b	-	-	-	8056	1570.87	718325	8224	104.56	123179	8404	41.04	32984	9575	171.20	108173
		c	-	-	-	8107	935.44	24836	8193	621.60	18781	8488	484.11	24969	8665	1022.22	49513
		e	7844	1.43	0	8432	1670.83	84736	8016	558.90	15548	8053	554.52	15896	7981	1636.31	75122

First of all, the larger values of `lowCostGap` perform better on the smallest instances (those that can be solved to optimality by the CP model), as they generally require fewer fails, hence reducing the solve time. Then, as the size of the instances increases and the time limit prevents the solver from reaching optimality, the smaller `lowCostGap` values consistently find better solutions.

However, looking at the largest instance in the table (*berlin52*), two of the three heuristics could not find any solution within 30 minutes when the `lowCostGap` was 0.01, whereas a solution was found for every heuristic for every other value of `lowCostGap`. This trend has been confirmed on a few larger instances that do not appear in the table. Why does this happen? When `lowCostGap` is very small, the heuristic is only concerned with solutions very close to optimal, for the 2-matching, not the TSP itself. This could explain why on smaller instances, it tends to find the solutions with the lowest values. That being said, it also con-

siders much fewer sub-optimal solutions than the larger `lowCostGap` values. Therefore, if it fails to find a good solution immediately on harder instances, backtracking could prove to be more difficult, since it could be locked in a series of decisions that would be good for the 2-matching relaxation but far away from a feasible solution of the TSP. On the other hand, with large values of `lowCostGap`, our branching decisions could lead to weaker solutions of the 2-matching, but they are more likely to lead to feasible TSP solutions.

Finally, the results suggest that, for these sizes of instances, the value 0.1 may be the best. Among these 5 values, it is the most consistent in terms of solution quality and time. In a sense, it provides a good compromise between focusing on the best solutions, like the smaller values, and making more informed decisions, like the larger values. It is possible that this value's performance is a result of it also being used for pre-processing, although we see no theoretical reason behind this. This value was also the most interesting for pre-processing because large values of `lowCostGap` could increase the time of computing SDs non-negligibly and small values of `lowCostGap` could lead to many edges having SDs of 0, which we need to avoid.

k% of edges

Now, we evaluate how the value of k, i.e., the percentage of edges we take from each relaxation when pre-processing, affects the performance of the CP solver on a few small instances, small enough to be solved to optimality when sparsified.

Table 4.9 reports, for 3 instances and values of k from 15 to 75, the best solution found within 30 minutes, the time to find that solution, the number of fails before finding it and the time required to prove optimality, if the optimal solution was found. These results were obtained by running the CP solver with the branching heuristic (b) (see section 4.2.1).

First of all, we note that as k increases, the total run-time, which includes the optimality proof, consistently increases, aside from one outlier ($k = 35$ for *gr24*), which is to be expected since the size of the search tree depends on the number of edges in the graph. Additionally, on the larger instance *bays29*, which only reaches optimality when k equals 15 or 20, the best solution found progressively gets worse, with some exceptions.

We do also note however that this is not necessarily the case for the time and fails necessary to simply find the optimal solution. For two of these instances, when k is between 45 and 60, the optimal is found in fewer fails than with the smallest value of k, 15. Again, this is not entirely surprising, considering that branching in CP is heuristic. It is not unlikely that

Table 4.9 Performance of ILOG CP solver (30 min timeout) depending on the k% of edges kept in pre-processing

	gr24				fri26				bays29			
k	best found	time (s)	fails	proof time (s)	best found	time (s)	fails	proof time (s)	best found	time (s)	fails	proof time (s)
15	1272	0.03	25	0.12	937	1.19	2163	5.39	2020	7.53	9341	59.90
20	1272	1.05	1927	11.15	937	55.93	46367	152.40	2020	3.51	4023	1800
25	1272	3.45	4319	24.42	937	5.62	4603	164.61	2028	70.01	46813	1800
30	1272	15.96	14962	45.13	937	25.23	7384	474.86	2028	144.20	72478	1800
35	1272	34.07	19479	93.53	937	32.12	8166	806.31	2036	302.19	108174	1800
40	1272	0.08	5	62.39	937	169.74	44734	818.71	2094	196.84	100630	1800
45	1272	0.09	0	66.16	937	1.84	1644	745.99	2086	1092.80	379392	1800
50	1272	0.10	0	80.40	937	1.12	665	990.18	2086	72.38	47038	1800
55	1272	0.09	0	82.22	937	1.47	505	1630.10	2086	7.49	5496	1800
60	1272	0.10	0	84.60	937	1.62	478	1800	2086	7.52	5496	1800
65	1272	0.17	199	89.16	937	16.53	10326	1800	2093	894.50	347034	1800
70	1272	0.18	199	89.41	937	736.29	160978	1800	2093	155.66	58747	1800
75	1272	0.18	199	91.03	937	761.16	165748	1800	2110	1689.67	576498	1800

adding certain edges opens up a branch of the search tree leading to an optimal solution that was previously inaccessible. Another interesting detail is that, despite requiring fewer fails than when $k = 15$, the solution time in most of these cases is still greater. We can conclude from this that the greater k is, the longer each branching decision takes, which is supported by the rest of the data in the table. Again, theory supports these results, since fewer edges means smaller domains for the decision variables and fewer solution densities to compute.

