Università degli Studi di Ferrara

DOTTORATO DI RICERCA IN MATEMATICA E INFORMATICA

COORDINATORE DOTT. GAETANO ZANGHIRATI

THE BI-OBJECTIVE TRAVELLING SALESMAN PROBLEM WITH PROFITS AND ITS CONNECTION TO COMPUTER NETWORKS

DOTTORANDA
DOTT.SSA ELISA STEVANATO

TUTORE
PROF. CARLO FILIPPI

XXII° CICLO

ANNI 2007 - 2009

To my grandmother Elda,
always in my heart.

Abstract

This is an interdisciplinary work in Computer Science and Operational Research. As it is well known, these two very important research fields are strictly connected. Among other aspects, one of the main areas where this interplay is strongly evident is Networking. As far as most recent decades have seen a constant growing of every kind of network computer connections, the need for advanced algorithms that help in optimizing the network performances became extremely relevant. Classical Optimization-based approaches have been deeply studied and applied since long time. However, the technology evolution asks for more flexible and advanced algorithmic approaches to model increasingly complex network configurations. In this thesis we study an extension of the well known Traveling Salesman Problem (TSP): the Traveling Salesman Problem with Profits (TSPP). In this generalization, a profit is associated with each vertex and it is not necessary to visit all vertices. The goal is to determine a route through a subset of nodes that simultaneously minimizes the travel cost and maximizes the collected profit. The TSPP models the problem of sending a piece of information through a network where, in addition to the sending costs, it is also important to consider what "profit" this information can get during its routing. Because of its formulation, the right way to tackled the TSPP is by Multiobjective Optimization algorithms. Within this context, the aim of this work is to study new ways to solve the problem in both the exact and the approximated settings, giving all feasible instruments that can help to solve it, and to provide experimental insights into feasible networking instances.

Sommario

Questa tesi tratta argomenti che riguardano l'Informatica e la Ricerca Operativa. Come è noto, questi importanti campi di ricerca sono strettamente collegati fra loro. Uno dei principali settori in cui questa interazione è fortemente evidente è il Networking. Negli ultimi decenni si è vista una costante crescita di ogni genere di connessioni informatiche di rete, e la necessitá di algoritmi avanzati che aiutino ad ottimizzare le prestazioni di rete è diventata estremamente rilevante. Approcci classici di ottimizzazione sono stati ampiamente studiati e applicati da tempo. Tuttavia, l'evoluzione tecnologica richiede procedure piú flessibili e algoritmi piú avanzati per far fronte a configurazioni di rete sempre pi complesse. In questa tesi studiamo una estensione del Problema del Commesso Viaggiatore (TSP): il Problema del Commesso Viaggiatore con Profitti (TSPP). In questa generalizzazione, ad ogni vertice è associato un profitto e, a differenza del classico TSP, non è necessario visitare tutti i vertici. Si vuole determinare quei percorsi che contemporaneamente minimizzano il costo di viaggio e massimizzano il profitto raccolto. Il TSPP modella il problema di inviare un pacchetto di informazioni attraverso un rete in cui, oltre ai costi di invio, è anche importante considerare il "profitto" che queste informazioni possono accumulare durante il percorso. A causa della sua formulazione, il modo migliore di affrontare il TSPP è mediante procedure di ottimizzazione multiobiettivo. In questo contesto, l'obiettivo è di studiare nuovi modi per risolvere il problema sia in maniera esatta che approssimata, fornendo tutti gli strumenti possibili che contribuiscano alla sua risoluzione, e introducendo nuovi spunti sperimentali in istanze di networking.

Acknowledgments

First of all, I would like to thank my two supervisors: Gaetano Zanghirati (University of Ferrara) and Carlo Filippi (University of Brescia), for their continuous support in the development of this work during these three years.

I would also like to thank Professor J.J. Salazar (University of La Laguna, Tenerife, Spain), for his kindness in accepting me twice in Tenerife and for the stimulating and extremely valuable discussions on the thesis topics. Many thanks also to Prof. D. Trystram (University of Grenoble, France) for his relevant suggestions and the thesis review.

Special thanks go to Salome Rodriguez, for her help during and after my stay at the University of La Laguna.

Part of this work was done in collaboration with Prof. F.C.R. Spieksma and with Ms. S. Coene, Ph.D. (University of Leuven, Belgium), so I would like to thank them too for everything I learnt with their help.

A remind also to Nievez Luz Rodriguez, that helped me during my stay in La Laguna, she is a real good friend.

A big thanks to the friends that shared with me the office in the Ferrara University, Block B, Third floor (the only place in the world where the postcards don't arrive): Ambra, Mimma, Silvia and Elisa M. (and the three omnipresent boys: Ric, Nicola and Marcello). Thanks for these wonderful years together. I hope to spend many more years with you!

I want also thank my parents for everything they have done for me so far. If I reached this objective, I owe it to the fact that they always encouraged me to achieve new goals, leaving me always free to follow my ambitions, supporting me emotionally and ... economically. A thought also to men of my life: my boyfriend Michele and my little brother Luca. I want to thank Michele because he always left me free to decide, although my work sometimes took me away from him. I thank Luca for being Luca because, even if he is already 24 years old, and he took the degree cum laude in nuclear physics, he continues to be my little brother and I am very lucky to have him by my side (remember me when you will win the Nobel).

A thought also goes to all my relatives: my beloved grandfather, a real power of nature, my uncles: Barbara, Ivan, Loris, Nicoletta, my favorite (and one) cousin Marzia, my godfather: the legendary Marco, unique and inimitable, my sister-in-law Enrica and finally, last but not in importance, Lela and Claudia, who live far away but are always present in the important moments of my life.

Thank also to my two friends of always: Erica and Orietta, because over the years you were present near me. Thank to all!

Contents

Introduction

This is an interdisciplinary work between Computer Science and Operational Research. As it is well known, these two very important research fields are strictly connected. Among other aspects, one of the main areas where this interplay is strongly evident is Networking. As far as most recent decades have seen a constant growing of every kind of network computer connections, the need for advanced algorithms that help in optimizing the network performances became extremely relevant. Classical Optimization-based approaches have been deeply studied and applied since long time. However, technology evolution asks for more flexible and advanced algorithmic approaches to model increasingly complex network configurations. This thesis will use the tools and the language of Operational Research to study the difficult situations where, in addition to the cost of sending a piece of information through a net, it is also important to consider what "profit" this information can get during its routing. One would clearly like to minimize the former and maximize the latter, but these are usually two conflicting goals. Profits and costs do not have to be necessarily meant as money amounts, but in real life they are ultimately related to money, of course. It will be clear very soon that the modelling will generate Combinatorial Optimization problems. Combinatorial Optimization is a field extensively studied by many researchers. Due to its potential for application in many real-world problems (not only in Computer Science) it has prospered over the last few decades. But as far as real-world decision making is concerned, it is also well known that decision makers have to deal with several, usually conflicting objectives. The growth in the interest of theory and methodology of multicriteria decision making (MCDM) over the last thirty years is witness of this fact. However, it is somewhat surprising that multi-objective (or multicriteria) combinatorial optimization (MOCO) has not been widely studied yet. Only in recent years, approximately since 1990, a profound interest in the topic is evident. Since then several PhD thesis have been written, specific methodologies have been developed, and the number of research papers in the field has grown considerably.

In this work we study a particular multi-objective problem: the *Traveling Salesman Problem with Profits* (TSPP) and its bi-objective counterpart. It is a generalization of the well known *Travelling Salesman Problem*. In particular, we analyze the problem from both the theoretical and the practical points of view, developing algorithms that solve the problem both in exact and in approximated ways.

Furthermore, we analyze the topologies under which the TSPP is solvable in polynomial time. The results have been collected in a paper submitted to *Networks* [35], which is currently under revision. Preliminary results were presented at the XXXIX Annual Conference of the Italian Operational Research Society (Ischia, Italy, 2008), and other advances were presented in the VI ALIO/EURO Workshop on Applied Combinatorial Optimization (Buenos Aires, Argentina, 2008).

This work is organized as follows. The first chapter gives an introduction to most of the concepts used in the thesis. These topics are Graph Theory, Computational Complex-

ity, Polyhedral Theory. In addition, some basic optimization problems appearing in the description of the proposed algorithms are also described in this chapter.

Chapter 2 gives an overview of the state-of-the art of *multi-objective* optimization. It describes the basic mathematical principles needed to formally define the multi-objective theory and the main properties of multi-objective problems. In the last part of the chapter, the most relevant multi-objective combinatorial optimization problems are depicted, together with their main features.

Chapter 3 gives a general description of the TSPP through an exhaustive dissertation of the state-of-the-art. A specific part of the chapter accurately describes a large set of real-life applications of this problem. Furthermore, the complexity of the problem is analyzed, and both lower and upper bounds on the size of the *Pareto-efficient frontier* are given.

In Chapter 4 we describe a feasible application of the TSPP in the Medical Area. We show as a local network of hospitals that needs to share informations about clinical data or feasible treatements for some particular deseases, can be seen as a graph with edge costs that represent the time needed to send-receive the requested information, and a nodes profit that represents the quality of the service obtained. We created a specific instance and we solve it analyzing the obtained results.

In Chapter 6, the TSPP is studied from a *bi-objective* point of view, focusing on the case where the underlying graph is a tree. In particular it is developed a so-called FPTAS to find an ϵ-approximate Pareto curve for the problem. The TSPP is also analyzed under simple metrics, such as a cycle, or a star, and some algorithms are developed to find the exact set of *efficient solutions*.

In Chapter 7 we introduce an algorithm based on cutting planes within a branch-and-bound approach, typically named branch-and-cut in single-objective optimization. It generates all (supported and non-supported) efficient points in the objective space with respect to both criteria. Here are also presented some heuristic approaches that use the branch-and-cut resolution idea to find an ϵ-approximation of the Pareto-efficient frontier.

Chapter 8 deals with an extension of TSPP, studying the problem with the *time window* restriction on the nodes. In particular, feasible resolution approaches are analyzed when the underlying metric is a line. It is interesting to see how different can be the computational complexity for small changes in the starting hypotheses.

Finally, in the last chapter we describe an exact approach that build the efficient Pareto frontier for the TSPP. The algorithm we propose derives from a dynamic programming approach for the Cycle Problem. The iteration formula is linear and simple: even if the method is inefficient from a computational viewpoint, its simplicity suggests that we could use it to develop some heuristic approaches to solve the TSPP.

The contributions contained in the chapters 6, 7, 8 are parts of works currently in preparation.

Chapter 1

Mathematical Background

The aim of this chapter is to give a gentle and concise introduction to most of the concepts used later in this thesis. Anyway, only a brief overview about the main contents is given. For a deeper treatments of the topics we refer to the bibliography.

Section 1 offers a brief summary of the basic definitions in Graph Theory: we chose to not focus on these definitions more than clarity requires; they want to be only a reference for later topics. Section 2 gives an overview about the complexity theory, focusing on the analysis of the complexity function, used later to evaluate algorithms. Section 3 contains a dissertation about Polyhedra Theory, that we will use in Chapter 6 in the development of a Branch and Cut approach. Finally, in the last section of the chapter, we describe some basic optimization problems, mentioned later in the thesis as parts of developed algorithms.

1.1 Graph Theory

Graph theory is a way to modeling and studying a very large set of mathematical problems. In this section we describe only the main concepts that we use later in this thesis. For a more accurate description we refer the reader for instance to [52].

A graph is a pair $G = (V, E)$, where $V = \{v_0, \ldots, v_n\}$ is a nonempty set of elements called *nodes* or *nodes*, and E contains nodes couples (v_i, v_j) called *edges* or *arcs*, such that $E \subseteq V \times V$. Let $v_0 \in V$ be designated as the *source* node.

If a specific orientation is assigned to each arc (v_i, v_j), so that it can be crossed only in one direction, then we have a *directed* graph, or *digraph*; otherwise the graph is *undirected*. The order in the couple is meaningful: (v_i, v_j) means that the arc connecting v_i and v_j is traversed from v_i to v_j. Hence, in an undirected graph we always have $(v_i, v_j) \in E \Leftrightarrow (v_j, v_i) \in E$, while this is not necessarily true in a directed graph. In this thesis we consider only undirected graphs.

In a graph $G = (V, E)$, the *degree* $d(v)$ of a node v is the number of edges incident in v. If all the nodes of G have the same degree k, then G is said to be *k-regular* or simply *regular*.

Proposition 1.1 *The number of nodes of odd degree in a graph is always even.*

Two nodes v_i, $v_j \in G$ are *adjacent*, if (v_i, v_j) is an edge of G. Two edges are *adjacent* if they have a common endpoint. If all the nodes of G are pairwise adjacent, then G is *complete*. It follows that, in a complete graph of cardinality N, there are exactly $N - 1$ edges incident in each node.

Let $S \subseteq V$ be given. Then the edge set:

$$\delta(S) := \{(v_i, v_j) \in E \,|\, v_i \in S, v_j \in V \setminus S\}$$

is defined to be the *cut* induced by S. In other words, $\delta(S)$ contains the edges of the graph G that have an endpoint in the set S and the other endpoint in $V \setminus S$. In particular, we indicate with $\delta^+(\{S\})$ and $\delta^-(\{S\})$ the set of edges exiting and entering, respectively, in the set S.

In a similar way, we define the:

$$E(S) := \{(v_i, v_j) \in E \,|\, v_i, v_j \in S, i < j\}$$

to be the set containing edges that have both endpoints in the set S. If $E(S)$ is empty, then S is an *independent* set.

Consider two subgraphs $G_1, G_2 \subseteq G = (V, E)$, where $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with $V_1, V_2 \subseteq V$ and $E_1, E_2 \subseteq E$. We recall that $G_1 \subseteq G_2$ if and only if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.

If $G' \subseteq G$ and G' contains all the edges $(v_i, v_j) \in E$ with $v_i, v_j \in V$, then G' is an *induced subgraph* of G.

1.1.1 Paths and Cycles

In a graph $G = (V, E)$, a *path* is a sequence of nodes $P = (v_0, v_1, \ldots, v_k)$ such that $(v_i, v_{i+1}) \in E$. The nodes v_0 and v_k are the starting and the ending points of the path, respectively, and P is called a *path from v_0 to v_k* (see Fig. 1.1). The number of edges of the path P is its *length*. Note that k is allowed to be zero.

Two or more paths are *independent* if they do not share any internal node. An *elementary path* is a path in which all nodes are different.

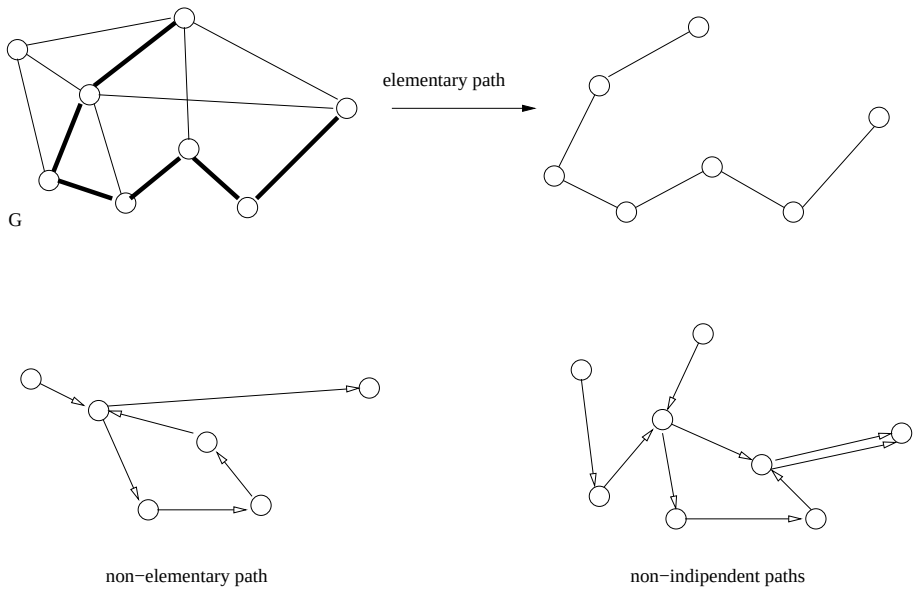

Figure 1.1: Examples of different kind of paths in the graph G.

A *cycle* is a path $(v_0, v_1, \ldots, v_p)$ of length $p > 1$ having $v_i \neq v_j$ for $i, j \in \{1, \ldots, p-1\}$ and $v_0 = v_p$. To emphasize its length p, it can be denominated as a *cycle of length p*. An edge that joins two nodes of a cycle but is not itself an edge of the cycle is called a *chord*.

A cycle in an undirected graph which visits each node exactly once and also returns to the starting node is known as a *Hamiltonian* cycle.

An *induced cycle* in G, *i.e.* a cycle in G generating an induced subgraph, is one that has no chords. If a graph contains no cycles is called *acyclic* graph.

The *distance* $d_G(v_i, v_j)$ in G between two nodes v_i and v_j is the length of the shortest path in G from v_i to v_j; if no such path exists then we set $d(v_i, v_j) = \infty$.

An undirected graph $G = (V, E)$ is *connected* if any two of its nodes are linked by a path in G; otherwise it is *disconnected*.

Definition 1.1 *A* connected component *of a graph G is a connected subgraph that is not contained in a larger one.*

Note that a component, being connected, is always non-empty (see Figure 1.2); the empty graph, therefore, has no components.

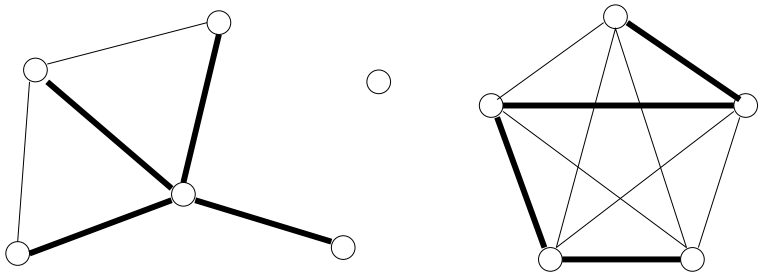

Figure 1.2: A graph with three connected components, and a minimal spanning connected subgraph in each component.

A graph $G = (V, E)$ is called *bipartite* if $\exists V_1, V_2 \subseteq V$ such that $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$ and each edge links a node in V_1 with a node in V_2. The graph is a *complete bipartite graph* if each node in V_1 is linked to all nodes in V_2. Clearly, a bipartite graph cannot contain an *odd cycle*, i.e. a cycle of odd length. The following statement characterizes this property:

Proposition 1.2 *A graph is bipartite if and only if it contains no odd cycle.*

1.1.2 Trees

A *tree* is a graph in which any two nodes are connected by exactly one elementary path. In other words, it can be seen as an undirected graph $G = (V, E)$ that satisfies any of the following equivalent conditions:

- G is a tree;
- G contains no cycles and $|E| = |V| - 1$;
- G is minimally connected, *i.e.*, G is connected but $G \setminus \{e\}$ is disconnected for every edge $e \in E$;
- any two nodes of G are linked by a unique path in G.

Figure 1.3 shows a general tree T.

A rooted tree is a graph with a node singled out as the root, in which case the edges have a natural orientation, towards or away from the root. In this case, any two nodes connected

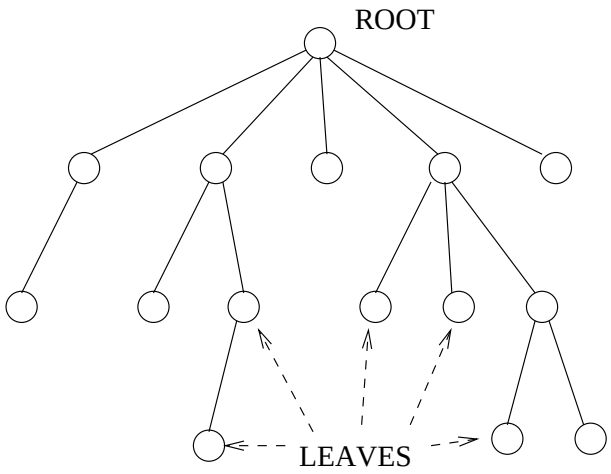

Figure 1.3: A tree.

by an edge inherit a parent-child relationship. A *parent* of a node is the node connected to it on the path to the root; every node except the root has a unique parent. A *child* of a node v is a node of which v is the parent. A *leaf* is a node without children. Rooted trees, often with additional structure such as ordering of the neighbors at each node, are a key data structure in Computer Science.

A *subtree* T' of a tree T is a tree including a node of T and all its descendants in T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a *proper* subtree.

An *n-ary tree* is a rooted tree for which each node which is not a leaf has exactly n children. The 2-ary trees are sometimes called *binary trees*.

Every connected graph G admits a *spanning tree*, which is a tree that contains every node of G and whose edges are edges of G. Figure 1.2 shows a spanning tree in each of the three components of the depicted graph.

For a deeper analysis on trees see [52], while for a dissertation about their role in Computer Programming see for instance [110].

1.1.3 Graph representation

The typical way to represent a graph, from a mathematical point of view, is through a matrix. The *adjacency matrix* and the *incidence matrix* are the structures most often used to represent the relation between edges and nodes in a graph. In particular, the adjacency matrix describes the relation between nodes and edges, while the incidence matrix represents the connections between couples of nodes in a graph G. In this thesis we chose to describe our problems through incidence matrices.

Let $G = (V, E)$ be a graph with n nodes: then the *incidence matrix* is a $n \times n$ matrix $A = [a_{ij}]$ defined by

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

With this representation it is simple to check if a given edge belongs to the graph or not. Unfortunately, the time needed to obtain the adjacency set of an edge is always $O(n)$ because it is necessary to check the full row to search for the non-zero entries. Moreover, the memory

needed to store this type of matrix is $O(n^2)$, and the time needed to explore all edges is $O(n^2)$ even if the matrix is sparse.

If the graph is weighted, the incidence matrix weights are used instead of the binary elements. If p_{ij} is the weight of edge (v_i, v_j), then the matrix becomes:

$$a_{ij} = \begin{cases} p_{ij} & \text{if } (v_i, v_j) \in E, \\ \infty & \text{otherwise.} \end{cases}$$

In this way we save memory during the storing of data related to graph G, even if the time needed to store and explore the matrix remains the same.

The choice of the structure for the representation of a graph determines, in most cases, the efficiency of the resolution algorithm. In Chapter 7 we can observe how the adoption of a new policy of graph representation took a great improvement of the algorithm performances.

1.2 Computational Complexity

The *Computational Complexity Theory* is a branch of the theory of computation in Computer Science that study the intrinsic difficulty of problems. It is based on a mathematical structure developed on Logic and Computer Science. Here, a problem is understood in the narrow sense of a task that is in principle amenable to be solved by a computer. The complexity theory gives a way to classify problems according to the resource request, by introducing mathematical models of computation, and casting computational tasks mathematically known as *decision problems*. The degree of difficulty can be quantified in the amount of resources needed to solve these problems, such as time and storage. In particular, the theory seizes the practical limits on what computers can and cannot do. Problems are the central objects in computational complexity theory. Normally, a problem is conceived as a formal language. The objective is to decide, with the aid of an algorithm, for a given input word if it is member of the formal language under consideration; this kind of problem is known as decision problem. In other words, a *decision problem* is a particular formulation of a problem for which the possible solution is a *yes-no* answer. An input word for a decision problem is referred to as problem instance, and should not be confused with the problem itself. In computational complexity theory, a problem refers to the abstract question to be solved. In contrast, an instance of this problem is a specification of its parameters, and can serve as input word for a decision problem. The way to represent instance problem is through a word over an alphabet. Usually, the alphabet is taken to be the set $\{0, 1\}$, since this corresponds to the information representation in modern computers.

To classify the computation time (or similar resources, such as space consumption), one is interested in proving upper and lower bounds on the minimum number of steps required by the most efficient algorithm that can solve the problem. The running time of a particular algorithm is measured as a function of the length $\|x\|$ of the input x. Since the running time may greatly vary among different inputs of the same length n, the running time in terms of input length n is defined as the maximum number of steps carried out by the algorithm on each of the possible inputs of length n.

1.2.1 Complexity measures

To treat concepts related to the computational resource consumption, we need to introduce the important concept of *Turing Machine*.

Definition 1.2 *Turing machines are basic abstract symbol-manipulating devices which, despite their simplicity, can be adapted to simulate the logic of any computer algorithm.*

From this definition, we can define the time required by a Turing machine M on the input x as the total number of transitions, or steps, that the machine takes before to halt and return the answer. Hence, if we introduce the concept of *complexity function* $f(n)$, we can say that a Turing machine M operates within time $f(n)$, if the time required by M on each input of length n is at most $f(n)$. Similarly, a decision problem A can be solved in time $f(n)$ if there exists a Turing machine operating in time $f(n)$ which decides A.

A *non-deterministic Turing machine* is a computational model that differs from *deterministic Turing machine* because it can branch out to check many different possibilities at once instead of only one. The non-deterministic Turing machine has very little to do with how we physically want to compute algorithms, but its branching exactly captures many of the mathematical models we want to analyze, so that non-deterministic time is a very important resource in analyzing computational problems.

A *complexity class* is a set of problems of related complexity. A typical complexity class contains the set of problems solvable by a Turing machine within time $f(n)$. Many complexity functions have been studied, but the main distinction between them is based on the asymptotic amount of operations. So, we distinguish between:

- *polynomial function*, if $f(n) = O(n^k)$;
- *exponential function*, if $f(n) = O(k^{Cn})$;

with C and k constant.

Definition 1.3 *The complexity class P contains decision problems for which exists an algorithm that solve them in polynomial time.*

Thus, P can be seen as a mathematical abstraction modeling of those computational tasks that admit an efficient algorithm.

Definition 1.4 *The complexity class NP is the set of decision problems that can be solved by a non-deterministic Turing machine in polynomial time.*

All the problems in the NP class have the property that their solutions can be checked efficiently. Since deterministic Turing machines are special non-deterministic Turing machines, it is easily observed that each problem in P is also a member of the class NP.

Definition 1.5 *The complexity class #P is the set of the counting problems associated with the decision problems in the set NP.*

Definition 1.6 *A problem A is NP-hard if the entire class of problems in the NP class can be reduced to it in polynomial time.*

Definition 1.7 *NP-complete is a subset of NP, the set of all decision problems whose solutions can be verified in polynomial time.*

Definition 1.8 *A problem is #P-complete if and only if it is in #P, and every problem in #P can be reduced to it by a polynomial-time counting reduction, i.e. a polynomial-time Turing reduction relating the cardinalities of solution sets.*

Generally, to prove that a problem is NP-complete, two thing must be showed:

- the problem must belong to the NP class;
- all problems in the NP class must be transformable in polynomial time in the given problem.

Thus, we can state that if any single problem in the NP-complete class can be solved quickly, then every problem in NP can also be quickly solved. Because of this, it is often said that the NP-complete problems are harder or more difficult than NP problems in general. Figure 1.4 shows a scheme of complexity classes.

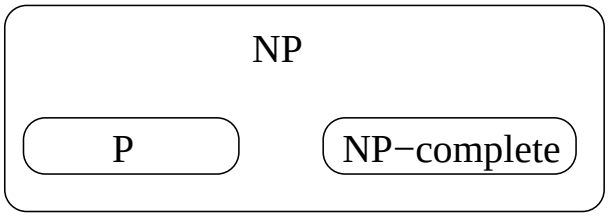

Figure 1.4: Complexity classes.

Whether P ∩ NP-complete is empty or not is unknown.

1.3 Polyhedral combinatorics

In this section we summarize some concepts relative to Polyhedral Theory, necessary to deeply understand the procedure described in Chapter 5. A detailed treatment of the Theory of Polyhedra can be found in Bachem and Grotschel [7], Grunbaum [86], Rockafellar [150], as well as in some book on Integer Linear Programming, such as Nemhauser and Wolsey [129] or Schrijver [156].

The *Polyhedral Combinatorics* is the research area in which polyhedra arise from combinatorial optimization problems. Connections between Combinatorial Optimization and continuous zero-one Linear Optimization can be established as follows.

Definition 1.9 *Given a finite set E, let $\mathcal{I} \subseteq 2^E$ be a collection of* feasible solutions*, and let $c : E \to \mathbb{R}$ be the so called* objective function*. For each set $F \subseteq E$ let $c(F) := \sum_{e \in F} c(e)$. A* linear combinatorial optimization problem *consists in finding a set $I^\star \in \mathcal{I}$ with:*

$$c(I^\star) = \min\{c(I) \mid I \in \mathcal{I}\}$$

and it is denoted by (E, I, c).

Definition 1.10 *Given a finite set E, and a set $F \subseteq E$, the* incidence vector $x^F \in \mathbb{R}^E$ *is defined as:*

$$x_e^F = \begin{cases} 1 & \text{if } e \in F \\ 0 & \text{if } e \notin F \end{cases}$$

Incidence vectors are exactly the nodes of the polytope $P_{\mathcal{I}}$:

$$P_{\mathcal{I}} = \text{conv}\{x^I \mid I \in \mathcal{I}\}$$

If we associate with the function $c : E \to \mathbb{R}$ of a combinatorial optimization problem a vector $c \in \mathbb{R}^E$, we can solve the combinatorial optimization problem by solving:

$$\min\{c^T x \mid x \in P_{\mathcal{I}}\}$$

where the solution space is defined as the convex hull of an implicitly described set of points. Actually, there are no efficient algorithms that solve an optimization problem of this kind. However, there exists a finite set of inequalities $Ax \geq b$, such that $P_{\mathcal{I}} = \{x \mid Ax \geq b\}$. Thus, we could transform the combinatorial optimization problem $(E, \mathcal{I}, c)$ in the linear program: $\min\{c^T x \mid Ax \geq b\}$. There are finite algorithms that transform one representation of the polytope $P_{\mathcal{I}}$ into the other for small problem instances.

The main problem arises when the number of constraints is too large to be represented in a computer and solved by an LP-solver. In this cases we developed an approach described in [116]. The method consists in selecting a small subset of constraints, used to compute an optimal initial solution. At this point, it checks if some of the constraints, belonging to the starting model but not to the current one, are not satisfied. When these constraints are identified, they are added to the current linear model and it is solved again. If no other constraints are violated, the optimal solution found is also an optimal solution for the starting problem, otherwise the violated constraint must be added, and the procedure restarts. This is the basic principle of the so called *cutting plane approach*, whose name derives from the fact that constraints added to the current linear model *cut off* the current solution that is infeasible for the original combinatorial problem. Thus, it is not needed to know the full set of starting constraints, but it is sufficient to have a procedure that identify only the violated ones.

Definition 1.11 *Given a bounded rational polyhedron $P_I \subseteq \mathbb{R}^n$ and a rational vector $v \in \mathbb{R}^n$, the* separation problem *is either conclude that v belongs to P_I or find a rational vector $w \in \mathbb{R}^n$ such that $w^T x > w^T v$ for all $x \in P_I$.*

Hence, the equivalence between solving an optimization problem and solving the equivalent separation problem follows.

Definition 1.12 *For any proper class of polyhedra, the optimization problem is polynomially solvable if and only if the separation problem is polynomially solvable.*

An algorithm that solves the general separation problem is called *exact separation algorithm.* A generic cutting-plane algorithm to solve the Mixed Integer Linear Programming (MILP) problem

$$\min\{c^T x^I \mid Ax^I \geq b, x^I \text{ integer for all } I \in \mathcal{I}\}$$

is outlined in Alg. 1 The cutting plane algorithm uses specific cutting planes, called *facets*, that define inequalities, but it not always stops with an optimal solution.

Branch-and-bound is an another technique used to solve hard mixed integer optimization problems. Branch and bound is a divide-and-conquer approach that try to solve the original problem by splitting it into smaller problems, for which lower and upper bounds are computed. The computation of these bounds is the most important part of the branch and bound approach, and impacts on the performance and the goodness of the solution.

Definition 1.13 *Let $F = \{x^I \mid Ax^I \geq b, x^I$ integer for all $i \in \mathcal{I}\}$ be the set of feasible solutions of a mixed integer optimization problem: $\min\{c^T x \mid x \in F\}$. A minimization problem*

$$\min\{r(x) \mid x \in R\}$$

Algorithm 1 Cutting plane algorithm for MILP problems

- Initialize the constraints system (A', b') with a small subset of the constraints system (A, b).
- Repeat
  - Compute a solution $\bar{x}$ of the problem: $\min\{c^T x \mid A'x \geq b', x \in \mathbb{R}\}$
  - If ($\bar{x}$ is not feasible for (A, b) then
    - Generate a cutting plane $(f, f_0), f \in \mathbb{R}^n$ with
      - $f^T \bar{x} \leq f_0$
      - $f^T \bar{x} > f_0$ for all $y \in \{x \mid Ax \geq b, x_i \text{ integer for all } i \in I\}$
    - Add the inequality $f^T x > f_0$ to the constraint system (A', b')
  - Endif
- Until ($\bar{x}$ is feasible)

is a relaxation of the mixed integer optimization problem if

$$F \subseteq R \text{ and } c^T x \geq r(x), \text{ for all } x \in F.$$

In this way, the solution of the relaxed problem gives a lower bound for the objective function value of the original problem it was derived from.

A basic relaxation in the cutting plane context is obtained by dropping the integrality constraints. Obviously, adding inequalities produces a tightening of the relaxation.

A branch-and-bound algorithm generates a list of subproblems deriving from the starting one. At each step, the algorithm solves one of these subproblems, computing a lower bound for it, and uses this bound to improve the lower bound of the original problem. If the lower bound computed for one subproblem is greater than the global one, then the subproblem is *fathomed*, (that is, it will not longer be considered) because its solution and the solutions of problems deriving from it cannot be better than the best known feasible solution. Consequently, no other subproblems can be derived from it. Otherwise, the solution of the relaxed subproblem is checked, to see if it is a solution for the original problem. In this case, the problem was solved, and thus, it is fathomed. Most often, the list of branch-and-bound subproblems is managed as a tree.

If the local lower bound exceeds the global lower bound and no feasible solutions were found for the active sub-problem, it is necessary to perform a branching step, to split the active subproblem in a collection of new ones, whose union of feasible solutions corresponds exactly to the feasible solution of the active subproblem. The easier way to perform this step, is to change the bounds of the variables. Suppose $i \in \mathcal{I}$ has a fractional value $\bar{x}_i$ in the LP-solution. Then, we can set to $\lceil \bar{x}_i \rceil$ the new lower bound of the i-th variable in the first new subproblem, whereas its upper bound remains unchanged, and to $\lfloor \bar{x}_i \rfloor$ the new upper bound of the second problem, whereas the lower bound remains unchanged.

When all subproblems are fathomed, and so the list of them is empty, the solution is the optimal solution for the original problem.

A *branch-and-cut* algorithm is a branch-and-bound algorithm in which cutting planes are generated throughout the branch-and-bound tree. This implies some differences in the

procedure to find a solution. In fact, instead of re-optimizing each node, the branch and cut method tries to get the most tight lower bound for each subproblem. The aim is to reduce the number of required branching in the tree through cuts and improved formulations, and use each possible technique (*i.e.*, heuristic or preprocessing) that can be useful to improve the results.

1.4 Some basic optimization problems

In this section we introduce some well-known optimization problems, used later in this thesis, describing their mathematical model, and their importance in the combinatorial optimization context. In particular, we focus our attention on the possible relations between these problems and the well known Traveling Salesman Problem (TSP), in order to find solving approaches applicable to an extension of TSP like the Travelling Salesman Problem with Profits (TSPP) that we will study in the following chapters.

We start with the *Assignment Problem*, that will appear in Chapter 6 as a relaxation of a subproblem deriving from the TSPP. The *Orienteering Problem*, the *Prize Collecting Travelling Salesman Problem* and the *Profitable Tour Problem*, are all mentioned in Chapter 5 to analyze the complexity of our problem. Finally, the *Cycle Problem*, appearing in Chapter 8, is used in the formulation of a Dynamic Programming approach to TSPP.

1.4.1 The Assignment Problem

Let us consider n workers (such as persons or machines) available to carry out n jobs. Each job has to be assigned to exactly one worker. Some workers are better suited to particular jobs than others, so there is an estimated cost c_{ij} assigned to the subject i if it performs the job j. The objective consists in finding an assignment worker-job with minimum cost. The variables used are the following:

$$x_{ij} = \begin{cases} 1 & \text{if the worker } i \text{ does the job } j \\ 0 & \text{otherwise.} \end{cases}$$

The mathematical model is:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{1.1a}$$

$$\text{subject to } \sum_{j=1}^{n} x_{ij} = 1, \text{ for } i = 1, \ldots, n \tag{1.1b}$$

$$\sum_{i=1}^{n} x_{ij} = 1, \text{ for } j = 1, \ldots, n \tag{1.1c}$$

$$x_{ij} \in \{0, 1\}. \tag{1.1d}$$

The total unimodularity of the constraint matrix guarantees than an optimal integer solution can be found by Linear Programming methods. This does no longer hold true in the multi-objective case.

The first algorithm to solve a mono-objective Assignment Problem was developed by Easterfield in 1946 [55], and consists in a non-polynomial time approach based on iterated

application of a particular class of feasible transformations; but the first polynomial-time method was presented in 1950 by Kuhn [113] with his famous Hungarian algorithm.

From a multi-objective point of view, the approaches studied in literature are focused on the determination of *supported efficient solutions*, computed through the use of methods based on convex combinations of objectives or on Goal Programming. For a complete state-of-the-art survey on this subject see [59].

1.4.2 Orienteering Problem

The *Orienteering Problem* (OP) is a generalization of the TSP. One possible formulation is the following: let $G = (V, E)$ be a graph of $|V| = n$ nodes, each one representing a city, with node v_0 representing a depot. Each node v_i is associated a profit p_i, while each edge (v_i, v_j) is associated a value c_{ij} that represents the cost to pass through the edge. The OP searches a route starting and ending in the depot that maximizes the profit, having a total cost non exceeding a fixed upper bound C.

A mathematical formulation of this problem uses the following variables:

$$x_{ij} = \begin{cases} 1 & \text{if the edge } (v_i, v_j) \text{ is visited,} \\ 0 & \text{otherwise,} \end{cases}$$

and

$$y_i = \begin{cases} 1 & \text{if the node } v_i \text{ is visited,} \\ 0 & \text{otherwise.} \end{cases}$$

We can then formulate the OP as the following 0-1 Integer Linear Programming problem:

$$\max \sum_{v_i \in V} p_i y_i \tag{1.2a}$$

$$\text{subject to} \sum_{e \in \delta(v_i)} c_e x_e \leq C \tag{1.2b}$$

$$c(\delta(v_i)) = 2y_i \text{ for all } v_i \in V \tag{1.2c}$$

$$x(\delta(S)) \geq 2y_i \text{ for all } S \subset V \text{ with } \emptyset \neq S \neq V, v_0 \notin S \text{ and } v_i \in S \tag{1.2d}$$

$$y_0 = 1 \tag{1.2e}$$

$$x_e \in \{0, 1\},\ y_i \in \{0, 1\} \text{ for all } v_i \in V \text{ and } e \in E. \tag{1.2f}$$

The constraint (1.2b) imposes that the total cost cannot exceed the bound C. The *degree equation* (1.2c) imposes that a feasible solution must contain each visited node exactly. The constraints (1.2d), called *generalized subtour Elimination Constraints* force each visited node $v_i \in V \setminus \{v_0\}$ to be reachable from the node v_0 through two paths sharing no edges. Finally, the constraint (1.2e) imposes that the depot must be visited.

This problem appeared in several routing and scheduling applications, and it was demonstrated that it is NP-hard. Many approaches were developed to compute its solutions. Tsiligirides [168], Golden, Levy and Vohra [82], Golden, Wang and Liu [83] and Chao, Colgen and Wasil [29] developed heuristic approaches, while Laporte and Martello [115] proposed exact methods. Ramesh, Yoon and Karwan [144], Leifer and Rosenwien [119] studied an LP-based bounding procedure. A branch-and-bound approach was developed by Gendreau, Laporte and Semet [71]. Fischetti, Salazar and Toth [66] introduced a branch-and-cut approach to solve it.

1.4.3 Prize Collecting Traveling Salesman Problem

The *Prize Collecting Traveling Salesman Problem* (PCTSP) was introduced by Balas[11] in 1989. It is an extension of the well known TSP but with some restrictions in the profit accumulation. We formulate the problem as follows: a traveller have to visit a set $V = \{v_0, v_1, \ldots, v_n\}$ of clients. A profit p_i is associated with the visit to the client $v_i \in V$, and a cost $c_{ij} > 0$ is associated with each edge $(v_i, v_j) \in E$: if the traveller decides to visit the clients v_i and v_j, in this order, he must pay the cost. We assume that the traveller route starts and ends in the depot, that is always served. The PCTSP consists in finding a subset $S \subseteq V$ of clients to serve such that the total cost is minimum and the total profit accumulated is at least a given lower bound L.

The variables used in the formulation are the same as for the Orienteering Problem (see the previous subsection). The mathematical formulation of the problem is the following:

$$\min \sum_{e \in E} c_e x_e \tag{1.3a}$$

$$\text{subject to} \sum_{v_i \in V} p_i y_i \geq L \tag{1.3b}$$

$$c(\delta(v)) = 2y_i \text{ for all } v \in V \tag{1.3c}$$

$$x(\delta(S)) \geq 2y_i \text{ for all } S \subseteq V \text{ with } \emptyset \neq S \neq V, v_0 \notin V \text{ and } v_i \in S \tag{1.3d}$$

$$y_0 = 1 \tag{1.3e}$$

$$x_e \in \{0,1\},\ y_i \in \{0,1\} \tag{1.3f}$$

The constraint (1.3b) imposes that the sum of profits associated to the visited nodes be at least a fixed positive value L. Constraints (1.3c) impose that each visited client v_i, have a predecessor and a successor in the tour. Otherwise, if v_i is not served, then it is not an extreme of any edge in the tour. Constraints (1.3d) are the well known generalized subtour elimination constraints, and impose the presence of no multiple subtour in the solution. We can easily note that the number of these type of constraints is very high, and this makes the problem intractable. To overcame this difficulty some approaches has been developed. Fischetti and Toth [67] presented an additive approach based on a branch-and-bound technique to obtain solutions through a progressive generation of lower bounds, while Balas [11, 13] obtained some polyhedral results.