4.3 CONCORDE

In this section, we report on the performance of CONCORDE, the state-of-the-art IP solver, to see if removing edges from input graphs with our algorithm affects it.

4.3.1 Smaller instances

To further evaluate the effect of sparsifying TSP instances as indicated in Section 3.2, we pre-processed 16 TSPLIB instances containing between 150 and 400 nodes, keeping only 1% of edges per relaxation. These pre-processed instances as well as the original instances were then solved on CONCORDE, yielding the results shown in Table 4.10. The reported characteristics are the number of branch-and-bound nodes, the overall computing time, the time required for branching, and the number of generated cuts. Since the performance of CONCORDE varies from run to run due to a random seed, the numbers are all averages over 10 executions.

Of course, these instances are not challenging for CONCORDE, since the largest average tree size for the original instances is only 11.2 and 9 of the 16 instances do not require branching at all. There is little room for improvement in that regard, but nevertheless there is an

Table 4.10 Average performance (10 runs) of CONCORDE to solve to optimality symmetric instances of size 150 to 400 from the TSPLIB, with and without preprocessing.

	original graph				graph with discarded edges			
instance	# bbnodes	total time (s)	branching (s)	# cuts	# bbnodes	total time (s)	branching (s)	# cuts
ch150	1.00	0.49	0	111.50	2.40	0.35	0.21	88.20
kroA150	1.00	0.88	0	164.30	1.00	0.19	0	32.90
kroB150	1.20	0.84	0.01	171.30	1.20	0.42	0.02	58.80
si175	2.80	3.42	0.24	294.20	1.00	0.25	0	53.80
brg180	1.00	0.71	0	5.00	1.00	0.09	0	1.80
rat195	5.60	5.49	3.14	314.80	4.60	4.07	2.67	325.60
d198	3.20	2.29	0.31	193.00	1.60	1.12	0.28	95.10
kroA200	1.00	0.77	0	250.10	1.40	0.39	0.10	103.70
kroB200	1.00	0.44	0	136.60	1.20	0.21	0.02	33.10
ts225	1.50	8.01	0.23	875.00	1.00	0.44	0	197.90
pr226	1.00	0.51	0	101.90	1.00	0.24	0	42.10
pr264	1.00	0.44	0	49.70	1.00	0.28	0	17.90
a280	1.00	1.18	0	107.00	3.00	1.04	0.23	111.40
pr299	1.80	3.24	0.09	446.90	2.20	2.11	0.84	387.40
lin318	1.00	1.77	0	237.00	2.40	3.53	0.62	306.90
rd400	11.20	18.20	12.30	754.10	11.00	17.97	14.62	814.20
Average	2.21	3.04	1.02	263.28	2.31	2.04	1.23	166.93

overall improvement in the solving process, as we can see a reduction in the overall time and, especially, in the number of cutting planes needed to reach optimality.

4.3.2 Larger instances

Larger instances of size 500 to 1000 can take several seconds to solve in CONCORDE, leaving more room for improvement compared to the smaller instances tested in the previous section. However, pre-processing instances still does not seem to have much of an effect on the solve time of CONCORDE.

Figures 4.1 and 4.2 show the evolution of the best upper and lower bounds over time for the complete and pre-processed versions of instances *d493* and *rat575*. We notice that CONCORDE quickly finds better upper bounds on the pre-processed instances, but, on the other hand, the lower bounds are consistently worse for the pre-processed instances, which unfortunately can lead to a slower convergence of the bounds and longer runtime. The behavior of the lower bound seems consistent with the results for the smaller instances, that showed that, on average, pre-processed instances took longer for branching.

It appears our pre-processing method is not competitive with CONCORDE for instances of any size, which is not entirely surprising considering how well optimized this solver is for the TSP. Even if there is a slight improvement on some instances, it does not compensate the

required time for pre-processing.