1.4.4 The Cycle Problem

Let $G = (V, E)$ be a undirected graph with c_e the cost associated to the edge $e \in E$. The *cycle problem* consists in finding a cycle of minimum cost that pass through each node at most once. If negative cost circuits are allowed, this problem is NP-hard: this easily follow by reducing the TSP to the cycle problem by subtracting a large positive constant from each edge. Balas [12] and Bauer [19], among others, studied the facial structure of a directed and undirected polytope respectively associated with the problem.

A mathematical description for the undirected cycle problem is given below. For the binary variables x_e we have $x_e = 1$ if and only if the edge e belongs to the solution. The variable y_i is set to 1 if node v_i is visited, otherwise it is set to 0.

$$\min \sum_{e \in E} c_e x_e \tag{1.4a}$$

$$\text{subject to} \sum_{e \in \delta(v_i)} x_e = 2y_i \quad \text{for all } v_i \in V \tag{1.4b}$$

$$\sum_{e \in \delta(S)} x_e \geq y_i + y_j - 1 \quad \text{for } v_i \in S \quad v_j \notin S \tag{1.4c}$$

$$x_e \in \{0,1\} \text{ for all } e \in E \tag{1.4d}$$

$$y_i \in \{0,1\} \text{ for all } v_i \in V \tag{1.4e}$$

The constraints set is similar to that of TSP one. Constraints (1.4b) impose that if an edge incident to a node v_i is included, then the node v_i must be visited. The constraints (1.4c), (1.4d) and (1.4e) are the generalized subtour constraints, and the integrality bounds for the variables, respectively.

In the literature we can find many resolution approaches for this kind of problem. Bauer [19] and Balas and Oosten [16] studied the undirected and directed cycle polytope, while Salazar [152] gave a polyhedral study for the Cycle Polytope with loop variables in the undirected case. Finally, a polynomial algorithm was developed by Coullard and Pulleyblank [41] in the case of edge with nonnegative-costs.

1.4.5 Profitable Tour Problem

The *Profitable Tour Problem* (PTP) was introduced for the first time by Dell'Amico *et al.* [48]. It is related to both the constrained flow problems and the vehicle-routing problems.

It is an extension of the well known TSP in which the objective function maximizes a profit function obtained by the difference between two terms:

- the sum of the profits associated to the visited nodes;
- the cost associated to the total distance covered.

A possible illustration of the problem is the following: a traveller can visit a set $V = \{v_0, v_1, v_2, \ldots, v_n\}$ of clients, where v_0 represents the depot. The profit associated to the visit of each client $v_i \in V$ is a positive value p_i. The main difference with respect to the TSP is that the traveller is not forced to visit all clients: he can decide to visit only a subset of them. The distance $c_{uv} > 0$ is the cost that the traveller pays if he decides to visit the clients u and v in this order. It is assumed that the traveller route starts and ends in the depot, that is always served. The PCTSP consists in finding a subset $S \subseteq V$ of clients to serve and an ordering of elements of S, such that the total profit accumulated is maximum.

The mathematical formulation of the problem is the following:

$$\max \sum_{v_i \in V} p_i y_i - \sum_{e \in E} c_e x_e \tag{1.5a}$$

$$\text{subject to} \sum_{e \in \delta(v_i)} x_e = 2y_i \quad \text{for all } v_i \in V \tag{1.5b}$$

$$\sum_{e \in E(S)} x_e \leq \sum_{u \in S \setminus v} y_u \quad \text{for all } S \subseteq V \setminus \{v_0\} \text{ and all } v \in S \tag{1.5c}$$

$$y_1 = 1 \tag{1.5d}$$

$$x_e \in \{0, 1\} \tag{1.5e}$$

$$y_i \in \{0, 1\}. \tag{1.5f}$$

In the objective function (1.5a) the contributions given by the cost and by the accumulated profit are clearly recognizable. The constraints (1.5b) impose that each visited client v_i (*i.e.*, such that $y_i = 1$) has a predecessor and a successor in the tour. Otherwise, if u is not served, then it is not an extreme of any edge in the tour. Thus, the generalized subtour elimination constraints, defined by (1.5c), imposes the presence of no multiple subtour in the solution.

A branch-and-bound algorithm to find the exact solution of PTP is described in Feillet *et al.* [64], while some meta-heuristic were proposed by Gendreau, Herts, Laporte [70] and by Mladenovic and Hansen [127]. Finally, new heuristic approaches were presented by Archetti *et al.* [2].

1.4.6 The 0–1 knapsack problem

Knapsack problems have been intensively studied, both from a theoretical and a practical point of view. The theoretical interest arises mainly from the simple structure of the problem that allows to exploit a number of combinatorial properties. From the practical point of view, these problems describe many real-life situations: cutting stock, cargo loading or capital budgeting. For a full description of all variants of the knapsack problem with the main resolutive algorithm see Martello and Toth [124].

The aim of the ordinary 0–1 knapsack problem is to find the optimal combination of items to pack in a knapsack under a single constraint on the total allowable resource, where all coefficients in the objective function and in the constraint are positive. Formally, we have a set of N items, each one with a weight w_i and an utility c_i. Let W be the maximum weight that can be put in the knapsack. Then, we want to choose the best combination of items in order to maximize the utility, subject to the capacity constraint. If we assume that the variable x_i is set to 1 if the object i is put into the knapsack and is set to 0 otherwise, we can formulate the problem as follows:

$$\max \sum_{i=1}^{N} c_i x_i \tag{1.6a}$$

$$\text{subject to} \sum_{i=1}^{N} w_i x_i \leq W \tag{1.6b}$$

$$x_i \in \{0, 1\}. \tag{1.6c}$$

The knapsack problem is one of the fundamental NP-hard combinatorial optimization problems. Thus, it is not surprising that many of the proposed algorithms to solve it are based on

either implicit enumeration methods (such as dynamic programming) or branch and bound or heuristic (or meta-heuristic) procedures.

From this problem a variety of knapsack-type problems can be derived in which a set of different entities are given, each one having an associated value and a size, and the goal is to select one or more disjoint subsets so that the sum of the sizes in each subset does not exceed a given bound, and the sum of the selected values is maximized.

In this thesis we consider a specific variant of the knapsack problem: *the partially ordered knapsack* problem. This problem differs from the standard one because it takes precedence relations between items into accounts. These precedence relations are modeled using a graph where each node corresponds to a specific item. Then, item i is a predecessor of item j if $(v_i, v_j) \in E$. An item can be selected if all items that precede it in the graph have been included. The graph that model the precedence relations is an *out tree i.e.*, a directed tree where all arcs are oriented away from a distinguished root node. More insights into the solution approaches for this problem are given by Johnson and Niemi [99] and by Samphaiboon and Yamada [153].

Chapter 2

Multicriteria Optimization

Combinatorial Optimization is a field extensively studied by many researchers. Because of its strong potential for real world applications, it has received increasing attention over the last few decades. A state-of-the-art survey can be found in [50].

The main difficulty that can be encounter in Combinatorial Optimization is the limitation in the modeling of problems. In fact, sometimes, decision maker have to deal with several, usually conflicting, objectives. *Multi-Objective Optimization* allows a degree of freedom which is lacking in mono-objective optimization. Indeed, it aims to simultaneously optimize two or more conflicting objectives subject to certain constraints. Thus, it is surprising that the scientific interest for this field became evident only in recent years. A few paper in the area have been published in the seventies, while the classical problem have been investigated in the eighties. Only since 1990 the interest for this matter has grown, and the number of research papers increased considerably.

Multi-objective optimization problems (MOP) arise in various situations: product and process design, finance, aircraft design, oil and gas industry, automobile design, network, scheduling problems and in general all those situations where optimal decisions need to be taken in the presence of trade-offs between two or more conflicting objectives. Some example of multi-objective optimization problems can be the maximization of profit and minimization of the product cost, or the maximization of performances and minimization of the fuel consumption of a vehicle, or minimization of weight and maximization of strength for a mechanical component. If a multi-objective problem is well formed, there should not be a single solution that simultaneously minimizes or maximizes each objective to its fullest. In each case we are looking for a solution such that each objective is optimized to such an extent that, if we try to optimize it any further, then the other objective(s) will suffer as a result. Thus, the goal in the setting up and solving a multi-objective optimization problem is to find such a solution, and quantifying how better is this solution compared to the others (indeed it will not be unique, in general). Then, at the end, a single solution must be selected: it will reflect the tradeoffs set by the user on the various objective functions. The user that selects one solution instead of another is called *decision maker*. The decision maker is a "human": so, multi-objective optimization tries to model his preferences and choices. There are many theories for the modeling of choices, for example the multi-attribute utility theory (see [106]) that does not accept any solution of equal rank, or the multicriteria decision aid theory (see [151]), that accepts solution of equal rank, trying to reproduce the selection processes of several decision makers.

To have an overview of the various existing approaches to solve a multi-objective problems see [149]. In particular, for examples of bi-objective and multi-objective methods in

scheduling problems see chapter 9 of [149], Jeannot *et al.* [98] and Dutot *et al.* [54].

Based on the decision maker role, the strategy to decide how to proceed are divided in four categories:

a posteriori optimization methods: the entire optimal solution set is computed and then shown to the decision maker, that will pick up what he considers to be best solution;

a priori optimization methods: the decision maker specifies his preferences before the execution of the optimization model, so only those solutions will be generated that will fit his preferences;

methods without preferences: the decision maker doesn't have any role in this type of methods, so any solution satisfies the requirements;

interactive methods: the decision maker specifies his preferences during the execution of the algorithm. In this way, it is possible to drive the solution towards his preferences.

In all cases, a large part of the solution methods for the multi-objective programming are based on the reduction of the original problem in a new mono-objective one. The technique that allows this type of operation is called *scalarization*.

In this chapter, we want to give an overview of the state-of-the art of multi-objective optimization. We begin with an introduction on the basic mathematical principles needed to formally define the multi-objective theory, then we describe the main properties of multi-objective problems. Finally we depict the most relevant multi-objective Combinatorial Optimization problems (MOCO) with their main characteristics.

2.1 Basic Principles

An optimization problem is defined as the search of a optimum (that can be a maximum or a minimum) for a given function. Mathematically speaking, a multi-objective optimization problem has the following form:

$$\min_{x \in X} \big(f_1(x), f_2(x), \ldots, f_k(x)\big) \tag{2.1}$$

where $k \geq 2$, $f_i : \mathbb{R}^n \to \mathbb{R}$ for $i = 1, \ldots, k$, and $X \subseteq \mathbb{R}^n$ represents the feasible set. Typically, in Combinatorial Optimization the objective functions are of two types:

- sum objective: $f(S) = \sum_{a \in S} w(a)$;
- bottleneck objective: $z(S) = \max_{a \in S} w(a)$;

where $S \subseteq X$ and $w : S \to Z$ is some weight function, where Z is defined in the following:

Definition 2.1 *Let $f_1, f_2, \ldots, f_k$ be k performance criteria. Then, we define the* criterion space *as:*

$$Z = \{z \in \mathbb{R}^k \mid z = (f_1(\sigma), f_2(\sigma), \ldots, f_k(\sigma), \sigma \in \mathcal{I}\}$$

where $\mathcal{I}$ is the feasible solutions set of an optimization problem.

At each *decision vector* $x \in \mathbb{R}^n$ we can associate a vector $z = (f_1(x), f_2(x), \ldots, f_k(x))^T \in \mathbb{R}^k$, called *objective vector* in the criterion space.

Since no solution optimizes simultaneously all objectives, one will search for an acceptable trade-off instead of an optimal solution. This compromise must be such that no strictly better solutions exists, even if some solutions might be considered as equivalent. This involves a partial ordering of the objective space, defined by a *dominance relation*. The latter is used to characterize the *Pareto Efficiency*, a concept that replaces that of optimal solution in single objective optimization problems.

Definition 2.2 *Let $z^1, z^2 \in \mathbb{R}^k$ be two solution vectors of (2.1). We say that z^1 dominates z^2 in a minimization problem when:*

$$z_i^1 \leq z_i^2 \ \forall i = 1, \ldots, k \text{ and } \exists j \in \{1, \ldots, k\} \text{ such that } z_j^1 < z_j^2$$

The main difficulties in these type of approaches is due to the fact that most often the objectives are conflicting. In other words, a solution that minimize an objective generally does not minimize the others.

Definition 2.3 *We define the **ideal vector of objectives** as the vector that contains the optimal values of each objective function:*

$$z_i^{id} = f_i(x^{i^*}) = \min_{x \in X} f_i(x) \tag{2.2}$$

This type of solution is very difficult to reach, but it is used as the goal in many approximation methods.

So, we can give the definition of Pareto optimality for a multi-objective problem:

Definition 2.4 *A decision vector $x^* \in$ X is a Pareto optimum for the problem* (2.1) *if there are no other vectors $x \in X$ such that*

$$f(x) \leq_P f(x^*)$$

where $\leq_P$ is the dominance relationship.

Definition 2.5 *The* efficient frontier *for problem (2.1) is the set of its Pareto optimal points.*

Thus, we can see the Pareto points as equilibrium points in the efficient frontier of the set Z.

The *efficient set* and *Pareto frontier* contain all the Pareto efficient solutions and all the non-dominated points in the objective space. Since the efficient set is defined on the solution space, while the Pareto frontier is defined on the objective space, the cardinality of the efficient set is always greater than or equal to the cardinality of the Pareto frontier. This happens because there might be many feasible solutions that correspond to the same point in the objective space. Multi-Objective optimization can approximate the Pareto frontier to provide a set of equivalent solutions to the decision maker, who will then be aware of many equivalent tradeoffs. This should help him to take a decision. In some cases, when the size of the efficient set is reasonable, it is even possible to provide the exact Pareto frontier to the decision maker.

Let us consider a particular case: the general bi-objective optimization problem

$$\begin{aligned} &\max f_1(\sigma), \quad \min f_2(\sigma) \\ &\text{subject to } \sigma \in \Omega \end{aligned} \tag{2.3}$$

where Ω is the feasible region defined by the constraints, then it follows that:

Proposition 2.1 *A solution σ^* is* Pareto-optimal *if there is not other $\sigma \in P$ such that $f_1(\sigma) \leq f_1(\sigma^*)$ and $f_2(\sigma) \geq f_2(\sigma^*)$, where at least one of the inequalities is strict. If σ^* is Pareto-optimal then $(f_1(\sigma^*), f_2(\sigma^*))$ is an* efficient point.

Solutions belonging to the objective space can be divided in two important sets, defined in the following:

Definition 2.6 *Efficient solutions which are not optimal for any scalarization obtained by using the weighted sum:*

$$\min_{x \in X} \sum_{k=1}^{p} \lambda_k f_k(x)$$

are called unsupported efficient solutions. *Those that are optimal for some weighted sum problem are called* supported efficient solutions.

Many methods use this distinction to compute the efficient Pareto frontier in two different phases: in the first one they search supported solutions, while in the second phase they compute unsupported solutions. We can find some of these approaches in Ehrgott [56], Lee and Pulat [117], Ramos et al. [145], Ulungu and Teghem [172], Visee et al. [175]).

2.2 MOCO Properties

By its nature, multi-objective optimization deals with discrete problems, although objectives are usually linear functions. As a consequence of this fact, it is usually not possible to determinate the entire efficient frontier with an aggregation of objective functions through a weighted sum. Indeed, there exist efficient solutions that are not optimal for any weighted sum of the objectives. This remains true also for the special case in which the constraints matrix is totally unimodular. These solutions are called *non-supported* efficient solutions (NE), while the remaining ones are called *supported* efficient solutions, (SE). The set of non-supported solutions is very important in the Pareto frontier: in fact, its cardinality is greater than the cardinality of the supported set. So, the time needed to solve a MOCO problem grows considerably if we decide to compute all efficient solutions in the *non-supported* set.

It was demonstrated that the problem of counting the number of supported and non-supported efficient points is #P-complete, while finding all efficient solutions in the Pareto frontier is NP-hard, even for these problems having efficient mono-objective solution methods. Some results on this matter can be found in [57, 61, 159]. Therefore, a lot of heuristics were implemented in order to reduce the computational time, and the obtained results are interesting. For example, in [57] we can find some general results on approximating the efficient set by a single solution, while in [140] it is used the Tchebycheff metric to measure the error. Another recent example of approximation scheme is given for instance in [98].

Another important aspect of MOCO problems is that the number of efficient solutions grows exponentially with the problem size. Hence, it is impossible to define a method that finds all efficient solutions in polynomial time. This result was demonstrated for a lot of problems, such as for instance the assignment problem [88] or the travelling salesman problem [161]. Consequently such problems are called *intractable*. In particular, it was shown for the knapsack problem [175] that the number of supported solutions grows linearly with the problem size, while the number of non-supported ones grows exponentially.

2.3 Solution methods for multi-objective programming

Usually, in the multi-objective programming (MOP), the resolution methods are chosen in accordance with role of the decision maker. Thus, the choice of the resolution approach depends on the method used by the decision maker to select the solutions. The most relevant approaches are listed in the following:

- the *goal programming* approach is the better to compute the Pareto set when the decision maker chooses solutions through an *a priori method*, *i.e.* when the preferences are given before the beginning of the process;
- heuristic methods gives an approximation of the solution set in a reasonable time. They are generally used when the decision maker selects solutions at the end of their generation, *i.e.* with an *a posteriori* method;
- *interactive methods* solve the problem through the introduction of computing steps alternated with dialogue steps to drive the search of solutions in the correct way. They are generally used when the decision maker introduces his preferences during the search of solutions. A lot of practical problems are solved in this way.

Thus, the appropriate resolution mode is chosen according to the situation of the decision process, and it can be exact or approximated.

2.3.1 Exact methods

In this section we describe the main existing resolution methods that compute the entire Pareto frontier of MOCO problems. A large part of resolution approaches consists in the combination of all objective functions into a single criterion, i.e. in the *scalarization* of the objectives. The combination is then parameterized so that optimal solutions for single-objective programs correspond to Pareto outcomes for the multi-objective problem. The most popular method to construct parameterized single objectives is the *weighted sum scalarization*. The algorithm proceeds by solving a sequence of subproblems for selected values of the parameters. The objective function:

$$\min \big(z_1(x), z_2(x), \ldots, z_n(x)\big)$$

becomes:

$$\min_{x \in X} \sum_{j=1}^{n} \lambda_j z_j(x)$$

where $0 \leq \lambda_j \leq 1$ and $\sum_{j=1}^{n} \lambda_j = 1$. Hence, it is possible to find all supported efficient solutions by varying λ_j (see [75] and [97]). Usually, parametric methods are used to solve this type of problem for all values of λ_j. This method was applied to a lot of multi-objective problems, for example to transportation problems [45], network flow problems [87, 157] or location problems [123]. The reader can find some example of resolution approaches in case of sum or bottleneck objectives in [126] and [142].

Another way to solve multi-objective combinatorial optimization problems consists in the minimization of the distance between the solution and an ideal point (2.2), that represents the best solution reachable (see [178]). To measure the distance between the ideal point and the efficient solution is generally used the Tchebycheff norm:

$$\min_{x \in X} \left(\max_{j=1,\ldots,n} \lambda_j |z_j(x) - z_j^I| \right)$$

Unfortunately, when we consider sum objectives, this type of problem belongs to the class of the NP-hard problems hence, even if it is possible to compute the entire efficient set, this method is not frequently used. Actually there are no approaches solving this problem using other types of norm.

A special approach is the *goal programming* (see [96] and [118]), in which the decision maker decides a target value for each objective function. The aim is to minimize the distance between the objective value and the target. Sometimes this approach is not considered part of multi-objective optimization still it is used frequently.

Ranking methods are frequently used for bicriteria optimization problems. These approaches define a lower and an upper bound on the objective values of efficient solutions. The ideal point $z^I = (z_1^I, z_2^I)$, having:

$$z_j^I := \min_{x \in X} z^j(x),\ j = 1, 2$$

defines a lower bound, while the Nadir point $z^N = \left(z_1^N, z_2^N\right)$, having

$$z_j^N := \min_{x \in X} \{z^j(x) \mid z^i(x) = z_i^I \text{ and } i \neq j\}$$

with $j = 1, 2$, defines an upper bound. This method computes the efficient set starting from a solution with $z^1(x) = z_1^I$ and searching other solutions in order to reach z_1^N. It was used in the shortest path problem [33] and in the transportation problem [51]. It is not obvious how to generalize this method to problems with more than two objective functions. The main difficulty is that it is not clear how to obtain the Nadir point when the number of objective functions is greater than two [112].

Many approaches use methods taken from single-objective optimization to solve multi-objective problems. One of these is *dynamic programming*. Dynamic programming is a general recursive decomposition technique for optimization problems. It is numerically feasible only for special classes of (typically discrete) problems, but when the structure is favorable, it is often the best method to use. The method solves a sequence of subproblems using a recursion formula. A procedure of this kind can work efficiently with multi-objective problems: the reader can find applications to the bi-objective travelling salesman problem in [35], or to shortest path problems in [27] and [90], or to transportation problems in [68] and [148].