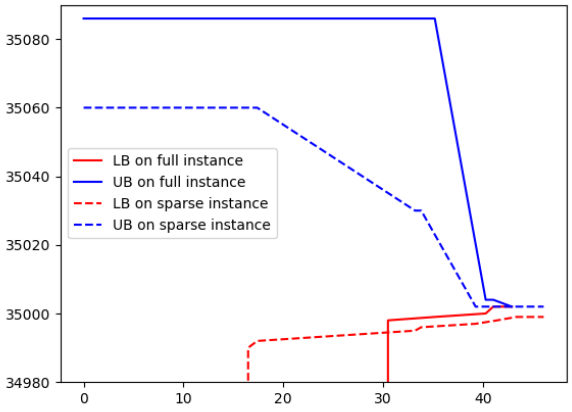

Figure 4.1 Evolution of bounds over time(s) on complete and sparse versions of *d493* in CONCORDE

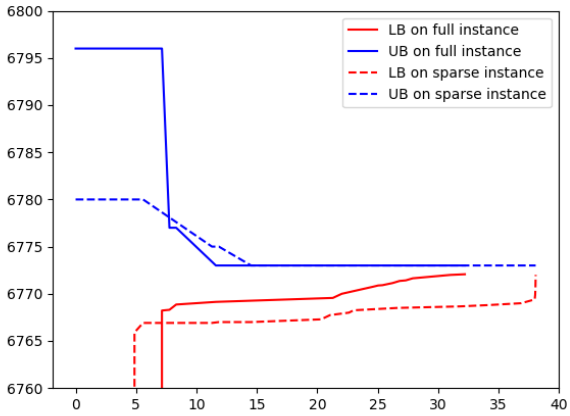

Figure 4.2 Evolution of bounds over time (s) on complete and sparse versions of *rat575* in CONCORDE

Hougardy and Schroeder [32] present their own method of removing edges from TSP instances, which has been shown to yield improvement on CONCORDE for extremely large instances. Unfortunately, our pre-processing method does not scale well for such large instances, especially the computation of SDs for the 1-tree.

4.3.3 Effect of k%

Table 4.11 reports the performance of CONCORDE on 2 instances, *rd400* and *d493*, pre-processed with varying values of parameter k. The results reported are, again, averages over 10 executions.

We can see, from this table, that the solve time, the number of branch-and-bound nodes and the number of cuts are all rather consistent across all values of k, with only some negligible variance. These results seem to confirm our previous observations, namely that our pre-processing algorithm has little bearing on the performance of CONCORDE, at least for the range of instance size that was observed.

Table 4.11 Performance of CONCORDE depending on the k% of edges kept in pre-processing

	rd400			d493		
k	time (s)	# bb nodes	# cuts	time (s)	# bb nodes	# cuts
25	18.10	9.40	763.60	60.34	7.40	1927.50
30	18.10	8.50	736.40	55.63	8.60	1991.80
35	19.07	9.80	723.90	59.23	7.80	1840.40
40	21.80	11.80	806.00	54.31	8.00	1856.00
45	20.01	10.00	753.90	60.41	7.40	1854.60
50	19.50	10.00	763.50	55.53	7.80	1867.60
55	18.20	9.20	756.10	55.62	7.40	1778.00
60	18.05	10.80	710.20	57.22	8.00	1805.20
65	18.61	10.20	706.80	64.85	8.40	1969.50
70	18.23	9.80	759.00	59.76	7.60	2001.20
75	17.69	9.20	750.00	58.36	6.80	1890.00

4.4 Mixed-Integer Model

In this section, we evaluate the performance of two MIP models for the TSP, implemented in Gurobi. These models are implementations of the two formulations presented in Section 2.1.1.

4.4.1 DFJ formulation

The first Gurobi model is an implementation of the DFJ formulation of the TSP. The exponential number of sub-tour elimination constraints are implemented as lazy constraints, meaning they are added progressively as they are violated.

Table 4.12 Performance of the DFJ model on Gurobi to solve symmetric instances of size 150 to 400, with a time limit of 7200 seconds

	original graph			graph with 25% edges			
instance	time (s)	# nodes	# cutting planes	pre-processing (s)	time (s)	# nodes	# cutting planes
ch150	21.87	2541	77	1.94	7.47	2539	72
kroA150	53.91	8556	116	2.04	9.94	6844	82
kroB150	60.56	16548	120	2.05	21.37	25718	216
rat195	168.54	25595	236	2.85	27.49	10049	142
d198	344.40	57912	246	3.41	51.90	51827	303
kroA200	345.48	57759	0	3.03	122.84	162874	399
kroB200	86.10	8799	109	3.39	18.70	9104	165
ts225	x	x	x	4.83	x	x	x
pr226	x	x	x	4.14	319.97	287358	582
pr264	243.57	37947	182	9.81	386.89	87383	329
a280	20.75	1109	61	6.07	6.72	776	52
pr299	312.51	32056	102	6.74	417.49	218922	332
lin318	605.64	71831	554	8.51	302.45	71910	558
rd400	428.31	20063	91	12.47	462.53	156359	459

Table 4.12 compares the performance of this DFJ model on the original complete instance and on a graph that we pre-processed keeping 25% of edges per vertex. Unsurprisingly, it

performs much worse than CONCORDE on these instances, though it greatly outshines a CP model. Overall, we see considerable improvement in most instances, when pre-processing the graph. It is interesting to note that, though the pre-processed instances sometimes requires more branch-and-bound nodes to reach optimality, there is still a time gain since each simplex iteration is much quicker.

Although some instances reached optimality quicker when they were complete, Figures 4.3 and 4.4 show that the sparse instances find good bounds much quicker, which can be useful in cases where we are constrained to strict time limits and have to settle for approximate solutions. We can see that it is especially beneficial for the upper bound, corresponding to the best found solution, as the pre-processed instances find a first upper bound several hundreds of seconds before the complete instance.

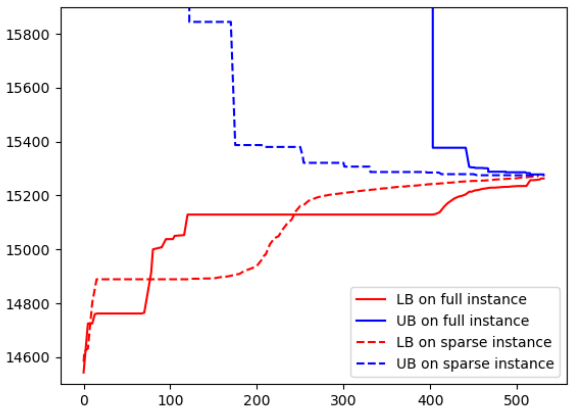

Figure 4.3 Evolution of bounds over time (s) on complete and sparse versions of *rd400* for the DFJ model

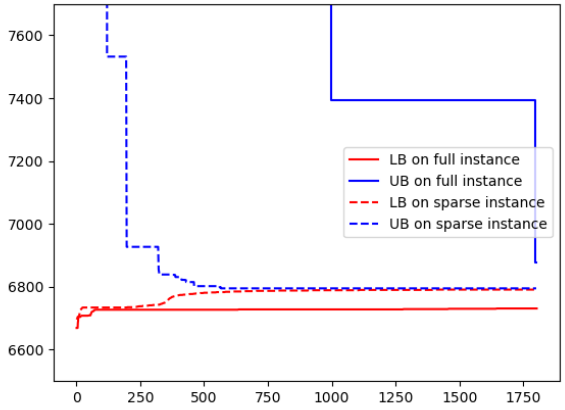

Figure 4.4 Evolution of bounds over time (s) on complete and sparse versions of *rat575* for the DFJ model

4.4.2 Effect of k%

In this section, we analyze how the value of parameter k affects the performance of the DFJ model in Gurobi. Table 4.13 reports, for three instances, the time, the number of branch-and-bound nodes and number of cuts required to reach optimality.

Looking at this table, we see that the behavior of the model can vary greatly from instance

to instance and, as k increases, there is no consistent increase in the solve time, the number of nodes or the number of cuts. For *kroA200*, the average number of nodes and cuts decrease when $k > 50$, but the opposite is true for *a280*. On the other hand, the number of nodes and cuts is pretty consistent for all values of k, for *lin318*.

However, we do see a trend in the correlation between the time and the number of nodes. It can be noted that an increase in k leads to an increase in time per branch-and-bound node and this trend can be observed in each of these instances. For *kroA200*, solving takes just under 100 seconds when $k = 25$, though it explores over 100 000 nodes, whereas when $k > 50$, it consistently takes over 100 seconds despite exploring much fewer nodes. For *a280*, both $k = 30$ and $k = 70$ yield a solve time of around 12 seconds, despite the case of $k = 30$ having over twice as many branch-and-bound nodes. For *lin318*, the $k = 75$ case is the second slowest case in terms of time, despite having the second lowest number of branch-and-bound nodes.

We can conclude from these results that removing edges aggressively will, in a majority of cases, be the most beneficial method, since it is difficult to predict the size of the branch-and-bound search tree and having the fewest possible edges leads to a better ratio of time over tree size.

Table 4.13 Performance of DFJ model on Gurobi on three instances depending on the k% of edges kept in pre-processing

	kroA200			a280			lin318		
k	time (s)	# bb nodes	# cuts	time (s)	# bb nodes	# cuts	time (s)	# bb nodes	# cuts
25	95.60	110324	251	8.51	776	54	302.45	71910	558
30	128.78	115726	282	11.50	1799	43	337.52	66306	494
35	199.13	193125	281	8.56	883	56	405.70	74097	561
40	118.54	100664	37	8.16	884	60	409.07	83502	566
45	231.04	221424	226	7.84	745	0	389.46	73474	497
50	100.44	56750	50	18.08	1048	64	421.52	79142	575
55	165.30	84032	0	19.58	2461	62	425.61	68067	462
60	191.26	82080	60	58.73	8088	149	526.84	86069	548
65	145.11	68862	54	34.08	4461	102	452.48	67818	438
70	135.81	64828	32	12.10	760	39	351.76	44350	351
75	166.92	66315	40	110.94	14849	223	490.07	57305	425

4.4.3 MTZ formulation

The second model implemented in Gurobi is a direct implementation of the MTZ formulation, with $n - 1$ extra variables and a quadratic number of sub-tour elimination constraints. As mentioned in Section 2.1.1, this formulation, unlike DFJ, is for the ATSP. It can also be used

to solve the STSP instances we worked with, but we have to slightly adapt how we apply the pre-processing to the model. The solution is rather intuitive: removing edge (i, j) from a STSP instance is equivalent to removing arcs $i \to j$ and $j \to i$ from its equivalent ATSP instance.