Two widely applied methods used to solve MOCO problems are *branch-and-bound* and *branch-and-cut*. Bounds and cuts are computed to delimitate the feasible region where to search for the solutions. These type of methods combine optimality of returned solutions (like dynamic programming or greedy algorithms, when the problem fulfills the required properties) with adaptability to a wide range of problems (like metaheuristics). An efficient multi-objective branch-and-bound scheme is therefore likely to be useful in many contexts. In MOCO, during the branch-and-bound procedures, the set Upper Bound(UB) of the best solutions found so far is kept. To be efficient, the branch-and-bound procedure has also to manage lower bounds on the sub-problems. In the literature, a large number of proposed branch-and-bound algorithms use the ideal point of a sub-problem for that goal. We can find some applications of this procedure to the spanning problem tree in [163], to purchaser problems in [116] and to the knapsack problem in [171] and [173]. Anyway, most of the algorithms used to solve multi-objective optimization are modifications of this kind of procedure: see for example [172] for the assignment problem, [56] and [117] for the network flow problem.

Finally, a general framework to find exact solutions is the *two-phase method*. In the first phase supported efficient solutions are computed through the scalarization technique, that

reduces the objectives to a single one by a weighted sum. In the second phase, information about supported efficient solutions computed in Phase 1 are used to reduce the search space to obtain a restricted area where to search the non-supported solutions. This procedure is frequently used, mainly in the bi-objective problem. See Section 3.4 for a deeper description of it.

2.3.2 Approximation methods

The approximation solution methods, called *heuristics and metaheuristics*, were developed in the last decades (see Osman and Laporte [131]).

In an optimization context the term heuristic is used in contrast to methods that guarantee to find a global optimum such as, *e.g.*, the *Hungarian method* for solving the assignment problem or implicit enumeration schemes such as branch-and-bound and dynamic programming. The interest in this type of solving methods arose from the difficulty to solve some combinatorial optimization problems.

According with [146], a *heuristic* is defined as a technique which seeks good (*i.e.*, near-optimal) solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality. Often heuristics are problem-specific, so that a method which works for one problem cannot be used to solve a different one. On the other side, *metaheuristic* are very efficient strategies that manipulate subordinate heuristics in order to exploring the search space (see [131]). Meta-heuristics include, but are not limited to, constraint logic programming, genetic algorithms, evolutionary methods, neural networks, simulated annealing, tabu search, non-monotonic search strategies, greedy randomized adaptive search, colony systems, variable neighborhood search, scatter search, and their hybrids. It can be applied to a large number of problems.

Such methods give a good tradeoff between the quality of an approximation of the efficient solution set and the computational time. The first attempt to use these techniques was made by Schaffer [155] on multiple objective genetic algorithms, while Serafini [160] started a stream of research on multiple extensions of local-search-based metaheuristics. Recently, international publishers, as *Journal of Heuristics* or *Foundations of Computing and Decision Sciences*, dedicated special journal issues to these methods.

The two most used approaches are the *local search* and the *population-based-methods*.

Methods of local search in objective space start from an initial solution, and then approximate a part of the nondominated frontier corresponding to a given search direction $d \in \mathbb{R}^n$. A procedure to aggregate the objective in a single one through a weighted sum focalizes the search in a specific part of the non-dominated frontier. The procedure is repeated for all possible directions in order to explore the entire objective space. The efficiency of these methods depend strongly on the definition of the direction d.

On the contrary, the population-based methods generate a population of solutions that contribute to the evolution process toward the non-dominated frontier. Most operational procedures are based on genetic algorithms: in [34] more than 320 references to papers using this technique to solve combinatorial problems are reported.

Simulated annealing

The *simulated annealing method* was introduced in multi-objective optimization by Serafini [160]. All the other approaches of this type developed since then are closely related to the original single objective method.

A lot of definitions and acceptance rules were defined in the last years. In this section we report the two most frequently used.

The first one was developed by Ulungu and Teghem [170]. Their approach is a direct derivation of the simulated annealing principle for handling multiple objectives. It starts from a randomly generated solution $x^n = 0$ and a neighborhood structure $N(x^n)$ for $n = 1, \ldots, k$, it computes a neighbor $x \in N(x_n)$ through a set of weights d_i defining search directions. If we consider two solutions x^1 and x^2, and define Δz^j as the difference between the two values of the j^{th} objective function evaluated at the solutions x^1 and x^2, the comparison these k differences between can give exactly one of three cases:

1) $\Delta z^j \leq 0$ for all $j = 1, \ldots, k$;

2) $\exists j, j' \in \{1, \ldots, k\}$ such that $\Delta z^j < 0$ and $\Delta z^{j'} > 0$;

3) $\Delta z^j \geq 0$ for all $j = 1 \ldots, k$.

At the n^{th} step a neighbor x is accepted if it dominates x_n. The main idea consists in the selection of solutions of type 1, and in the use of a weighted norm component in the acceptance of a solution of lower quality (Cases 2 and 3). In the first version of this approach, a neighbor of Case 2 was usually accepted. In the last version of the method the acceptance rule was revised to include the search direction in the decision.

The difference between two solutions was measured through a scalarizing function $S(f(x), \lambda)$. This function makes a "local aggregation" of objectives in order to compute the "weighted distance" between $f(x)$ and $f(x_n)$. The seek of new neighbors can stop after a specific number of iterations or when a good solutions set is reached.

The second method was developed by Czyzak and Jaszkiewicz [43, 44], and it is a combination between simulated annealing principles and genetic algorithms. They determinate a set S of starting solutions, and optimize them by iteratively generating neighbor solutions that can be accepted based on a probabilistic strategy. For each solution $x \in S$, weights are chosen to increase the probability of moving it away from its closest neighbor in S. The interaction between generated solutions in the objective space gives information about the generation of new solutions. This kind of exploration technique leads to a uniformly generated approximation of the Pareto frontier.

Tabu search

Tabu Search (TS) is a local search guided by a selection function which is used to evaluate candidate solutions. To avoid being trapped in local optima, a TS heuristic has two important features: the move operator and the tabu list. A new solution (candidate solution) is produced by the move operator slightly perturbing a current solution. The set of all candidate solutions produced by the move operator is called the *neighborhood* of the current solution. The best candidate solution is the one with the best objective function value in the neighborhood. To avoid cycling, some aspect(s) of recent moves are classified as forbidden and stored in a tabu list for a certain number of iterations. In canonical TS, in each iteration, the algorithm is forced to select the best move which is not tabu. In some cases, however, if a tabu move improves upon the best solution found so far, then that move can be accepted. This is called an *aspiration criterion*. After the neighborhood of the current solution is investigated and the best candidate is determined, the tabu list is updated, the best candidate is assigned to the current solution, and the entire process starts again. The search continues until a predetermined stopping condition is satisfied.

The first attempt to use tabu search to solve a multi-objective combinatorial optimization problem is due to Gandileaux *et al.* [69]. They use a scalarizing function and a reference point, and perform a series of tabu processes guided automatically in the objective space by the current approximation of the non-dominated frontier. This approach was capable to generate both supported and non-supported efficient solutions. It uses an ideal point z^U as reference point and a scalarizing functions $S(x, \lambda)$ to browse the non-dominated frontier. So, let us consider, at the n^{th} iteration, the solution x_n and its sub-neighborhood $N(x_n)$ obtained according to a move $x_n \rightarrow x$ defined according to the structure of the feasible domain X. The new solution obtained x_n^1 is selected from the list of neighbor solutions as the best one according to the current search direction following $S(x, \lambda)$. A tabu memory is kept connected with objectives, it contains an improvement measure such as indifference or weak improvement or strong improvement, used to update the search direction to find the efficient frontier.

A new tabu search approach was developed by Pires *et al.* [139]. Their approach includes two phases: the first one uses a list of tabu moves, while the second one is a diversification phase. The diversification phase starts from a previously generated solution, and tries to generate new solutions that embody different features, searching for them in regions not yet explored. To do this, it uses a *frequency memory* collected from data since the process beginning. This memory stores the number of times each variable takes a value different from 0. Each variable can change its value if the move is not tabu, and if its frequency is smaller than a threshold initialized as the average frequency of variables.

Genetic algorithms (population-based methods)

Genetic algorithms (GA) are popular metaheuristic methods. The concept of genetic algorithm was inspired by the *evolutionist theory* explaining the origin of species. In fact, GA operate with a collection of randomly generated solutions, called a *population*. From these solutions GA iteratively generate new solutions in two different ways: *crossover* and *mutation*. Crossover takes the best existing solutions to generate a new one, while mutation generates new solutions randomly.

The importance of genetic algorithms in multi-objective optimization derives from the ability of these algorithms to simultaneously explore different regions of a solution space. They make possible to find a set of solutions for difficult problems with non-convex, discontinuous, and multi-modal solution spaces.

The *vector evaluated genetic algorithm* (VEGA) [155], was the first genetic algorithm used to approximate the Pareto optimal set by a set of non-dominated solutions. In VEGA the population is divided in equal sized sub-populations. New solutions are searched in the sub-populations using proportional selection for crossover and mutation.

An example of the ranking type approach is *SPEA* [180]. It uses a ranking procedure to assign better fitness values to non-dominated solutions at under-represented regions of the objective space. In SPEA, an external list E of fixed size stores non-dominated solutions that have been investigated thus far during the search. This kind of procedure was proposed by Goldberg and Richardson [79] in the investigation of multiple local optima for multi-modal functions.

One of the most important things in the search of new solutions is the maintenance of different populations to obtain solutions uniformly distributed over the Pareto frontier. *Fitness sharing* promotes the search in unexplored sections of the Pareto frontier by reducing fitness of solutions in densely populated areas.

The *crowding distance approaches* aim to obtain a uniform spread of solutions along the best-known Pareto frontier without using a fitness sharing parameter. NSGA-II [46] uses this type of approach: this crowding distance measure is used as a tie-breaker in a selection technique called the *crowded tournament selection operator*.

The reader is referred to [111] for a good explanation of existing genetic algorithm procedures to solve multi-objective optimization problems.

Chapter 3

The Traveling Salesman Problem with Profits

In this chapter we give a general description of the Traveling Salesman Problem with Profits (TSPP) through a dissertation over the state-of-the-art. We start with a overview on works available in the literature, then we give a deep description of the real-life applications of TSPP. Finally we briefly analyze some related problems, that will be studied and further discussed in the next chapters.

The TSPP is a generalization of the well known *Travelling Salesman Problem* (TSP). It can be described as follows: let us consider the situation in which a set of customer is given, but only a subset may be selected and served. This kind of situation is becoming frequent, for example in the cases in which shippers post demands on the web, and so the carriers have to decide to which demand offer the service. The decision has to be taken starting from the study of two different parameters: the profit and the cost. In fact, a specific profit is associated to each customer and this profit will be gained if that customer is serviced, while reaching a customer from another one has a known cost. Considering a given starting point (generally referred to as the *source* or the *depot*), the service carrier must decide which customers have to be serviced to maximize the accumulated profit and minimize the costs, supposing to finally closing the service to the starting point again. Actually, we assume that the tour between customers that the carried decides to visit is elementary, which essentially means that the profits are available only once. A recent survey by Feillet, Dejax, Gendreau (2005) defines those problems as Traveling Salesman Problem with Profits (TSPP). The objective function may be:

- the maximization of the collected total profit and the minimization of the total traveling cost, or
- the optimization of a combination of both (*Profitable Tour Problem*).

Hence, the TSPP can be seen as the bi-criteria version of the TSP, where two opposite objectives need to be optimized: one pushing the salesman to travel (to collect the profits associated with nodes) and the other inciting him to minimize the travel costs (with the right to drop nodes). In this light, solving the TSPP should result in finding a set of feasible solutions such that neither objective can be improved without deteriorating the other.

Many approaches developed to find solutions of this problem, address the model to a single-objective one. At the time this work is written, only very few approaches are known to solve the TSPP from a bi-objective point of view. One of these is due to Keller [107],

and Keller and Goodchild [109], who call it the *multi-objective vending problem*: it consists of sequentially solving single-criterion versions of the problem. Recently, Jozefowiez *et al.* [100] proposed a metaheuristic method to build up an approximate description of the efficient solution set and Bérubé *et al.* [21] developed an exact approach. For an overview on multi-objective routing problems we refer the reader to Ehrgott [57] and Jozefowiez *et al.* [101].

In general, the TSPP derives from three different kind of problems, in which the two objective functions are addressed in one of these ways:

1) two objectives are combined in a single one, so the aim becomes the search of a route that minimize the difference between total travel cost and the collected profit;

2) the profit maximization becomes the unique objective while the travel cost is stated as a constraint, so that the traveller has to accumulate the larger profit not exceeding a well defined value of cost $C_{\max}$;

3) the objective becomes the minimization of the accumulated cost, with the constraint that the profit must be at least a fixed quantity $P_{\min}$.

These problems appeared many times in the literature. The first one is known as *Profitable Tour Problem* and was introduced by Dell'Amico *et al.* [48]. In most cases, it is solved as an elementary shortest path problem between two copies of the depot. A recent study about this problem was made by Archetti *et al.* [2]; they proposed some metaheuristics based on the tabu search algorithm to solve it.

The second problem is known as *Orienteering Problem* (OP). Other names under which OP can be found in literature are *Selective TSL* (see Laporte and Martello [115]), or *Maximum Collection Problem* (see Katakoa and Morito [104]). Many resolutions approaches were developed in the last years for the OP. In particular, two exact approaches were studied: the first is a branch-and-bound approach, developed by Arnd *et al.* (see [3]), the second one is a branch-and-cut approach developed by Fischetti *et al.* [66]. The first Fully Polynomial Time Approximation Scheme (FPTAS) for OP was presented by Chen and Har-Peled in 2006 [31], while a genetic approach was developed by Tasgetiren [165], and by Tasgetiren and Smith [166].

The third problem is known as *Prize Collecting TSP* (PCTSP), and in its first definition due to Balas [11], penality terms for unvisited nodes are also added to the objective function. Most of the authors who have worked on this problem, however, have null penalty terms in their applications. An additive approach for the optimal solution of PCTSP was developed by Fischetti and Toth [67], while a Lagrangean heuristic is explored by Dell'Amico *et al.* [47], and a hybrid heuristic is considered by Chaves and Lorena [30].

In our TSPP definition we assume that the route must start and finish at the source. Several problems have similar features, except that all circuits are searched in the graph. While it would be easy to enforce the visit of any particular node, the opposite is not true: finding a circuit in a graph is a more general problem than finding a circuit linked to a given source. In particular, it is not possible to simply introduce a dummy node representing the source. In our definition and throughout this thesis, we restrict ourselves to problems with exactly one source.

This chapter is organized as follows: in the Section 3.1 we give a general formulation for the undirected TSPP. In the Section 3.2 we study the complexity of TSPP and in particular, we show that exists a corrispondence among the starting data and the cardinality of the efficient frontier. In the 3.3 we describe several applications of the TSPP in real-life

situations. Finally, in the Sections 3.4,3.5,3.6,3.7 we give an overview of the exact, heuristic and metaheurisitc approaches developed in the last years for TSPP.

3.1 Integer Linear Programming Formulation for TSPP

In this section we describe the Integer Linear Programming (ILP) formulation for the undirected TSPP. This formulation uses the notation introduced in the Graph Theory section (see Chapter 1).

Let $G = (V, E)$ a complete undirected graph, with $V = \{v_0, v_1, \ldots, v_n\}$ the set of nodes, and E the set of edges. Let p_i be the profit of node v_i, and c_e the cost of edge $e \in E$. To model the TSPP we use two different set of variables. The first one, associated to the edges is the following:

$$x_{ij} = \begin{cases} 1 & \text{if edge } (v_i, v_j) \text{ belongs to the solution,} \\ 0 & \text{otherwise.} \end{cases}$$

The second one is associated to nodes:

$$y_i = \begin{cases} 1 & \text{if node } v_i \text{ is visited,} \\ 0 & \text{otherwise.} \end{cases}$$

For an easier notation, when we want to emphasize the dependence of a variable x on a given, yet unspecified edge $e \in E$, we write x_e in place of x_{ij}, with the meaning that $x_e = x_{ij}$ where $e = (v_i, v_j) \in E$

We can formulate the TSPP as the 0–1 ILP problem as follows:

$$\max \sum_{v_i \in V} p_i y_i, \quad \min \sum_{e \in \delta(v_i)} c_e x_e \tag{3.1a}$$

$$\text{subject to} \quad x(\delta(v_i)) = 2y_i, \forall v_i \in V \tag{3.1b}$$

$$\sum_{e \in \delta(S)} x_e \geq 2y_i \quad \forall S \subseteq V \text{ with } \emptyset \neq S \neq V, v_0 \in S \text{ and } v_i \notin S \tag{3.1c}$$

$$x_e \in \{0, 1\} \ \forall e \in E \tag{3.1d}$$

$$y_i \in \{0, 1\} \ \forall i = \{0, 1, \ldots, n\}. \tag{3.1e}$$

The objective function consists in the maximization of the sum of collected profits (i.e. the sum of profits belonging to the solution), and simultaneously the minimization of the sum of accumulated routing cost, given by the sum of the cost associated to the edges belonging to the solution.

The constraints (3.1b) are called the *degree constraints*: they ensure that the degree of each node is 2 if node is visited, or 0 if node is not visited. This means that a node can be visited at most one time.

The constraints (3.1c), called the *subtour elimination constraints*, they impose that no subtours are allowed inside the solution. They can be written also in this way:

$$\sum_{e \in E(S)} x_e \leq \sum_{v_j \in S \setminus \{v_i\}} y_j, \forall S \subseteq V \quad \forall v_i \in S$$

These constraints ensure that each subset of visited nodes must be connected to the remaining ones by at least two edges, in order to avoid unconnected subcycles. It is easy to note that

the number of these inequalities is 2^n, that is exponential in the size of the problem. This becomes a difficulty in searching the solutions: so, some approaches have been developed to overcame the difficulty by separating the set of inequalities and by selecting only those necessary to the construction of the optimal route.

The constraints (3.1d) and (3.1e) impose bounds and integrality conditions on the variables.

3.2 Complexity of the TSPP

The TSPP complexity can be easily derived from the complexity of TSP, because we can transform an instance of the TSP in an instance of TSPP by simply adding large profits to every node of the graph. It follows that:

Lemma 3.1 *The Travelling Salesman Problem with Profits belongs to the class of NP-hard problems.*

We prove this result by using some literature references. As we already mentioned, the TSPP is related to some known problems: the Prize Collecting TSP (PCTSP), the Orienteering Problem (OP), and the Profitable Tour Problem (PTP). We can easily note that an algorithm which is able to compute all efficient TSPP solutions can be used to solve both OP and PCTSP and, with some little modifications, it can be used also for PTP. So, it follows that:

Lemma 3.2 *If either OP or PCTSP are NP-hard, then TSPP is NP-hard.*

Golden *et al.* [82] gave a formal proof about the NP-hardness for OP using a recognition version of the TSP. They consider an instance of the recognition TSP, then they assign a unit profit to each node, one of which is chosen as the source, and set the time limit to $C_{\max}$. So, they claim that solving the OP problem on this instance gives an answer to the following question:

Given a set of nodes, is there a tour of length less or equal than $C_{\max}$ through all the nodes?

If the total score is equal to the number of nodes, then the answer is yes, otherwise is no. A similar proof was proposed by Laporte and Martello in [115], using the Hamiltonian circuit problem.

In the literature we can find some studies about polynomially solvable instances. Balas [14] introduces ordering constraints for which the PCTSP becomes polynomially solvable. In the same way, Kabadi and Punnen [102] extend results on polynomially solvable cases of the TSP to the PCTSP.

3.2.1 Complexity of the efficient frontier

As explained in the previous sections, the cardinality of the efficient frontier for a multi-objective combinatorial optimization problem can be exponential. So, in this section we want to show, through an example, that the Pareto-optimal set for TSPP may have exponential size.

Let $G = (V, E)$ be a complete undirected graph on the node set $V = \{v_0, v_1, \ldots, v_n\}$. We may describe the edge set as $E = \{(v_i, v_j) : i, j \in \{0, 1, \ldots, n\}, i < j\}$.

Let us define the node profits and edge costs as follows:

- $p_i = 2^{i-1}$ for all $i = 0, 1, \ldots, n$;

- $c_{ij} = 2^{j-1}$ for all $(v_i, v_j) \in E$ with $i < j$.