Table 4.14 Performance of the MTZ model on Gurobi to solve instances of size 48 to 100, with a time limit of 7200 seconds

	original graph			graph with 25% edges			
instance	time (s)	# nodes	# cuts	pre-processing (s)	time (s)	# nodes	# cuts
gr48	37.77	14033	151	0.59	36.47	24871	342
hk48	7.78	4956	126	0.59	12.95	11676	102
berlin52	1.52	397	113	0.73	1.45	604	57
brazil58	276.05	93543	413	0.67	105.27	83663	527
st70	672.10	155387	455	0.99	374.13	165210	568
pr76	1567.91	181319	952	0.93	447.00	154772	759
kroA100	4010.80	468474	1017	1.52	583.88	138651	489
kroB100	x	x	x	1.26	x	x	x
kroC100	x	x	x	1.37	x	x	x
kroD100	x	x	x	1.53	x	x	x
kroE100	1527.41	151098	660	1.28	676.49	92565	1176

The MTZ implementation in Gurobi is a lot less efficient than the DFJ model evaluated in the previous section. As we can see from Table 4.14, a majority of instances of size 100 could not be solved to optimality within 2 hours.

On the smallest instances, pre-processing the instances did not yield much improvement in terms of time, but we can see again that, with fewer edges, less time is required per branch-and-bound nodes visited. On the other hand, pre-processing was very beneficial for the larger instances in this set. The solving time for these instances was reduced to at least 55% and at best 15% (for instance *kroA100*) of the original solve time.

Figures 4.5 and 4.6 serve to illustrate a point made in Section 4.4.1, which is that in case optimality cannot be reached quickly, pre-processing graphs helps to find much better bounds on the optimum in a limited amount of time. On the instance *kroD100*, after 2 hours of runtime, the complete graph has a gap of approximately 14% between the upper and lower bound, whereas the sparse graph reduces this gap to under 2%.

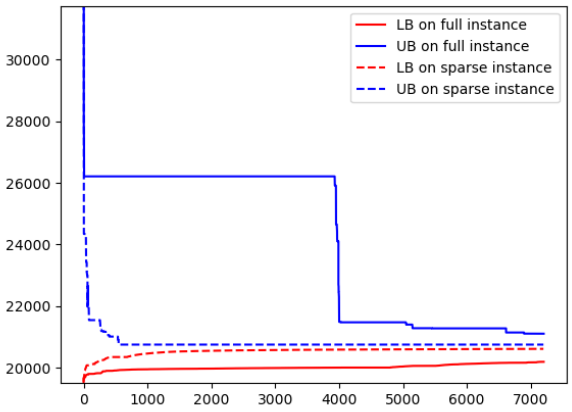

Figure 4.5 Evolution of bounds over time (s) on complete and sparse versions of *kroC100* for the MTZ model

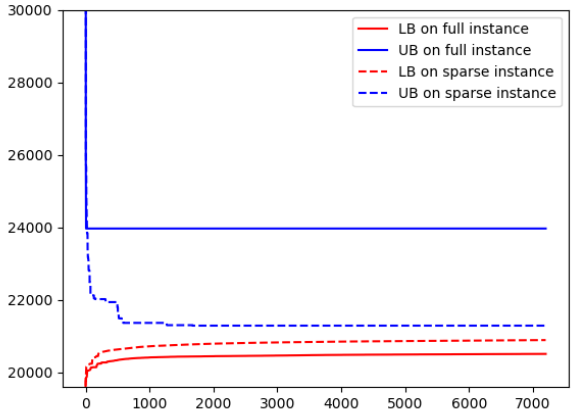

Figure 4.6 Evolution of bounds over time (s) on complete and sparse versions of *kroD100* for the MTZ model

CHAPTER 5 CONCLUSION AND RECOMMENDATIONS

5.1 Summary of Works

In this thesis we presented an efficient method removing undesirable edges from the input graph of TSP instances. We explained how to combine the best edges, in terms of solution density, from two different relaxations of the TSP to make this method as robust as possible, while removing edges in a relatively aggressive manner.

We also propose faster alternatives to rank the solution densities of the `minWeightAllDifferent` and `minWeightSpanningTree` constraints.

5.2 Limitations

The method presented in this thesis does have certain limitations and weaknesses, which we will discuss now.

Firstly, there is no definitive method for choosing the number of edges to keep in the graph, while guaranteeing that no edge of the optimal tour is removed. In other words, there is no way of proving optimality for a given parameter $k\%$. As a result of this limitation, it is necessary to find this percentage empirically for every instance and although a large majority of instances tested have proven to stay optimal with 25% of edges, some instances required more, making it difficult to obtain results that are uniform while being as efficient as possible.

Secondly, the computation time of our method scales somewhat poorly when the size of instances approaches 2000. This is due to the algorithm itself and the computation of SDs for the `SpanningTree` constraint, which on top of scaling poorly can fail due to overflow when instances become so large. Improving these 2 algorithms' performance would be necessary to see if this approach of removing edges can help CONCORDE on very large instances that require a lot of time to be solved.

Another limitation of our method in this context is that we could only use the SDs from two constraints, `AllDifferent` and `SpanningTree`, in our pre-processing, since the `Regular` constraint we implemented was not exploitable and no other prominent relaxation of the TSP came to mind. Having more relaxations to choose from could have given us access to more information regarding which edges can be safely removed from the input graph. There

is a caveat to using more relaxations though, namely the computation time of the SDs for the extra relaxations.

One major weakness of our method is that it does not help the state-of-the-art solver for the pure TSP, CONCORDE. This weakness appears two-fold. First of all, the time required to pre-process large instances (size of multiple hundreds to the thousands) is much larger than the time required for CONCORDE to solve the complete instance. Second of all, disregarding the time required for pre-processing, solving the sparse instance on CONCORDE hardly ever improves upon the initial solve time.

The reason it is difficult to improve on CONCORDE with our pre-processing reduction is that, on large symmetric Euclidian instances, CONCORDE does not receive, on input, the full cost matrix but only the Euclidian coordinates. The edges are priced in as they are needed, i.e., when they are "cheap". In other words, it is likely that CONCORDE never needs to price the edges that we would remove in pre-processing and that would explain the lack of benefit we observe. Another partial confirmation of this interpretation is that, in [32], the authors report an improvement through edge elimination especially on instances in which the LP relaxation is degenerate [43], i.e., instances in which a lot of edges need to be priced in.

5.3 Future Research

There is plenty of room for improvement in this field and future research is required to expand this method's range of application.

Future works could potentially be done to compute some sort of "proof of optimality" for the pre-processing, to deal with the first limitation mentioned in Section 5.2. Since the method involves solution densities, it is a heuristic method and, thus, it would be impossible to guarantee 100% that the optimal solution is preserved given the percentage of edges kept in pre-processing. However, it may be possible to develop a probabilistic value, indicating the likelihood of optimality being preserved after pre-processing.

Furthermore, our approach could be combined with other methods, to make it more robust or exact. Discrepancy-based search [44] could be considered, for example, to make the approach exact by considering an increasingly large percentage of edges, until reaching 100% or until we can prove that the optimal solution was found.

In my opinion, the best way to expand upon this research in the future is to generalize the

method to other types of routing problems. There are multiple motivations for this.

For starters, I believe that our pre-processing method can be adapted somewhat intuitively to many other routing problems. After all, these problems certainly share some relaxations and possess similar structures.

Secondly, CONCORDE can only solve pure TSPs and therefore TSP variants and other routing problems, whose exact solvers may not be as efficient as CONCORDE is for the TSP, could further benefit from the pre-processing.

Additionally, an interesting aspect of adapting this method to more complex routing problems is that their additional constraints could yield new relaxations and new constraints to compute SDs for. For instance, the n-path may be more exploitable for the Vehicle Routing Problem with Time Windows or the Capacitated Vehicle Routing Problem. These new relaxations could bring more information on the problem's structure, which opens up more options when selecting edges for pre-processing and could allow for a more robust method. We should address a concern mentioned in Section 5.2, regarding the additional computation time of SDs when working with more relaxations. Given more relaxations, it is possible for the pre-processing time not to increase, if we simply choose not to use every relaxation available. We could study each relaxation or constraint, and keep only those that provide the most useful information for pre-processing or, if we are very worried about computation time, keep those with the fastest method of computing SDs.

REFERENCES

[1] G. Pesant, M. Gendreau, J.-Y. Potvin, and J.-M. Rousseau, "An exact constraint logic programming algorithm for the traveling salesman problem with time windows," *Transportation Science*, vol. 32, no. 1, pp. 12–29, Dec 1996. [Online]. Available: http://dx.doi.org/10.1287/trsc.32.1.12

[2] G. Pesant, C.-G. Quimper, and A. Zanarini, "Counting-based search: Branching heuristics for constraint satisfaction problems," *J. Artif. Int. Res.*, vol. 43, no. 1, pp. 173–210, Jan. 2012. [Online]. Available: http://www.polymtl.ca/labo-quosseca/publications

[3] R. Gomory, "Outline of an algorithm for integer solutions to linear programs," *Bulletin of the American Mathematical Society*, vol. 64, no. 5, pp. 275–278, Sep. 1958. [Online]. Available: http://dx.doi.org/10.1090/S0002-9904-1958-10224-4