The input size of the above instance is $O(n^2)$, whereas the size of the efficient frontier is $O(2^n)$. In order to better understand the problem, we can analyze its structure more deeply. In this specific instance, the cost and profit values are dependent on the node index they are associated to. In fact, the profit associated to the node v_i is exponential in the value of index i, while the cost associated to the edge (v_i, v_j) is determined by the largest index of the two connected nodes. To find solutions of this problem we must find all the couples *(profit,cost)* that satisfy the constraints. The efficient solutions are the set of non-dominated solutions, *i.e.*, pairs (p_i, c_i) for which:

$$\big((p_j \leq p_i) \wedge (c_j > c_i)\big) \vee \big((p_j < p_i) \wedge (c_j \geq c_i)\big) \quad \forall j = 1, \ldots, n, \ \forall i < j \, .$$

We can search all solutions of this problem by building a list of possible profits values. A profit is determined by the specific combination of nodes that we decide to visit: we can then define vectors containing list of nodes, and in this way enumerate all possible combinations of nodes. We can assign at each vector a profit and a cost computed as the sum of profits and costs of the nodes that belong to the vector. If we decide not to visit any node, the corresponding vector will contain only the node v_0, with total profit $P_{\text{tot}} = 0$ and total cost $C_{\text{tot}} = 0$. Instead, if we decide to visit all nodes, the total profit will be the sum of all profits:

$$C_{\text{tot}} = \sum_{i=0}^{n} 2^{i-1} = 2^n - 1$$

If we represent all the existing subsets of nodes with row vectors containing the list of nodes in lexicographic order, we can see that at the kth vector it corresponds profit k and cost k. We can observe this behaviour in the specific case of a graph with 4 nodes:

Vector of visited nodes	P_{tot}	C_{tot}
(v_0)	0	0
(v_0, v_1)	1	1
(v_0, v_2)	2	2
(v_0, v_1, v_2)	3	3
(v_0, v_3)	4	4
(v_0, v_1, v_3)	5	5
(v_0, v_2, v_3)	6	6
(v_0, v_1, v_2, v_3)	7	7
(v_0, v_4)	8	8
(v_0, v_1, v_4)	9	9
(v_0, v_2, v_4)	10	10
(v_0, v_1, v_2, v_4)	11	11
(v_0, v_3, v_4)	12	12
(v_0, v_1, v_3, v_4)	13	13
(v_0, v_2, v_3, v_4)	14	14
$(v_0, v_1, v_2, v_3, v_4)$	15	15

In this case, all feasible couples (p_i, c_i) belong to the Pareto-efficient frontier, because each one is not dominated by any other. So, the size of the efficient frontier is maximal: $2^4 = 16$. This example can help to deduce the following lemma:

Lemma 3.3 *The efficient frontier of the TSPP has polynomial size if the number t of distinct profits is fixed.*

To better understand this concept we can proceed as follows. We divide the nodes in subsets $S_1, \ldots, S_t$, each one containing nodes that have the same profit. In the simple case of 2 possible profit values p_1 and p_2 (*i.e.* $t = 2$), we have only two sets: S_1 and S_2, where S_1 contains nodes with profit p_1 and S_2 contains nodes with profit p_2. If the graph contains n nodes, we can suppose that k nodes belong to the set S_1 and $n - k$ nodes belong to the set S_2. The efficient frontier of the Pareto set will contain, at most, the maximum number of different total profits. In fact, as we have seen in the previous example, the cardinality of the set of non-dominated efficient solutions can contain, at most, all the feasible solutions of the problem, in the hypotesis that no solution dominates any other. In the case just described of an instance with only 2 profit values, we have at most $(k + 1)$ possible choices in the set S_1 and $(n - k + 1)$ choices for S_2. Hence the number of solution in the Pareto-efficient frontier is smaller or equal than:

$$\max_{k \in \{1,\ldots,n\}} (k + 1)(n - k + 1) \tag{3.2}$$

The bound is tight for $k = n/2$. Then, complexity in the case $t = 2$ is $O(n^2/2)$. We can generalize the reasoning to the case of t possible profit values, where there are t different sets: $S_1, \ldots, S_t$, each one containing k_i nodes with

$$\sum_{i=1}^{t} k_i = n.$$

In this general case, the maximum cardinality of the Pareto-efficient frontier bounded by:

$$\max_{k_1,\ldots,k_t \in \{1,\ldots,n\}} (k_1 + 1)(k_2 + 1) \cdots (k_t + 1) \tag{3.3}$$

Given that the right-hand side of (3.3) reaches its maximum for $k_i = n/t$, $i = 1, \ldots, t$, in the worst case the size of the efficient frontier is $O(n^t/t)$.

3.3 Applications

The TSPP can be applied to many situations in real-life problems. In this section we describe several applications in the scheduling context, in the orienteering events and in the vehicle routing problems.

3.3.1 Scheduling Problems

A typical TSPP application can be found in the scheduling context. Consider a simple scheduling application, in which there are n jobs to be processed sequentially on a machine. Let c_{ij} be the set up cost required for processing job j immediately after job i. To each job is assigned a profit p_i, that describe the successful completion of the job. In these type of situations some constraints can enforce a specific selection of the job to be processed. We can find this type of application for instance in the steel context in the work of Gensch [74], or in chemical firm context in the work of Pekny *et al.* [134].

Another well known application of TSPP described in the work of Balas and Martin [15], it is the scheduling of daily operations of a steel rolling mill. This type of context gives rise

to complex production scheduling problems. A steel hot rolling mill subjects steel slabs to high temperatures and pressures to form steel coils. Let C_{ij} be the cost of processing order j just after order i, and p_i the weight of a slab assigned to order i. From an inventory, schedulers have to choose a set of slabs and order them to minimize the global cost of the sequence. This commercial system is currently in use in several steel mills worldwide. It gives rise to a PCTSP with penalty terms in the objective function. A recent contribution in this area has been given by Cowling [42]. It describes the model and heuristics used to solve the PCTSP, and considers the impact on existing planning and production systems and the quality improvements resulting from the system's implementation.

Useful general references on machine scheduling problems include the books by Conway *et al.* [39], Backer [9, 10]), Coffman [37], Blazewicz *et al.* [23], Brucker [24] and Pinedo Graham [138]. For a full review on TSP-based approaches to scheduling problems see Bagchi *et al.* [8].

3.3.2 Orienteering events

Another famous application of TSPP are *orienteering events*, introduced by Tsiligirides [168] in 1984. Orienteering is a sport that takes place in mountainous or forested areas. Each competitor starts from a control point, and is provided with a topographic map marked with a course consisting of a series of terrain or man-made features to be visited. The competitors on each course are started individually at two-minute intervals and navigate through a series of checkpoints until a finish line. They are not forced to visit intermediate checkpoints, but if they decide to visit some of them, they accumulate score. The winner is the competitor ending with the shortest elapsed time and the maximum score. Solutions of this problem are obtained by solutions of the OP. The aim to maximize the score can be seen as the second objective function in the model formulation.

3.3.3 Vehicle Routing Problem with an inventory component

Golden *et al.* [81] apply the same modeling to a vehicle routing problem with an inventory component. The *Inventory Routing Problem* (IRP) involves the integration and coordination of two components of the logistics value chain: inventory management and vehicle routing.

The IRP is concerned with the distribution of a single product from a single facility to a set of n customers over a given planning horizon of length T, possibly infinite. Customers consume the product at a rate u_i and can maintain an inventory of the product up to a level C_i. A fleet of m homogeneous vehicles of capacity Q are available for the distribution of the product. A first step of the solution procedure is to determine which customers to serve each day. The objective is to minimize the distribution costs during the planning period. This can be done through the resolution of an OP. Many different industries are currently involved with this kind of problems: some of them are from the chemical area, other work on electronic assembly, or on metal fabrication, or in the aerospace field. Works on this topic are due to Campbell *et al.* [26] and to Houssaine *et al.* [60].

3.3.4 Vehicle-routing cost allocation and other problems

TSPP appears also as a subproblem in solution procedures devoted to different kinds of problems. For example, we can find it in the vehicle-routing cost allocation problems. In these types of problems the aim is to find a good cost-allocation method, *i.e.*, a method that

according to specified criteria allocates the cost of an optimal route configuration among customers. Cost-allocation methods can be based on different concepts concerning cooperative game theory, such as the core and the nucleolus (see Engevall *et al.* [62] and Gothe-Lundgren et al.[84]).

Noon *et al.* [130] propose an heuristic procedure for the resolution of the vehicle routing problem based on an iterative solution of TSPP.

Helmberg [91] faces a problem that he calls the *m-asymmetric TSP*. This problem arises from the necessity to study a scheduling problem on m non-identical machines with sequence dependent setup times. It consists in an asymmetric TSP in which there are m different salesmen. The travel cost of each salesman is different from the cost of the others, thus the costs are distinct. A TSPP can be derived from the one-machines subproblem that comes out from it.

3.4 Exact Solution Approaches

The aim of this section is to give an overview about the exact solution approaches that were developed in the last years for TSPP. Most of them are related with branch-and-bound procedures, often taken from TSP solution approaches. The main difference between resolution methods for TSP and TSPP resides in the bounding schemes, that we describe in the rest of this section.

3.4.1 The Assignment Problem relaxation

A first way to solve TSPP consists in the relaxation of some constraints to make the solution search easier. A typical way to relax a TSP problem consists in dropping subtour elimination constraints. Then, the remaining constraints define the well-known *Linear Assignment Problem*, that can be easily solved by one of the existing algorithm [25]. In many resolution approaches for TSPP the relaxation of the subtour elimination constraints was used for the simplicity of the approach. To this purpose, the constraints are re-written with a trick to eliminate the variables y_i: the variable x_{ii} is used to replace y_i in describing the nonvisited nodes, that is:

$$x_{ii} = 1 - y_i.$$

Then, $x_{ii} = 1$ if node i is left unrouted, and $x_{ii} = 0$ otherwise. With these definitions, a valid formulation of the TSPP constraints is:

$$\sum_{v_j \in V} x_{ij} = 1, v_i \in V$$
$$\sum_{v_i \in V} x_{ij} = 1, v_j \in V$$
$$x_{11} = 1$$
$$x_{ij} \in \{0, 1\} \text{ for all } i, j \in \{1, \ldots, n\}$$

Many authors changed the model using this substitution: see for example Gensh [74], and Fischetti and Toth [67]. So, every feasible solution gives a family of disjoint subtours, each one visiting the source and covering all nodes. The difficulty that arises in some TSPP solution approaches consists in the presence of a constraint involving some resource, that

can be the profit or the cost. This constraint usually belongs to the generalized covering constraints class, as in the case of PCTSP problem

$$\sum_{v_i \in V} p_i x_{ii} > P,$$

or in the knapsack constraint class, as in the case of OP:

$$\sum_{v_i, v_j \in V} c_{ij} x_{ij} < C$$

This type of constraints modifies the structure of relaxation, and the problem becomes a *linear constraint assignment problem*. To solve it, many resolution approaches were developed, the most famous being the following:

- using inequality to strengthen the lower bound. For example, Fischetti, Salazar, and Toth [66] developed an algorithm to find the exact solutions of the OP based on several families of valid inequalities. Similar approaches were developed by Leifer and Rosenwein in [119] and Gendreau *et al.* in [71];
- using Lagrangean relaxation. This kind of approach was developed by Gensch [74] in the study of an industrial application of the TSP. In this problem, a time constraint restricts the salesman to visit a small subset of his total customers for each planning period. Hence, the solution requires the simultaneous selection of the proper subset and the identification of the calling order. The problem is formulated as a maximization of the expected sales subject to the time constraint; when the Lagrangean relaxation is used the problem is solved by a bisection method, and gaps are taken into account by taking advantage of the underlying transportation structure of the problem, thus providing a tight upper bound for the optimizing branch and bound routine in the algorithm. The operator theory developed by Srinivasan and Thomson [164] is used to catch the optimal integer solution near the end of the procedure;
- using a relaxation of the integrality constraints, so that the problem becomes easier to solve with standard algorithms. Anyway, there exist approaches, like that by Fischetti and Toth [67], where the lower bound is computed by using a Lagrangean relaxation of the resource constraints. They use the addition of bounding procedures yielding sequences of increasing lower bounds. These bounds are computed by relaxing in a Lagrangean fashion, that is by embedding constraint on the resource in the objective function, and by solving the corresponding Lagrangean dual problem. Finally, it was developed a special algorithm to solve the resulting assignment problem.

The bounding procedure just described allows to find an efficient lower bound for the *Asymmetric Travelling Salesman Problem with Profits*. Obviously, the complexity of these procedures for the computation of bounds leads to an increment of computation time. This is an advantageous when the number of nodes in the graph is very large, and when the graph is asymmetric. In case of a small number of nodes or of a symmetric instance, many efficient and very fast algorithms are available in the literature, so that the computation of a bound becomes useless. Actually, it is possible to solve OP symmetric instances with up to 300 nodes in less than 3 hours, and instances with up to 500 nodes in a few more hours. In the asymmetric case, instances that actually can be solved are of moderate size. Moreover, the computation time is generally longer when the problems of selection and ordering of the nodes are non-trivial.

3.4.2 Shortest Spanning 1-Tree relaxation

The relaxation method just described, based on the assignment problem, works in symmetric and in asymmetric cases, even if in the former case it has not very good performances. A different type of relaxation, developed by Fischetti and Toth in the PCTSP context [67], is obtained excluding the following constraints:

$$\sum_{v_j \in V \setminus \{v_i\}} x_{ij} = y_i \quad \forall v_i \in V$$

It states that each node must have a successor and removing it gives better results in terms of computational time.

A dummy node v_0 is introduced, and each node of the graph is connected to this node, $V' = V \cup \{v_0\}$. To eliminate the variables y_i, the variables $x_{0,i}$ are set to $1 - y_i$, so that $x_{0,i}$ holds 1 if node i is left unrounted, and $x_{0,i} = 0$ otherwise. In addition the costs of edges entering in the node v_0 are set to 0, while $c_{0,0} = c_{1,0} = \infty$. So, the TSPP constraints become:

$$\sum_{v_i \in V \cup \{v_0\}} x_{ij} = 1, \text{ for all } v_j \in V'$$

subtour elimination constraints

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in V'.$$

This formulation gives the *Shortest Spanning 1-Arborescence Problem* (1-SSAP). Every solution of this problem is a collection of $|V'|-1$ edges defining a directed spanning tree rooted at node v_0, plus an arc entering the source v_0. This relaxation can be used to calculate a lower bound for the TSPP. However, Fischetti and Toth point out that it would not be effective, because by removing the artificial node v_0 it could produce a solution given by a family of disconnected branchings. This impossibility to get from node v_0 to nodes with profit greater than 1 could result in a poor lower bound. So, they propose a relaxation in a Lagrangean fashion to obtain a better lower bound.

Kataoka *et al.* [105] proposed a similar approach to solve the OP. They used the minimum directed 1-subtree problem as a new relaxation of the OP and developed two methods to improve its lower bound: a cut and dual simplex method and a Lagrangean relaxation method. Then, they constructed an algorithm that combines these two methods in an appropriate way. In particular, they added for each arc (v_i, v_j) the following constraints:

$$x_{ij} + x_{0i} \leq 1$$

With these, every node descendent from v_0 that is not visited can not have a descendent node, which strongly enhances the quality of relaxation. The resulting problem is NP-hard, and it is not possible to compute effective lower bound for it. Anyway, it is proved that this kind of relaxation is superior to the ordinary assignment one.

In any case, relaxation based on the 1-arborescence problem does not enable to solve instances as large as those that branch-and-cut methods can solve.

Another approach is described by Dell'Amico *et al.* [48] to solve the PTP problem. They first transform the problem in an asymmetric TSP and then apply a classical bounding scheme to the resulting problem. Unfortunately, the computed bound is not very effective.

3.4.3 Lagrangean Decomposition Approach

The *Lagrangean Decomposition approach* allows to obtain a bound when resource constraints are present.

Haouari *et al.* [89] presented a new approach of this kind to calculate a lower bound. The lower bound is derived by formulating the problem as a minimum spanning tree problem with additional packing and knapsack constraints. The Lagrangean decomposition is then used to construct a Lagrangean dual problem that is solved by using the volume algorithm. This newly developed algorithm was found to consistently outperform the well-known subgradient algorithm. The upper bound is based on an enhanced genetic approach. The main feature is that it uses both primal and dual informations produced during the lower bound computation.

Gothe-Lundgen *et al.* [85] propose a similar approach: they duplicate the resource constraint and insert it in the subtour elimination constraints. Furthermore, they duplicate the variables y_i into new variables z_i, and introduce matching constraints $y_i = z_i$, for every $v_i \in V$ relaxed in the Lagrangean fashion. To compute the Lagrangean multipliers two different problems are used: an assignment problem and a knapsack problem. The difference between the optimal solutions of these problems gives the lower bound searched for the original problem. Computational results show that this method performs well especially for symmetric instances. However, the computing time is rather long and the duality gap is still too large to consider it suitable for solving large instances.

3.4.4 Two phases method

The two phases method is a general resolution scheme for multi-objective Combinatorial Optimization problem (see Ulungu and Teghem [172]), and it has been used, until now, to solve bi-objective problems. The main idea is to use efficient algorithms for the single objective problem related to the starting one and compute efficient solutions of this. As efficient algorithms for single objective problems are specific for them, it is necessary to preserve the constraint structure of the problem throughout the solution procedure.

In the first phase, supported efficient solutions are computed by using the statement given by Geoffrion's theorem [75].

In the second phase, information from the first phase are used to reduce the space in which the non-supported efficient solutions must be searched. In the implementation, the second phase is generally enumerative.

In particular, in the bi-objective case, the second phase explores the triangle defined by two consecutive supported efficient solutions in the objective space (Fig. 3.1). The search of new efficient points inside the triangle can be done in several ways, and depends on the problem. This kind of procedure was applied to a lot of bi-objective problems, for example: bi-objective assignment problems (see Przybylski, Gandibleux, Ehrgott in [141]), network flow problems (see Lee and Pulat in [117] and Sedeño-Noda and González-Martín in [158]), knpasack problems (see Ulungu in [169] and Visee *et al.* in [175]), and spanning tree problems (Ramos *et al.* in [145]).

3.4.5 The knapsack bound for the OP

Laporte and Martello [115] proposed a method to find a bound for the OP. The bound is computed as the solution of the corresponding knapsack problem, with:

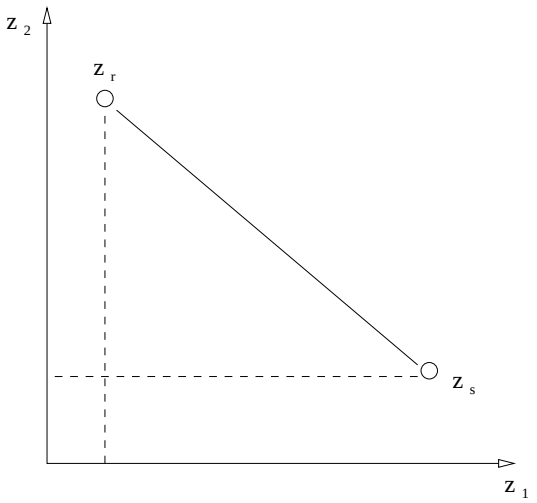

Figure 3.1: the triangle defined by solutions z_r and z_s gives the space where to search non-supported efficient solutions

- a similar objective function

$$\sum_{v_i \in V} p_i y_i$$

- the OP constraint

$$\sum_{(v_i,v_j)\in A} c_{ij} x_{ij} \leq C_{\max}$$

  that can be seen as the capacity constraint of the knapsack.

This bound is valid in both directed and undirected cases. To compute it, weights w_j are defined for each node $v_j \in V$ in this way:

$$w_j = \alpha \min_{i \neq j} c_{i,j} + (1-\alpha) \min_{k \neq j} c_{j,k}$$

with $0 \leq \alpha \leq 1$. Hence, the length of a route is greater or equal than the sum of the weights of its nodes. Thus, the optimal solution of a 0-1 knapsack problem, applied to all nodes except v_0, that collects maximum profit with cost at least $C_{\max} - c_1$, is an upper bound for the OP. This method was proved to be very efficient, it solved instances of up to 90 nodes in less than 100 seconds. However, it must be noticed that in the solved instances, the knapsack constraint is very tight. This procedure could be applied to the TSPP, as long as a resource constraint strongly limits the maximal number of visited nodes.

3.4.6 ϵ-constraint method

In the scalarization methods context, we can find an approach based on the *ϵ-constraint method*, efficiently used to solve the TSPP.

Here, one objective function is retained as a scalar-valued objective while all the other objective functions generate new constraints. The k-th ϵ-constraint problem is formulated

as:

$$\min\ f_k(x) \tag{3.4a}$$
$$\text{subject to } f_i(x) \leq \epsilon_i, i = 1, \ldots, p,\ i \neq k \tag{3.4b}$$
$$x \in X \tag{3.4c}$$

Hence, upper bounds of these constraints are given by the ϵ-vector and varying it, the exact Pareto frontier can theoretically be generated. In practice, this method has mostly be integrated with heuristic and interactive approaches, because the number of subproblems to solve is very large and it is very difficult to find an efficient variation scheme for the ϵ-vector.

Bérubé *et al.* [21] developed an approach that generates the Pareto frontier for TSPP by solving ϵ-constraint problems. Their idea was to construct a sequence of ϵ-constraint problems based on a progressive reduction of ϵ_j. They compute the ideal point and the Nadir point that define lower and upper bounds on the value of the efficient solutions, and then solved a sequence of problems through branch-and-cut, adding the optimal solution returned to the set of feasible solutions F.