[4] H. Marchand and L. Wolsey, "Aggregation and mixed integer rounding to solve MIPs," *Operations Research*, vol. 49, no. 3, pp. 363–371, Jun. 2001. [Online]. Available: https://dx.doi.org/10.1287/opre.49.3.363.11211

[5] Z. Gu, G. Nemhauser, and M. W. Savelsbergh, "Lifted cover inequalities for 0-1 integer programs: Computation," *INFORMS Journal on Computing*, vol. 11, no. 1, pp. 117–123, Feb. 1999. [Online]. Available: https://dx.doi.org/10.1287/ijoc.11.1.117

[6] M. Padberg and G. Rinaldi, "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems," *SIAM Review*, vol. 33, no. 1, pp. 60–100, 1991. [Online]. Available: https://dx.doi.org/10.1137/1033004

[7] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, Jul. 1960. [Online]. Available: https://www.jstor.org/stable/1910129

[8] S. J. Russel and P. Norvig, *Artificial intelligence: a modern approach*, 3rd ed. Upper Saddle River, New Jersey: Prentice Hall, 2009, pp. 122–125.

[9] F. Glover, "Tabu search - part 1," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190–206, Aug. 1989. [Online]. Available: https://dx.doi.org/10.1287/ijoc.1.3.190

[10] F. Glover, "Tabu search - part 2," *ORSA Journal on Computing*, vol. 2, no. 1, pp. 4–32, Feb. 1990. [Online]. Available: https://dx.doi.org/10.1287/ijoc.2.1.4

[11] S. Kirkpatrick, C.D. Gelatt Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983. [Online]. Available: http://dx.doi.org/10.1126/science.220.4598.671

[12] M. Dorigo and C. Blum, "Ant colony optimization theory: a survey," *Theoretical computer science*, vol. 344, no. 2-3, pp. 243–278, 2005. [Online]. Available: https://dx.doi.org/10.1016/j.tcs.2005.05.020

[13] R. Karg and G. Thompson, "A Heuristic Approach to Solving Travelling Salesman Problems," *Management Science*, vol. 10, no. 2, pp. 225–248, Jan. 1964. [Online]. Available: https://dx.doi.org/10.1287/mnsc.10.2.225

[14] D. Fulkerson, "Flow networks and combinatorial operations research," *The American Mathematical Monthly*, vol. 73, no. 2, pp. 115–138, Feb. 1966. [Online]. Available: https://www.jstor.org/stable/2313544

[15] G. Dantzig, R. Fulkerson, and S. Johnson, "Solution of a large-scale traveling-salesman problem," *Journal of the Operations Research Society of America*, vol. 2, no. 4, pp. 393–410, Nov. 1954. [Online]. Available: https://dx.doi.org/10.1287/opre.2.4.393

[16] M. Bellmore and G. L. Nemhauser, "The traveling salesman problem: A survey," in *Mathematical Models in Marketing. Lecture Notes in Economics and Mathematical Systems (Operations Research).* Springer,Berlin,Heidelberg, 1976, vol. 132.

[17] C. E. Miller, A. W. Tucker, and R. A. Zemlin, "Integer programming formulation of traveling salesman problems," *Journal of the ACM*, vol. 7, no. 4, pp. 326–329, Oct. 1960. [Online]. Available: https://dl.acm.org/citation.cfm?id=321046

[18] B. L. Golden, "Vehicle routing problems : Formulations and heuristic solution techniques," Aug. 1975.

[19] G. A. Croes, "A method for solving traveling-salesman problems," *Operations Research*, vol. 6, no. 6, pp. 791–812, Nov. 1958. [Online]. Available: https://doi.org/10.1287/opre.6.6.791

[20] S. Lin, "Computer solutions of the traveling salesman problem," *Bell System Technical Journal*, vol. 44, no. 10, pp. 2245–2269, Dec. 1965. [Online]. Available: https://doi.org/10.1002/j.1538-7305.1965.tb04146.x

[21] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations Research*, vol. 21, no. 2, pp. 498–516, mar-apr 1973. [Online]. Available: http://www.jstor.org/stable/169020

[22] K. Helsgaun, "An effective implementation of the lin-kernighan traveling salesman heuristic," *European Journal of Operations Research*, vol. 126, no. 1, pp. 106–130, Oct. 2000. [Online]. Available: http://dx.doi.org/10.1016/S0377-2217(99)00284-2

[23] B. L. Golden, "A statistical approach to the tsp," *Networks*, vol. 7, no. 3, pp. 209–225, 1977. [Online]. Available: https://dx.doi.org/10.1002/net.3230070303

[24] W. Cook and P. Seymour, "Tour merging via branch-decomposition," *INFORMS Journal on Computing*, vol. 15, no. 2, Aug. 2003. [Online]. Available: https://dx.doi.org/10.1287/ijoc.15.3.233.16078

[25] H. Tamaki, "Alternating cycles contribution: a strategy of tour-merging for the traveling salesman problem," in *Research report MPI 2003-1-007*. Saarbrucken, Germany: Max-Planck-Institut für Informatik, 2003. [Online]. Available: http://hdl.handle.net/11858/00-001M-0000-0014-6B66-B

[26] H. D. Nguyen, I. Yoshihara, K. Yamamori, and M. Yasunaga, "Implementation of an effective hybrid GA for large-scale traveling salesman problems," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 37, no. 1, pp. 92–99, Feb. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSMCB.2006.880136

[27] *Concorde TSP Solver*, http://www.math.uwaterloo.ca/tsp/concorde/index.html.

[28] W. J. Cook, D. L. Applegate, R. E. Bixby, and V. Chvátal, *The Traveling Salesman Problem: A computational study.* Princeton University Press, 2006. [Online]. Available: https://www.jstor.org/stable/j.ctt7s8xg

[29] D. Applegate, W. J. Cook, and A. Rohe, "Chained lin-kernighan for large traveling salesman problems," *INFORMS Journal on Computing*, vol. 15, no. 1, pp. 82–92, Feb. 2003. [Online]. Available: https://doi.org/10.1287/ijoc.15.1.82.15157

[30] P. Toth and D. Vigo, "The granular tabu search and its application to the vehicle-routing problem," *INFORMS J. on Computing*, vol. 15, no. 4, pp. 333–346, Dec. 2003. [Online]. Available: http://dx.doi.org/10.1287/ijoc.15.4.333.24890

[31] M. Gendreau, A. Hertz, and G. Laporte, "New insertion and postoptimization procedures for the traveling salesman problem," *Oper. Res.*, vol. 40, no. 6, pp. 1086–1094, Dec. 1992. [Online]. Available: https://doi.org/10.1287/opre.40.6.1086

[32] S. Hougardy and R. Schroeder, "Edge elimination in TSP instances," in *Graph-Theoretic Concepts in Computer Science. WG 2014*, ser. Lecture Notes in Computer Science,

D. Kratsch and I. Todinca, Eds. Springer, Cham, 2014, vol. 8747. [Online]. Available: https://dx.doi.org/10.1007/978-3-319-12340-0_23

[33] J. Edmonds, "Maximum matching and a polyhedron with 0,1-vertices," *Journal of research of the National Bureau of Standards*, vol. 69B, pp. 125–130, 1965.

[34] M. Held and R. M. Karp, "The traveling-salesman problem and minimum spanning trees," *Operations Research*, vol. 18, pp. 1138–1162, 1970.

[35] M. Held and R. M. Karp, "The traveling-salesman problem and minimum spanning trees: Part ii," *Mathematical Programming*, vol. 1, pp. 6–25, 1970.

[36] N. Christofides, A. Mingozzi, and P. Toth, "State space relaxation procedures for the computation of bounds to routing problems," *Networks*, vol. 11, pp. 145–164, 1981.

[37] G. Pesant, "Counting-Based Search for Constraint Optimization Problems," in *AAAI*, D. Schuurmans and M. P. Wellman, Eds. AAAI Press, 2016, pp. 3441–3448. [Online]. Available: http://www.aaai.org/Library/AAAI/aaai16contents.php

[38] Y. Caseau and F. Laburthe, "Solving various weighted matching problems with constraints," in *Proc. CP'97*, ser. Lecture Notes in Computer Science, G. Smolka, Ed., vol. 1330. Springer, 1997, pp. 17–31. [Online]. Available: http://dx.doi.org/10.1007/BFb0017427

[39] G. Soules, "New Permanental Upper Bounds for Nonnegative Matrices," *Linear and Multilinear Algebra*, vol. 51, no. 4, pp. 319–337, 2003.

[40] A. Delaite and G. Pesant, "Counting weighted spanning trees to solve constrained minimum spanning tree problems," in *CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, ser. Lecture Notes in Computer Science, D. Salvagnin and M. Lombardi, Eds., vol. 10335. Springer, 2017, pp. 176–184.

[41] A. Z. Broder and E. W. Mayr, "Counting Minimum Weight Spanning Trees," *Journal of Algorithms*, vol. 24, no. 1, pp. 171 – 176, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0196677496908512

[42] S. Brockbank, G. Pesant, and L.-M. Rousseau, "Counting Spanning Trees to Guide Search in Constrained Spanning Tree Problems," in *CP*, ser. Lecture Notes in Computer Science, C. Schulte, Ed., vol. 8124. Springer, 2013, pp. 175–183. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40627-0_16

[43] W. Cook, Apr. 2019, Personal communication.

[44] W. D. Harvey and M. L. Ginsberg, "Limited discrepancy search," in *IJCAI*, 1995.