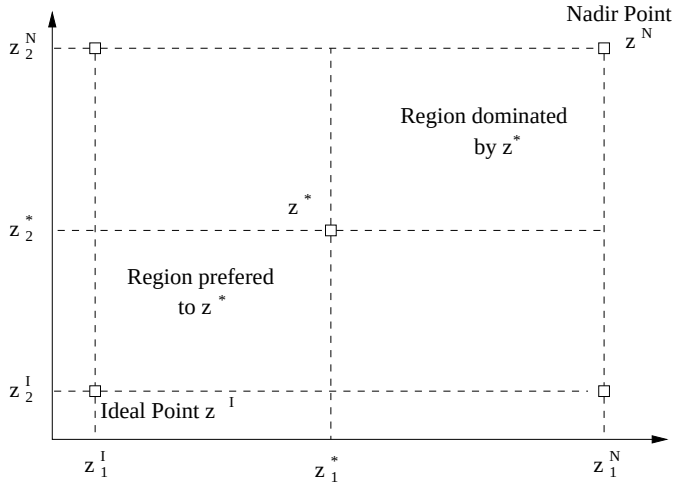

Figure 3.2: Dominance relation in the objective space

Throughout the algorithm, ϵ_j is decreased by a constant value Δ at each iteration. The algorithm solves problems of 150 nodes for easy instances, and up to about 100 nodes for harder instances.

3.5 Classical Heuristic Procedures

In this section we describe the main approximation procedures to solve the TSPP. Then, we explain the basic principles of heuristic approaches, and give some examples.

3.5.1 Approximation algorithm with a performance guarantee

Awerbuch *et al.* [6] gave the first approximation algorithm with poly-logarithmic performance guarantee for PCTSP and OP problems. The approach derives from an approximation algorithm from the k-minimum spanning tree problem (k-MST). The problem consists in

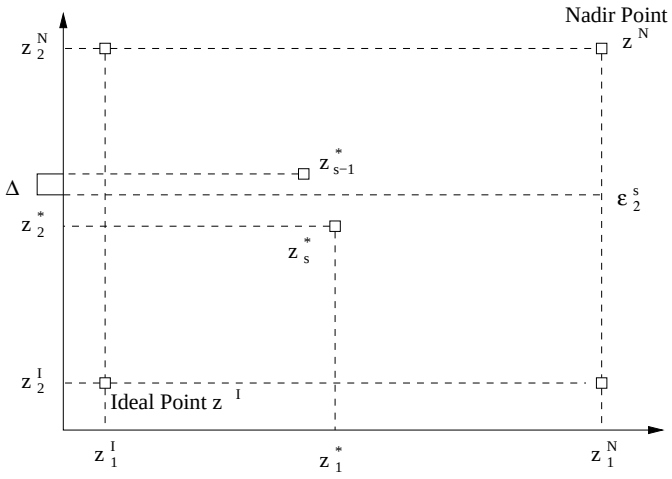

Figure 3.3: Illustration of two consecutive points computed by the algorithm

finding a tree of least weight that spans $k \leq n$ nodes in a undirected n-graph with non-negative edge weights. Each node $v_i \in V$ is replaced with p_i copies of itself at the same location, and the solution of k-MST on this graph with $k = p_{\min}$ is computed. In other words they describe an algorithm that achieves an approximation ratio $O(log^2(k))$ from the k-MST problem on general graphs, improving previous results for the case of points in the Euclidean space. Then, they extend the results to PCTSP by concatenating the tour found by their algorithm to a 2-approximation algorithm by Goemans and Williamson [78], achieving a factor of $O(log^2(\min(n, p_{\min})))$.

Bienstock *et al.* [22] proposed another approximation algorithm with a factor of 2.5 performance guarantee based on Cristofides algorithm to solve the PCTSP. The heuristic is based on three steps. In the first step they solve the Linear Programming relaxation of the whole problem for each node in the graph using the ellipsoid method. Then, they transform the obtained solution into a feasible solution of PCTSP, and construct $|V|$ feasible solutions to it, each corresponding to a different relaxed problem. Finally, they choose the best of the feasible solutions obtained, whichever yields the better cost. The algorithm was called *Modified LP relaxation heuristic*, and its worst case performance is 2.5.

3.5.2 Main principles in Heuristic Procedures

The most important difference between the standard TSP problem and the TSPP resides in the fact that in the TSPP it is not necessary to visit all the nodes. So, routes limited to the source as well as routes visiting all nodes are allowed. These two extreme solutions are optimal for both objective functions but, during the search of other solutions, we can notice that a good value for one of the objectives can yield to a bad value for the other. The aim of heuristic procedures is to find a balance between the objectives. Then, to modify an existing solution by transforming the associated route, four operation can be used:

- adding a node to the route;
- delete a node from the route;
- resequencing the route;
- replacing a node of the route with a node outside the route.

An important question is how to manage these four operations to improve the quality of solutions. In fact, the use of these procedures generally yields to an improvement of one objective and a worsening of the other ones. This point is analyzed in the following subsections, where we describe each of the four previous operations.

3.5.3 Adding a node to the route

Adding a node to a route yields obviously to an increment of both the profit and the cost, supposing that costs and profits values are positive. The increment of the profit corresponds to the profit of the added node p_i. The best node that can be added to a route corresponds to a node v_k for which the added cost to visit v_k instead of its current successor v_s, is minimum. The standard criterion for choosing the node to add consists in the selection of the node that maximize the ratio between the added profit and the added cost.

Golden *et al.* [82] proposed other criteria to decide the node to add to the route, such as the distance from the center of gravity, or the distance from destination. The aim of these methods is to attract the route toward nodes that seem promising for future modifications. Other ideas are described in Ramesh and Brown [143].

3.5.4 Delete a node from the route

The deletion of a node yields to a decrement of both the profit and the cost. The way to select the node to remove is the same explained for the adding operation: the standard procedure searches the node that minimize tha ratio between the lost profit and the lost cost. Ramesh and Brown [143] propose the deletion of every node followed by some 2-opt interchanges. The node deleted minimize the profit-to-savings ratio. The same approach can be applied to the adding procedure.

3.5.5 Resequencing the route

Starting from a given solution for a TSP problem, it may be possible to search a way to visit nodes in order to reduce the cost. This procedure is often used in routing problems, and aims to iteratively improve the result. Procedures built for TSP, and generally used for TSPP are 2-opt and 3-opt of Lin [29, 143], or the Lin-Kernigham method [92]. The 2-opt and 3-opt algorithms are special cases of the $\lambda - opt$ algorithm, where in each step λ edges of the current tour are replaced by λ edges in such a way that a shorter tour is obtained. In other words, in each step λ links are deleted, and a new shorter path is created reversing one or more of them.

Actually, attention focuses on these methods even if it is not so clear, how much looking for of a good solution with a series of resequencing procedures deteriorate performances.

3.5.6 Replacing a node

Another possibility to obtain better tours is to swap two nodes, more precisely to substitute one node that belongs to the current tour with a node outside the tour. In the context of TSPP, the procedure improves a solution if it causes to an increment of the profit and a decrement of the cost. To obtain always new non-dominated solutions, the simplest procedure can be the substitution of a node with one that has a larger profit. Keller [108] extends this scheme by allowing the deletion of two consecutive nodes, while Dell'Amico *et al.* [49] propose the deletion of a chain.

3.6 Heuristic procedures

The operations just described are often combined together in the classical heuristic methods. However, mixing these components may induce some difficulties, in particular, cycling. Some procedures that we present below iteratively apply the basic steps described above, with a descent strategy. The most famous heuristic is the *greedy procedure*, that consists in a sequence of insertion-deletion steps. The insertion step consists in iteratively adding nodes in a route until no more ones can be added (OP case), or the solution is feasible (PCTSP case), or no improvements are possible (PTP case). The insertion of a node can be combined with the swap of two or more nodes or the resequencing of the route, to reach better solution values. The *greedy procedure* can use also the deletion operation. In this case it starts from a TSP solution and searches, at each step, the best node to delete to reach a non-dominated solution.

Based on these approaches, Dell'Amico *et al.* [49], propose two heuristics for the PCTSP. They use some relaxation of the model to obtain a starting solution that can be improved with the procedures just described. In particular, they iteratively use extension and collapse operations, until no further improvements are possible. Extension applies insertion as long as insertions exceed a computed average ratio, while collapse replaces a chain with a single node.

Based on these heuristics, other approaches were developed to solve the TSPP. For example, the *Path-extension procedure*, that is very similar to the greedy procedure, extends a path until no more nodes can be added.

Tsiligirides [168] selects nodes to be randomly added in a probabilistic fashion. He also proposed an approach based on the *sweep procedure*. The geographic area is divided into sectors determined by two concetric circles and an edge of given length. Other approaches with the same computational time give clear inferior results.

Finally, in the *partitioning-based procedure* the nodes are partitioned in several set of feasible routes [29]. Then, nodes belonging to a route can be moved in an another route through local search procedures, so that the movement result in an increment to the largest profit among the sets. An important point is that the best route might change during the process.

3.7 Metaheuristic Procedures

Metaheuristic procedures provide solutions that avoid cycling problems. Several types of metaheuristic have been developed for the TSPP. In this section we describe the most frequently used.

3.7.1 Ejection chain local search method

Jozefowiez, Glover and Laguna [100] proposed a hybrid metaheuristic based on an ejection chain local search method combined with a multi-objective evolutionary algorithm to generate diversified starting solutions in the objective space for the TSPP. The first step of their algorithm is the definition of a neighborhood search process. To do that, two problems have to be solved:

- the resolution of many TSP on different set of nodes;
- the selection of different subset of nodes to visit.

These problems are faced by using two sets of neighborhood moves in an ejection chain process. To provide starting solutions for the ejection chain process, a multi-objective evolutionary algorithm has been developed. The combination of Multi Objective Evolutionary Algorithms (MOEA) and Ejection Chain (EC) processes gives a mean for both exploring and approaching the optimal Pareto set: in fact, while *EC* process can bring solutions towards the optimal Pareto set, *MOEA* can provide solutions in the objective space thanks to its population, and so it can diversify the search.

3.7.2 Tabu Search

Tabu Search (TS) is a local (neighborhood) search guided by a selection function which is used to evaluate candidate solutions (see [76] and [77]). See Section 2.3.2 for a general description of the TS.

Ramesh and Brown [143] propose an heuristic for the OP problem, that makes use of a tabu list to avoid that the algorithm cycles. Their procedure consists of four phases. At the beginning, the nodes are iteratively added to build a route, until no other insertions are possible. Then, for each pair of edges is carried out the 2-opt or the 3-opt improvement. If the total improvement has gone beyond another threshold, the algorithm returns to the first phase. Finally, there were attempts to achieve a decrease in the length of the path in such a way that one point is removed and another is inserted.

The stopping criterion is included in the phases. It is based on the improvement between two successive iterations and their total number. When the stopping criterion is met, then the final phase can begin. Due to the deletion phase, the algorithm may cycle, so routes are stored in a tabu list to avoid them in the following iterations.

Gendreau *et al.* [72] propose an approach consisting in the deletion of a part of the route or in the insertion of nodes cluster. A parameter, that indicates the importance of travel cost and profit of a route, is introduced to balance insertion and deletion. It is updated at each iteration, to favor the insertion or deletion procedures. This local search scheme is embedded in a tabu search approach to avoid cycling. When a node in removed, a tabu status for a randomly selected number of iterations is assigned. This procedures yields optimal or near optimal solutions, with computational times that never exceeds a couple of minutes. This is one of the most effective heuristic approaches.

3.7.3 Genetic algorithm

The Genetic Algorithm (GA) is an adaptive heuristic search method based on population genetics, developed by Holland [93]. See Section 2.3.2 for a brief description of GA.

Tasgetiren and Smith [165] developed a genetic algorithm to solve the OP. They consider a chromosome as a sequence of visited nodes. The offspring are generated by a classic order-based crossover operator. Mutations are obtained with procedures described in the previous sections. Non-feasible solutions are accepted with a penality, computed on the distance from feasibility. In Genetic Algorithms the search is carried on from a population of solutions. This is an advantage with respect to the heurstic approaches: in fact, they start from an initial solution and rely on a point-to-point improvement on it. The results provided by genetic algorithms are comparable to other metaheuristics, but with a longer computational time.

3.7.4 Deterministic annealing

The Deterministic annealing mathod [179] arose in Statistical Physic. It has been applied with varying degrees of success to a variety of image matching and labeling problems. Furthermore, the deterministic annealing has been applied to a variety of combinatorial optimization problems multiclass labelling based on winner-take-all (WTA) criterion, linear assignment, quadratic assignment (including the traveling salesman problem), graph matching and graph partitioning, clustering (central and pairwise), the Ising model etc. and to nonlinear optimization problems as well, with alternate success.

In the TSP context, Chao *et al.* [29] developed a procedure based on the record-to-record approach. The procedure is based on a partition of vertices. A nearest neighbor procedure is used to find the starting solution, and at each step a new node is added and a new route is computed until all graph nodes are covered. The procedure keeps the set of routes that contain the route with the highest score. At this point the record-to-record procedure is applied. Thus, the 2-opt operator is used on the current solution and tries to improve it. A deterioration of the solution is allowed, if it is not larger than a given percentage. After a fixed number of iterations, or when there is no improvements on the solution, the procedure stops and returns the best result obtained.

3.7.5 Neural Network approach

Neural Network are network of simple elements and of their gerarchic organization, interconnected in a parallel way. They interact with objects of the real world, in an analogous way that the biological neural system.

Wang et al. [176] apply the neural network method to solve the *OP* problem. The neural network consists of a matrix, whose rows correspond to nodes and columns correspond to a specific position in the path. The energy function is built in order to penalize columns with more than one activated node, solutions with fewer than n nodes and solutions exceeding the travel cost limit. The edge weights are not constant, each weight become the second partial derivative of the energy function with respect to the state. The method contains also a route-insertion procedure and a 2-opt improvement routine, that are very important for the success of the procedure.

This approach is very competitive with other solutions procedures for the resolution of *OP* problems.

Chapter 4

Computer Science applied to Medicine

In recent decades, the boundary line separating Mathematics and Computer Science from Medicine and Biology has been gradually whittled away, and new research areas were created, that use computer tools and mathematical models to study and facilitate the solution of a number of different problems and needs, that occur in the medical and the biological fields.

Early in the Eigthies the Medical Informatics appeared, a science that provides a wide range of knowledge and tools to improve many aspects of the medicians professional activity: among those, one the most relevant contributions are the computer-aided systems, that help for instance in planning optimal diagnostics and therapeutic decisions (see [38] and [174]). Even if the limits of this discipline are still indistinct, it is established that Medical Informatics comprises most of the technologically oriented fields (Information Technology , I.T., and Computer Science), but includes another vast field that studies the creation, the training, the management and the dissemination of information for medical purposes. The study and the understanding of this discipline strongly affected the way information is processed and converted into knowledge; this allowed in turn the creation of intelligent machines, programmed to mimic some human processes, and able to assist the decision-makers.

The information also plays a big role in the interpretation of data. The task of Medical Informatics is to fully evaluate how much the data are reliable, how the information is derived from the data, what kind of knowledge is needed to interpret the data and how best the knowledge, as well as the data, can be stored in computers.

The clinical decision support always includes the use of information to help the clinician to diagnose and/or treating a health problem of the patient, while a system for decision support (DSS), applied to the clinic can be defined as a system that consists of a knowledge database and an inference engine, and that is able to use the data to generate recommendations on specific cases.

It is worth to remember that a knowledge database refers to a body of knowledge systematically organized and stored in a computer to make decisions or solve problems. A DSS can also be defined as any software designed to directly or indirectly help to make clinical decisions in a situation where the characteristics of a particular patient are matched with a computerized database or knowledge base, in order to generate a specific assessment for that patient or to produce specific recommendations for the clinician's advice. It can also be defined as a system of information and planning with the ability to enquire computers

on an ad-hoc basis, analyze information and predict the impact of decisions before they are taken, or as a cohesive set of integrated programs that share data and information (not just a single application). These systems should be kept separate from decision support systems-operative, which are only data archives (albeit well-structured), mostly used for statistical, managerial, or financial purposes. Some patient data can be captured by diagnostic and monitoring tools, bypassing human errors in entering data. Using relational database with query strings, I.T. allows also a quicker, easier and targeter documents recovery. These technologies are increasingly liked to the clinician is practice.

4.1 Diagnostic Imaging

The terms "imaging", or "biomedical imaging", or "diagnostic imaging" [162, 137], refer to the general process by which it is possible to observe a non-visible area of an organism: examples of such a process are the ultrasound technique, spectroscopy technique, X-ray technique... The production of radiology images strongly increased with the introduction of new diagnostic equipments, this technological evolution has brought the research to the design of more interactive and easy to use visualization systems, that make it possible to simultaneously display data from different sources, with high processing power and transmission speed. In last-generation medical equipments, diagnostic images are automatically scanned by the machine, and the problem that now arises concerns the transfer of data through networks. These images are usually stored in an hospital-owned database, and can be locally used as comparison in the analysis of other similar clinical cases. Currently some tools have been developed to support medical decisions: they analyze medical data and provide automatic diagnosis by comparing the clinical data of the patient concerned with the clinical data of previously treated patients. It would be a great improvement if one could compare patient's data with those stored in the databases of other hospitals around the world. Moreover, it would also be worth to have a medical opinion from specialized medical teams of other hospitals. The usefulness of this type of support is evident in the case of very rare diseases, studied in only a few structures, for which to have some advices from people who really have treated similar cases can be of vital importance. The main problem in this type of operation resides in the time needed to send a request and receive an answer. Indeed, the size of diagnostic images can be very large, and therefore the identification of a preferential path through the network that minimizes the time that the packet data need to get to the destination, could be very important. For this reason, the development of a software that decides a priori which channel is the best in terms of time optimization, could contribute to improving the performance of such a tool.

4.2 International network of hospitals

Let us suppose to represent the connection between the various hospitals in Western Europe through a graph. Each node in the graph represents the main hospital in each European Union capital, while the different edges represent data network connecting the nodes (see Figure 4.1).

Let us suppose to send a packet containing data related to diagnostic images and clinical data of a patient through this network. Each hospital can be considered as the source of the graph, and all other nodes as possible destinations of the data. While sending files of small size can be very fast, sending high resolution images, or data of considerable size, may take

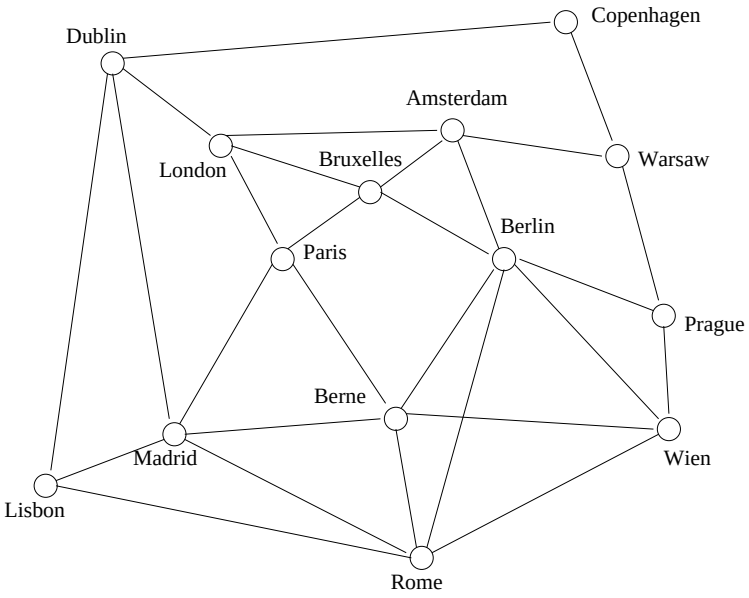

Figure 4.1: Hospital connections in the Western Europe

a longer time, that depends on various parameters such as the network speed, the average number of applications asking for the network, etc. This does not happen for the send operation only: indeed, we must consider also the fact that the request for advice or for a medical opinion in a wide structure may require more time than a search in a small hospital. Let us consider for instance the case where a computerized comparison is required between patient data and similar data contained in the databases of other hospitals: we can reasonably expect that the diagnosis made by comparing against a database containing a large number of tests will produce a substantially more accurate output, but in a considerably larger time. Hence, we have conflicting goals: on one side obtaining a very accurate medical advice or computerized diagnosis and, on the other side, obtaining a diagnosis as quickly as possible. That is to say: accuracy as against to speed.

We can use these considerations to associate profits to each node and costs to each edge of the graph. For a given disease, the profit can be one of the following parameters:

- the number of known cases in the hospital database;
- the number of cases that the hospital treats every day, on average;
- the number of specialized personnel available to the structure;
- the percentage of cases successfully resolved in the hospital;
- ...

One could also consider as the profit a combination of the previous, depending on the type of the required consultation. Similarly, the cost of each edge joining two hospitals can be for instance:

- the time needed to send a packet of given size through that edge;
- the time needed to complete the consultation procedure and send back the answer;

- the average number of packets that pass through the edge in a time unit;
- the cost for a consultancy from a highly ranked specialist;

It is easy to recognize the analogy between this problem and the Travelling Salesman Problem with Profit. The "traveler" is the medical information to send to one or more hospitals (not necessarily all), by maximizing profits and minimizing costs. So, to solve this TSPP is equivalent to find the path through which to send the medical data to obtain best answers in a reasonable time.

In this thesis we analyzed the TSPP from multiple points of view. The main objective has been to identify information that help in the resolution of the problem. We therefore sought approaches that compute the whole set of solutions, we studied the various metrics that make the problem easy to solve, and we studied some extensions, such as time windows, immediately identifiable in real-life cases. The latter are meaningful for medical experts (as well as in many other office-related situations), where a timetable is set for consultations, outside of which it is not possible to send or receive any type of communication.

The set of exact TSPP solutions gives a series of pairs (profit, cost), represented by a diagram, which describe all possible ways to gain a specific value of profit with a minimal cost. In this way, each solution represents the best way to send information in a subset of structures, with the smaller possible cost. If we find all TSPP solutions on the graph obtained from the initial data of the problem just described, we obtain one scheme of all possible ways to optimally send information. Thus, a hospital that needs a quick but not too detailed consultation, can choose the solutions with a cost suitable to its needs, while a clinical case on a very rare disease, for which it is necessary to obtain all possible information from all the possible structures, can choose the solution that correspond to the maximum profit.

4.3 An example

Let us consider the network of hospitals described in the Fig.4.1. For simplicity we suppose that only 3 different types of department are present in each hospital: Oncology, Cardiology, Neurology. For the profit function we choose to consider the percentage of cases successfully resolved in each hospital, that are documented in each structure's database (see Table 4.1).

The considered data are entirely fictitious and do not correspond to any real situation. A profit value of 0, means that the hospital does not have such a department and therefore no cases have been studied. Ficticious cost values for the edges are shown in Table 4.2.

We can assume that the cost cannot change quickly, and therefore it is possible to update them once a year without loss of information. We cannot make the same consideration for values associated with profits: in fact, in any hospital, new cases can appear, and therefore any feasible value associated to the cases treated by the structure often changes: for this reason, one could think to update the profit value of each hospital once a month. Each time the data on costs and profits are modified, it is necessary to solve a new TSPP for each department of every hospital. In our example, we should solve a TSPP by considering as the profits the values associated with the Department of Oncology, another TSPP by considering as the profits the Cardiology scores, and a final one by considering as the profits the Neurology scores. In this way, we find a set of solutions for each one of the three instances. Each hospital will solve these three problems by considering itself as the source node and all the others as possible destinations. As an example, if we consider Lisbon as the source

	Oncology	Cardiology	Neurology
Lisbon	70	50	50
Madrid	60	70	50
Dublin	80	65	0
Copenhagen	60	80	40
Warsaw	0	50	90
Prague	85	70	0
Wien	70	80	50
Rome	60	90	70
Berne	0	80	80
Paris	80	70	80
London	70	60	60
Bruxelles	80	70	0
Amsterdam	0	80	80
Berlin	70	0	80

Table 4.1: Feasible Profits for West Europe Hospital

hospital *i.e.*, the node from which the request is sent, all feasible ways to obtain information are described in the graphics 4.2, 4.3 and 4.4.

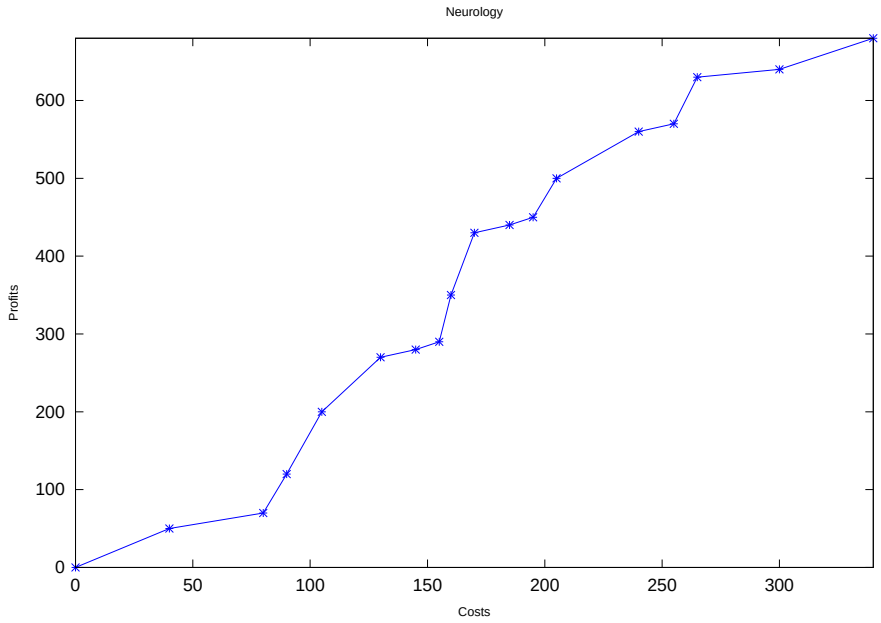

Figure 4.2: Efficient frontier for Neurology values

Hence we can see that, if the Neurology department needs a medical counceling of profit at least 500, the minimum implied cost is 205. Analogously, if the department of Oncology requests a case history of at least 600, the minimum cost is 260.

Thus, once the efficient set is computed, it is immediate to receive informations about the best ways to have a specific medical counceling of a given level from foreign departments.

	Lisbon	Madrid	Dublin	Copenhagen	Warsaw	Prague	Wien	Rome	Berne	Paris	London	Bruxelles	Amsterdam	Berlin
Lisbon	-	20	30	-	-	-	-	40	-	-	-	-	-	-
Madrid	20	-	50	-	-	-	-	30	20	30	-	-	-	-
Dublin	30	50	-	40	-	-	-	-	-	-	20	-	-	-
Copenhagen	-	-	40	-	10	-	-	-	-	-	-	-	-	-
Warsaw	-	-	-	10	-	30	-	-	-	-	-	-	20	-
Prague	-	-	-	-	30	-	15	-	-	-	-	-	-	40
Wien	-	-	-	-	-	15	-	65	60	-	-	-	-	50
Rome	40	30	-	-	-	-	65	-	25	-	-	-	-	70
Berne	-	20	-	-	-	-	60	25	-	30	-	-	-	30
Paris	-	30	-	-	-	-	-	-	30	-	10	10	-	-
London	-	-	20	-	-	-	-	-	-	10	-	20	30	-
Bruxelles	-	-	-	-	-	-	-	-	-	10	20	-	10	20
Amsterdam	-	-	-	-	20	-	-	-	-	-	30	10	-	20
Berlin	-	-	-	-	-	40	50	70	30	-	-	20	20	-

Table 4.2: Feasible Costs for West Europe Hospital

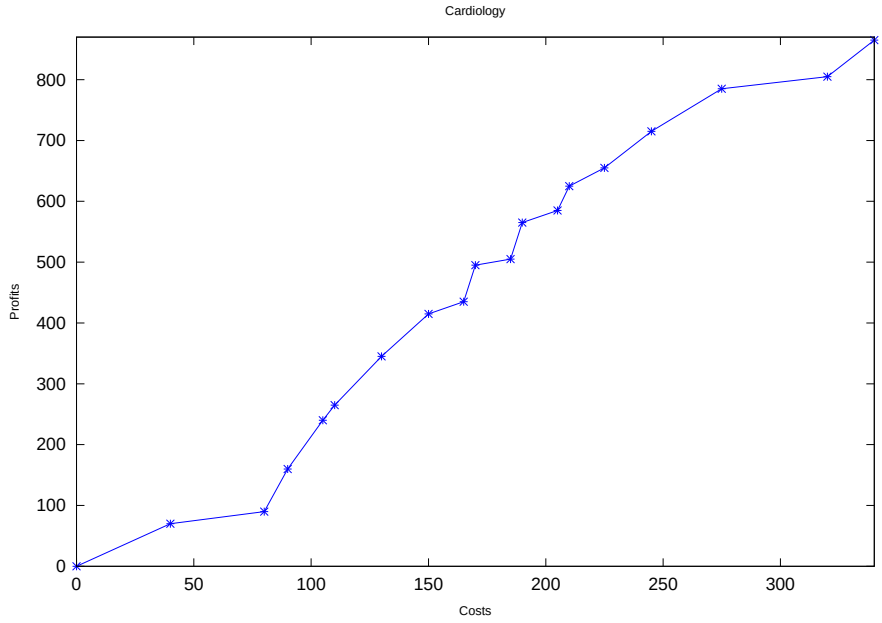

Figure 4.3: Efficient frontier for Cardiology values

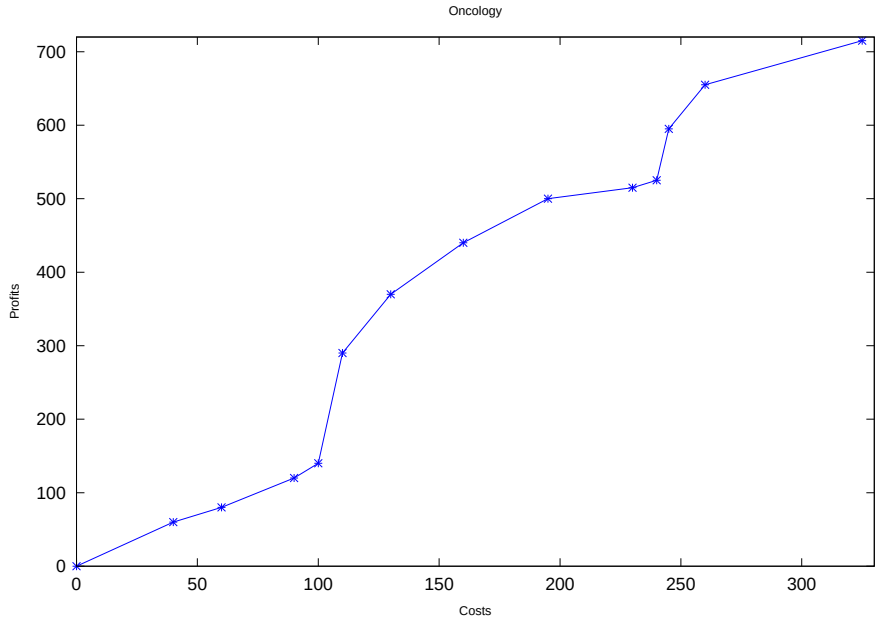

Figure 4.4: Efficient frontier for Oncology values

It is clear from this artificial example that real situations can be made extremely more complex, so that optimization tools like those studied in this thesis can provide meaningful improvements.

Chapter 5

The traveling salesman problem on trees: balancing profits and costs

In this chapter we study the TSP with Profits from a bi-objective point of view, focusing on the case where the underlying graph is a tree. It is clear that trees are fundamental network topologies, and many practical problems feature a tree as the underlying graph. More in particular, vehicle routing problems on trees have been discussed in, for instance, Labbé et al. [114], Averbakh and Berman [5], Muslea [128], Lim et al. [120], Chandran and Raghavan [28], and the references contained therein. Motivation for the tree topology comes usually from transportation contexts where the underlying network is a railway network (in pit mines, for instance), a river network, or a sparse road network (in rural areas). Karuno et al. [103] study the problem of scheduling and routing a vehicle, such as an automated guided vehicle, in a building with a simple structure of corridors (each floor corresponds to a subtree and each room to a leaf node). In the works mentioned above it is required to visit each location. In this work, we relax this requirement and assume that a given profit is incurred when a client is visited. Notice further that we deal here with a bi-objective problem featuring a single vehicle with unbounded capacity, *i.e.*, featuring a traveling salesman.

We notice that finding all efficient points is NP-hard. In a quite general context, approximating the set of efficient solutions (also known as the Pareto curve) has been dealt with by Papadimitriou and Yannakakis [132]. They assert that an FPTAS for constructing an approximate Pareto curve for a linear optimization problem A exists if there is a pseudopolynomial algorithm for the exact version of A. This immediately implies the existence of a so-called FPTAS to find an ϵ-approximate Pareto curve for our problem. In fact, after exploiting the analogy between our problem and a precedence constrained knapsack problem studied by Johnson and Niemi [99], we revise a pseudopolynomial dynamic programming approach proposed in [99] and we adapt it to our problem. Then, we develop a simple FPTAS for our bi-objective problem, using ideas from Erlebach et al. [63] for the multi-objective knapsack problem. For finding the set of extreme supported efficient points we propose a $O(n^2)$ algorithm, and we show that no algorithm can exist with complexity lower than $O\big(n\log(n)\big)$, where n is the number of nodes in the tree. Finding one supported efficient point, corresponding to a given combination of the two objectives, takes only linear time. Thus, we prove that, when the graph is a tree, computing the set of all efficient solutions is more difficult than computing the set of all extreme supported efficient solutions (assuming $P \neq NP$), which in turn is proven to be more difficult than computing a single supported efficient point.

The chapter is organized as follows: in Section 5.1 we define the three problems studied and we give some theoretical background on bi-objective optimization. In Section 5.2 we study the complexity of finding all efficient points (referred to as Problem 1) and we develop a FPTAS for this problem. Some polynomially solvable special cases are also analyzed. In Section 5.3, a polynomial time algorithm is developed for finding the set of extreme supported efficient points (Problem 2), thereby also settling complexity of finding only one supported efficient point (Problem 3). We finish with a conclusion and an overview of our results in Section 5.4.

5.1 The bicriteria TSP with profits on trees: three problems

Let $T = (V, E)$ be a tree where $V = \{0, 1, 2, \ldots, n\}$ is a set of $n + 1$ nodes and E is a set of edges. We consider node 0 (the depot) as the root. Let a profit p_i be associated with each node $i \in V$ and a cost c_{ij} be associated with each edge $(i, j) \in E$. We will assume that profits and costs are strictly positive, with the only exception that $p_0 = 0$. A *feasible subtour* $S = (V(S), E(S))$ of T is a circuit that starts and finishes at node 0, visits each node of $V(S) \subseteq V$ exactly once, and consists of the resulting edges $E(S) \subseteq E$. Notice that we distinguish between *visiting* a client and *passing* a client. A client is only visited when the server collects profit at that client and profit can only be collected once at every client. Feasible subtours are then identified by subtrees containing node 0, where every subtree T' corresponds to a family of equivalent subtours, characterized by the order in which its nodes are visited. In every subtour corresponding to T', each edge is traversed twice, so the cost of the subtour is twice the sum of the costs of the edges of T' and the profit is the sum of the node profits. Notice further that we assume a given position of the salesman; this is relevant since, in contrast to the ordinary TSP, one does not need to visit all nodes. However, all results we state also hold for the case where one is allowed to choose the salesman's position (at the expense of a factor n in the running times).

The cost of a feasible subtour S is then

$$c(S) = 2 \sum_{(i,j) \in E(S)} c_{ij},$$

whereas the profit of S is

$$p(S) = \sum_{i \in V(S)} p_i.$$

The goal is to find a feasible subtour that minimizes the total length of the tour and simultaneously maximizes the profit gained. We refer to Ehrgott [58] and Hoogeveen [94] for an introduction into multicriteria optimization. Here, we review basic terminology.

A feasible subtour S is *Pareto optimal* if there exists no feasible subtour S' such that $c(S') \leq c(S)$ and $p(S') \geq p(S)$, and at least one inequality is strict. A point $(\gamma, \pi) \in \mathbb{R}^2$ is an *efficient point* if there exists a Pareto optimal subtour S such that $c(S) = \gamma$ and $p(S) = \pi$. Let $\mathcal{E}$ denote the set of efficient points. In Fig. 5.1 every point represents a feasible solution associated with a subtour S. The bold dots correspond to efficient points.

An efficient point $(\gamma, \pi) \in \mathbb{R}^2$ is *supported* if there exists a scalar $\lambda \in [0, 1]$ and a feasible subtour S such that S maximizes $\lambda p(S) - (1 - \lambda)c(S)$ and $c(S) = \gamma$, $p(S) = \pi$. A supported efficient point that is an extreme point of the convex hull of all solution points, is called an *extreme* supported efficient point (Ehrgott [58]). Let $\mathcal{SE}$ denote the set of extreme supported efficient points (see Figure 5.2). Clearly, $\mathcal{SE} \subseteq \mathcal{E}$.

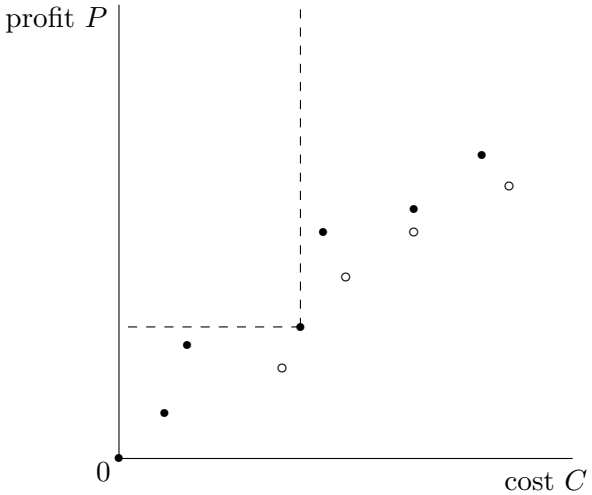

Figure 5.1: Pareto optimality.

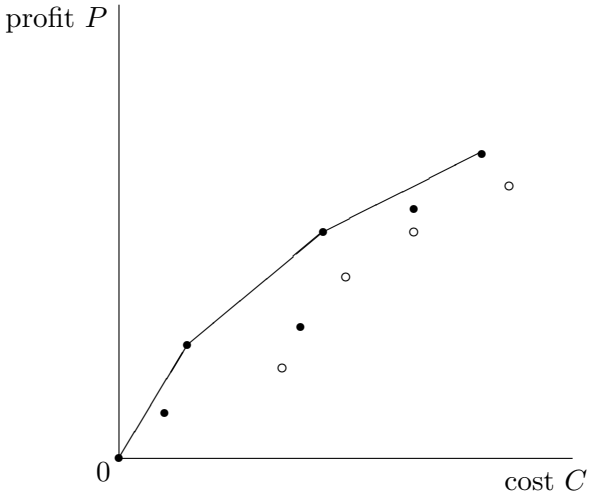

Figure 5.2: Supported set.

The goal of our work is to investigate the difficulty of the bi-criteria TSP with profits on trees. We distinguish the following three problems:

Problem 1 ($\mathcal{E}$): find all efficient points and, for each of them, a corresponding Pareto optimal subtour.

Problem 2 ($\mathcal{SE}$): find all extreme supported efficient points and, for each of them, a corresponding Pareto-optimal subtour.

Problem 3: for a given value of $\lambda \in [0, 1]$, find an associated extreme supported efficient point and a corresponding Pareto-optimal subtour. Note that this is a mono-objective. problem.

Although the topology of our setting is restricted, the questions are ambitious since Problems 1 and 2 are not aiming for a single solution, but instead for a set of solutions. Of course, in order to do so in polynomial time, the number of points in the set should be polynomial in the input size. Notice that the above problems are sorted by decreasing complexity. Indeed, computing a single supported efficient point is not harder than computing the set of extreme supported efficient points that contains it, which in turn is not harder than computing the set of efficient solutions. We prove that there is, from a complexity point of view, a true difference among the three problems since it turns out that, assuming that $P \neq NP$, Problem 1 is more difficult than Problem 2. Also, we show that Problem 2 is in fact more difficult than Problem 3.

Problems 1, 2, and 3 are inherently related to some well-known problems in literature. The following three optimization problems with single objective are usually considered (see also Feillet *et al.* [65]).

Orienteering Problem (OP): find a feasible subtour of maximum profit among those with cost at most C, see, *e.g.*, Golden *et al.* [82].

Prize-Collecting TSP (PCTSP): find a feasible subtour of minimum cost among those with profit at least P. This problem is originally defined by Balas [11].

Profitable Tour Problem (PTP): find a feasible subtour S maximizing the difference $p(S) - c(S)$. This problem is defined as PTP by Dell'Amico *et al.* [48].

Note that solving the PTP is equivalent to finding the supported efficient point corresponding to the combination of the two objectives with $\lambda = 1/2$. Thus Problem 3 is a slight generalization of PTP.

An algorithm for Problem 1 can be used to solve both OP and PCTSP. An algorithm for Problem 3 can be used to solve PTP. We then have the following.

Lemma 5.1 *If either OP or PCTSP are NP-hard then Problem 1 is NP-hard. If PTP is NP-hard then Problem 3 is NP-hard.*

The inverse connection between Problem 1 on the one hand and OP and PCTSP on the other hand is given through the following procedure. The Mono-Bi-Objective algorithm 2 builds up the set of efficient points by solving $|\mathcal{E}|$ instances of PCTSP and $|\mathcal{E}|$ instances of OP. We thus have:

Theorem 5.1 *Suppose that all of the following conditions hold true:*
i) the number of efficient points is polynomial in the size of the input,

Algorithm 2 Mono-Bi-Objective algorithm

1. initialize the list of Pareto optima with $S = \emptyset$ and the list of efficient points with $\{(0,0)\}$; set $P_{\text{tot}} = \sum_{i \in V} p_i$ and $P = 1$;
2. find an optimal subtour S'' for the PCTSP with profit at least P;
3. find an optimal subtour S' for the OP with cost at most $c(S'')$;
4. append S' to the list of Pareto optima, and $(c(S'), p(S'))$ to the list of efficient points (note that $c(S') = c(S'')$); if $p(S') < P_{\text{tot}}$ then set $P = p(S') + 1$ and go to step 2, else return the list of Pareto optima and the list of efficient points.

ii) problem OP is polynomially solvable,
iii) problem PCTSP is polynomially solvable.
Then Problems 1 and 2 are polynomially solvable.

5.2 Problem 1 on trees

5.2.1 Complexity

We show here that, on trees, computing the set of Pareto optimal points (*i.e.*, Problem 1) is NP-hard (Theorem 5.2), and that OP on a tree is equivalent to a generalized knapsack problem with precedence constraints on the items.

Theorem 5.2 *Problem 1 on a tree T is NP-hard, even if T is a star.*

Proof. We show this by proving that the OP on a tree is NP-hard. It then follows from Lemma 5.1 that Problem 1 is NP-hard. Consider the knapsack problem, defined by a set of n elements and a knapsack with capacity B. Each element $i \in \{1, 2, \ldots, n\}$ has an associated weight w_i and value v_i. The problem consists in finding a subset $S \subseteq \{1, 2, \ldots, n\}$ of maximum value $\sum_{i \in S} v_i$ among those with total weight $\sum_{i \in S} w_i$ not exceeding capacity B.

Now consider an instance of OP on a tree consisting of $n+1$ nodes and n edges such that each edge connects the origin (node 0) with a node i, $1 \leq i \leq n$, where each node (except the origin) represents a client. Note that the resulting graph is known as a star. Assign to each edge $(i, 0)$ a cost $c_{i0} := \frac{1}{2} w_i$, and associate with each node $i \neq 0$ a profit $p_i := v_i$. Here the problem consists in finding a subtour maximizing the total profit among those with total cost not greater than B. One easily verifies that there is a one-to-one correspondence between the solutions of the knapsack problem and the solutions of OP. □

The proof of the above result suggests in fact an equivalence between the knapsack problem and OP on a star. Thus, by using a dynamic programming approach for the knapsack problem, we can solve Problem 1 on a star in pseudo-polynomial time. It is then natural to ask if the latter is true also for a general tree.

In order to give an answer, we resort to the *partially ordered knapsack* problem (POK), see also Johnson and Niemi [99] and Samphaiboon and Yamada [154]. POK is a generalization of the knapsack problem that takes into account precedence relations between items. These precedence relations are modeled using a graph where each node corresponds to an item.

Then, item i is a predecessor of j if there is an arc from i to j. An item can only be selected in the knapsack if all its predecessors have been included. In particular, if the graph representing the precedence relations is an *out-tree*, i.e., a directed tree where all arcs are oriented away from a distinguished root node, then we have an *out-tree knapsack problem*.

More precisely, let $T = (V(T), A(T))$ be an out-tree, and let a nonnegative value a_j and a nonnegative weight w_j be associated with every node $j \in V(T)$. Furthermore, let a positive capacity B be given. A node subset $V' \subseteq V(T)$ is called *closed under predecessor* if $j \in V'$ and $(i, j) \in A(T)$ imply $i \in V'$. The out-tree knapsack problem consists in finding a subset $V' \subseteq V(T)$ which is closed under predecessor, such that $\sum_{j \in V'} w_j \leq B$, and $\sum_{j \in V'} a_j$ is maximized.

Johnson and Niemi [99] proposed an efficient dynamic programming procedure that solves the out-tree knapsack problem in $O(nA^\star)$ time, where $n = |V(T)|$ and $A^\star$ is the optimal value of a solution. We observe the following.

Proposition 5.1 *The OP on a tree is equivalent to the out-tree knapsack problem.*

Proof. Consider the OP on a tree. There is a node for each client j with an associated profit p_j and there is a depot, node 0. Each edge has a cost c_{ij}, and is oriented away from the root. Finally, there is a maximum cost C. The corresponding out-tree knapsack problem has an item j with value $a_j := p_j$ and weight $w_j := 2c_{ij}$, where i is the unique predecessor of j in the oriented tree. The budget B is equal to C. It is clear that a solution to the out-tree knapsack problem is equivalent to a solution of OP on the original tree and vice versa. □

A brute force approach for the solution of Problem 1 on trees is the following. First, solve an instance of OP, for every value of cost (capacity) between 1 and $C_{\text{tot}} = 2\sum_{(i,j)\in A} c_{ij}$, by using Johnson and Niemi's algorithm. This results in a list of feasible solutions ordered in C. From this ordered list we can easily select the efficient points as follows. Go through the list in increasing order of C. Consider two neighboring points (γ', π') and (γ'', π''), with cost $\gamma' < \gamma''$. Note that the resulting feasible solutions all have different costs. Then, if $\pi'' \leq \pi'$, eliminate (γ'', π'') and move to the next element in the list. If $\pi'' > \pi'$ immediately move to the next element in the list. Only the efficient points remain. Total time complexity is then $O(nC_{\text{tot}}P_{\text{tot}})$, which is pseudopolynomial. However, we can do better as is explained in the following section.

5.2.2 A dynamic programming algorithm for Problem 1 on trees

In this section we review the "left-right" dynamic programming algorithm by Johnson and Niemi [99] for the out-tree knapsack problem, revised to fit Problem 1 on a tree.

Let $0, 1, 2, \ldots, n$ be a depth first ordering of the nodes of tree $T = (V, E)$, starting with the depot (the root). Let $d(i)$ be the number of children of node i, going away from the depot. Obviously, if the node i is a leaf then $d(i) = 0$.

For each $i \in V$ and $0 \leq s \leq d(i)$, we define $T[i, s]$ as the subtree of T induced by i, the first s children of i taken in order of index, all their successors, and all nodes in V with index lower than i (see Figure 5.3)

We order the trees so that:

(a) $T[i, s]$ precedes $T[i, s + 1]$ for all $i \in V$ and $s \in \{1, \ldots, d(i) - 1\}$;

(b) if j is the sth child of i then $T[i, s - 1]$ precedes $T[j, 0]$ and $T[j, d(j)]$ precedes $T[i, s]$.

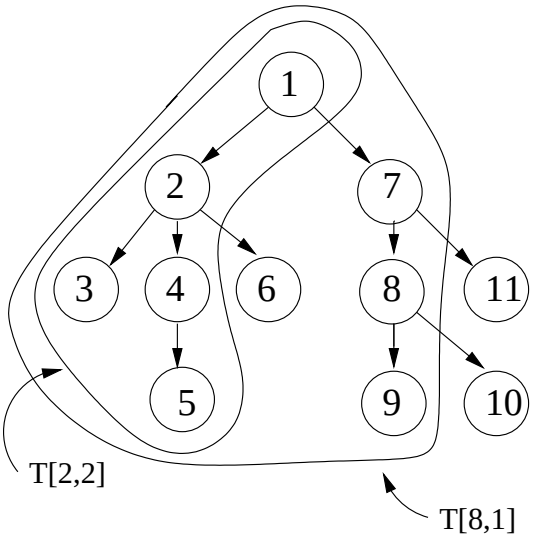

Figure 5.3: Example of subtree in left-right approach.

We define as "Left-Right" this way to order the trees. Note that with the above ordering, every subtree contains all the subtrees that precede it. From the initial tree $T[0,0] = \{0\}$ we gradually expand first down the left edge of the tree and then across the tree to the right. Note further that some trees may be identical. More precisely, if j is the sth child of i then $T[j, d(j)] = T[i, s]$. In other words, each time we have to backtrack while searching the tree in depth-first order, we replicate the same tree. Anyway, we consider all defined trees as distinct objects. As a consequence, the total number of considered subtrees is exactly $2n-1$.

A *q-subtour for $T[i, s]$* is a feasible subtour that starts and ends at node i, visits only nodes of $T[i, s]$ (at most once), and has total profit equal to q. Note that, in the previous example, although $T[j, d(j)] = T[i, s]$, a q-subtour for $T[j, d(j)]$ must contain j, while a q-subtour for $T[i, s]$ needs not.

Let again $C_{\text{tot}} = 2\sum_{(i,j)\in E} c_{ij}$ and $P_{\text{tot}} = \sum_{i\in V} p_i$. For each triple $[i, s, q]$ with $i \in \{0, 1, \ldots, n\}$ and $s \in \{0, 1, \ldots, d(i)\}$ we define $C[i, s, q]$ as the minimum cost of a q-subtour for $T[i, s]$. If there is no q-subtour for $T[i, s]$ then we define $C[i, s, q] = \infty$. This obviously happens for $q < 0$ and $q > P_{\text{tot}}$.

We compute function C by the following algorithm, which is a restatement of the left-right approach in [99]. In step 2(a), i is the largest element in the subtree and, by definition, i will be visited. Thus, the cost of this state is equal to the cost of a subtour with profit q in the largest subtree of T not containing i, increased with the extra cost of visiting i (5.1). In 2(b), the largest element in the subtree is either the subtour with profit $q + p_i$ visits the s-th child of i, client j, or it does not. If j is not visited the cost of this subtour will be equal to the cost of a subtour with the same profit in a subtree not containing j. If client j is visited, the cost of this subtour is equal to the cost of a subtour (with the same profit) that by definition contains j. The minimum of both determines the cost of (5.2).

The advantage of algorithm LR-DP with respect to more intuitive dynamic programming procedures, based on a "bottom-up" search strategy of the tree, is that the evaluation of every entry of the array C takes a constant time. As a consequence, the complexity of LR-DP is linear in the number of entries in C, which is $(2n-1)P_{\text{tot}}$.

It is easy to verify that the values $C[i, s, q]$ returned by algorithm LR-DP satisfy the

Algorithm 3 LR-DP algorithm

1. (Initialization) $C[0,0,0] = 0$; $C[0,0,q] = \infty$ for $q = 1, 2, \ldots, P_{\text{tot}}$;
2. (Recursion) For all subtrees $T[i,s]$ considered in the Left-Right order: for all $q = 0, 1, \ldots, P_{\text{tot}}$:
   (a) if $s = 0$ then
   $$C[i, 0, q + p_i] = C[k, t-1, q] + 2c_{k,i} \tag{5.1}$$
   where i is the t-th child of k;
   (b) if $s \in \{1, \ldots, d(i)\}$ then
   $$C[i, s, q + p_i] = \min\{C[i, s-1, q + p_i], C[j, d(j), q + p_i]\} \tag{5.2}$$
   where j is the s^{th} child of i.

definition. As a consequence, the collection of ordered pairs

$$(C[0, d(0), 0], 0), (C[0, d(0), 1], 1), \ldots (C[0, d(0), P_{\text{tot}}], P_{\text{tot}}),$$

contains the efficient set $\mathcal{E}$. As in the previous section we can easily select the efficient points from this ordered list.

In order to reconstruct the Pareto-optimal subtours associated with the efficient points just computed, we keep track of the computations done in algorithm LR-DP by using a new function W with the same domain as C. More precisely, in step 2(a) of algorithm LR-DP, after computing the value of $C[i, 0, q+p_i]$, we set $W[i, 0, q+p_i] = (k, t-1, q)$, where i is the t-th child of k; in step 2(b), after computing the value of $C[i, s, q+p_i]$, if $C[i, s, q+p_i] = C[i, s-1, q+p_i]$ then we set $W[i, s, q + p_i] = (i, s - 1, q + p_i)$, if $C[i, s, q + p_i] = C[j, d(j), q + p_i]$, where j is the s^{th} child of i, then we set $W[i, s, q + p_i] = (j, d(j), q + p_i)$. For example, since the value of $C[1, 0, p_1]$ derives from $C[0, 0, 0]$, we set $W[1, 0, p_1] = (0, 0, 0)$.

At the end of algorithm LR-DP, starting from an entry of W corresponding to an efficient point (γ, π), we initialize a node subset $V' = \{0\}$. Then, we backtrack on the entries of W until we reach the entry equal to $(0, 0, 0)$. At each step, if we find an entry $(i, 0, q)$, then we add i to V'; if we find an entry $(j, d(j), q)$ then we add j to V'. When we reach $(0, 0, 0)$, V' contains the nodes that form the Pareto optimal subtour corresponding to (γ, π).

This procedure takes at most $O(n)$ steps for every efficient point; this implies that, in the worst case, we can compute all Pareto optimal subtours in $O(nP_{\text{tot}})$.

We summarize the above discussion by the following statement.

Theorem 5.3 *Problem 1 on trees can be solved in $O(nP_{tot})$ time.*

5.2.3 A FPTAS for Problem 1 on trees

It is well-known that the existence of pseudopolynomial dynamic programs lead, under certain conditions, to the existence of polynomial time approximation schemes, see Woeginger [177]. From Papadimitriou and Yannakakis [132] we know that such a polynomial time approximation scheme must exist for our bi-criterion setting. In this section we investigate how to approximate the efficient set in an effective way. We start with a few definitions.

For $\epsilon \geq 0$, a pair (γ, π) is called an *ϵ-approximation* of a pair $(\gamma^\star, \pi^\star)$ if $\gamma \leq (1+\epsilon)\gamma^\star$ and $\pi \geq \pi^\star/(1+\epsilon)$. A set $\mathcal{E}'$ of points in the cost-profit space is called an *ϵ-approximation of the efficient set* $\mathcal{E}$ if, for every $(\gamma^\star, \pi^\star) \in \mathcal{E}$ there exists a point $(\gamma, \pi) \in \mathcal{E}'$ such that (γ, π) is an ϵ-approximation of $(\gamma^\star, \pi^\star)$. Note that the closer ϵ is to zero, the better the approximation of the efficient set.

An algorithm that runs in polynomial time in the size of the input and that always outputs an ϵ-approximation of the efficient set is called an ϵ-approximation algorithm. A *polynomial time approximation scheme* (PTAS) for the efficient set is a family of algorithms that contains, for every fixed constant $\epsilon > 0$, an ϵ-approximation algorithm A_ϵ. If the running time of A_ϵ is polynomial in the size of the input and in $1/\epsilon$, the family of algorithms is called a *fully polynomial time approximation scheme* (FPTAS).

In this section we develop a FPTAS for Problem 1 on trees. We use the standard idea for developing a FPTAS for a knapsack problem, i.e., we scale the profits and apply an exact dynamic programming approach with the scaled profits. A FPTAS for the out-tree knapsack based on this idea is suggested in [99]. However, such a scheme does not translate directly to a FPTAS for Problem 1 on trees. Indeed, in [99] a classical partitioning of the profit space into intervals of equal size is used, that guarantees a bound on the absolute error on every generated point. The bound is chosen so that when the maximum admissible cost (weight) is reached, the relative error ϵ is guaranteed. However, in order to get an ϵ-approximation of the efficient set, we require an algorithm that computes a feasible solution with a relative error ϵ for every possible cost and profit value. To this end, we use the partition of the profit space suggested by Erlebach *et al.* [63] for the multi-objective knapsack problem.

We partition the profit space in u intervals:

$$[1, (1+\epsilon)^{1/n}), [(1+\epsilon)^{1/n}, (1+\epsilon)^{2/n}), [(1+\epsilon)^{2/n}, (1+\epsilon)^{3/n}), \ldots, [(1+\epsilon)^{(u-1)/n}, (1+\epsilon)^{u/n})$$

with $u := \lceil n \log_{1+\epsilon} P_{\text{tot}} \rceil$. Note that the union of all interval generates the whole profit range $[1, P_{\text{tot}})$, and that u is of order $O\big(n \cdot (1/\epsilon) \log P_{\text{tot}}\big)$[1], hence polynomial in the length of the input and in $1/\epsilon$. We can see that in every interval, the upper end-point is $(1+\epsilon)^{1/n}$ times the lower endpoint. Then, we adapt algorithm LR-DP to the new interval profit space. We consider as profits the value 0 and the u lower endpoints of the intervals. For convenience, we denote by ℓ_w the lower endpoints, with $w = 1, 2, \ldots, u$, and we define $\ell_0 = 0$.

A *scaled q-subtour for $T[i,s]$* is a feasible subtour that starts and ends at node i, visits only nodes of $T[i,s]$ (at most once), and has total profit *q or more*.

Instead of C, we consider a different function, denoted $\widetilde{C}$. For each triple $[i, s, \ell_w]$ with $i \in \{0, 1, \ldots, n\}$, $s \in \{0, 1, \ldots, d(i)\}$, and $w \in \{0, 1, \ldots, u\}$, we define $\widetilde{C}[i, s, \ell_w]$ as the minimum cost of a scaled ℓ_w-subtour for $T[i,s]$. If there is no scaled ℓ_w-subtour for $T[i,s]$, then we define $\widetilde{C}[i, s, \ell_w] = \infty$.

In order to obtain the feasible subtours corresponding to the returned points in the cost-profit space, we may use the array W already described for algorithm LR-DP. Notice that in Scaled-LR-DP we directly return the last row of the array $\widetilde{C}$, containing only efficient points (in every state we calculate the cost when profit is equal or larger than ℓ_w).

Theorem 5.4 *Algorithm Scaled-LR-DP is a FPTAS for Problem 1 on trees with time complexity $O(n^2(1/\epsilon)\log P_{tot})$.*

Proof. This proof follows the lines used in Erlebach et al. [63].

[1]Recall that $\log_a(b) = \log(b)/\log(a)$ and $\lim_{x\to 0} \log(1+x) = x$.

Algorithm 4 Scaled LR-DP algorithm

1. (Initialization) $\widetilde{C}[0,0,\ell_0]=0$; $\widetilde{C}[0,0,\ell_w]=\infty$ for $w=1,2,\ldots,u$;
2. (Recursion) For all subtrees $T[i,s]$ considered in the Left-Right order:
   for all $w=0,1,\ldots,u$:
   let $r=\max\{j:\ell_j\le\ell_w+p_i\}$ (i.e., ℓ_r is the largest lower endpoint not greater than ℓ_w+p_i);
   (a) if $s=0$ then
   $$\widetilde{C}[i,0,\ell_r]=\widetilde{C}[k,t-1,\ell_w]+2c_{k,i} \tag{5.3}$$
   where i is the t^{th} child of k;
   (b) if $s\in\{1,\ldots,d(i)\}$ then
   $$\widetilde{C}[i,s,\ell_r]=\min\{\widetilde{C}[i,s-1,\ell_r],\widetilde{C}[j,d(j),\ell_r]\} \tag{5.4}$$
   where j is the s^{th} child of i.
3. (Output) Return the points
   $$(\gamma_w,\pi_w)=(\widetilde{C}[0,d(0),\ell_w],\ell_w)\qquad(w=0,1,\ldots,u).$$

Let the subtrees be numbered according to the Left-Right ordering, i.e.,

$$T_0=T[0,0],\ T_1=T[1,0],\ldots,\ T_{2n-1}=T[0,d(0)].$$

We show that algorithm Scaled-LR-DP returns an ϵ-approximation of the efficient set. More precisely, we show the following claim.

Claim 5.1 *For every $m\in\{1,2,\ldots,2n-1\}$, let $T_m=T[i,s]$, and let h be the largest index of a node belonging to T_m. After performing all update operations, for the optimal cost function C and the approximate cost function $\widetilde{C}$ there exists, for every entry $C[i,s,q+p_i]$ an entry $\widetilde{C}[i,s,\ell_r]$ with:*

(a) $\widetilde{C}[i,s,\ell_r]\le C[i,s,q+p_i]$,

(b) $(1+\epsilon)^{h/n}\ell_r\ge q+p_i$.

where $r=\max\{j:\ell_j\le q+p_i\}$ and $w=\min\{j:\ell_r\le\ell_j+p_i\}$.

Note that when $m=2n-1$ then $T[i,s]=T[0,d(0)]$, $h=n$, and conditions (a) and (b) above imply that the point set returned by algorithm Scaled-LR-DP is an ϵ-approximation of the efficient set.

We prove the claim by induction on the tree index m.

The basis of the induction is given by $T_1=T[1,0]$, where $h=1$. We distinguish between two cases:

Case $q=0$**.** We have:

$$C[1,0,p_1]=C[0,0,0]+2c_{0,1}=2c_{0,1}=\widetilde{C}[0,0,0]+2c_{0,1}=\widetilde{C}[1,0,\ell_r]$$

This proves that property (a) holds with equality.

Property (b) follows from the fact that p_1 and ℓ_r are in the same interval, hence:

$$(1+\epsilon)^{1/n}\ell_r \geq p_1$$

Case $q \geq 1$**.** We have:

$$C[1,0,q+p_1] = C[0,0,q] + 2c_{0,1} = \infty$$

and

$$\widetilde{C}[1,0,\ell_r] = \widetilde{C}[0,0,\ell_w] + 2c_{0,1} = \infty.$$

Then property (a) trivially holds with equality. Also in this case we note that ℓ_r and $q+p_i$ are in the same interval, so that:

$$(1+\epsilon)^{1/n}\ell_r \geq q + p_1$$

This ends the basis of the induction.

Assume that the claim is true for any m, $1 < m < 2n-1$ and consider $T_{m+1} = T[i,s]$. Again, we distinguish between two cases.

Case $s = 0$**.** In this case the highest index in $T_{m+1} = T[i,0]$ is given by node i, then $h = i$. Property (a) follows from (5.1) and (5.3):

$$C[i,0,q+p_i] = C[k,t-1,q] + 2c_{k,i} \geq \widetilde{C}[k,t-1,\ell_w] + 2c_{k,i} = \widetilde{C}[i,0,\ell_r]$$

where the inequality holds for the inductive hypothesis.

Now we consider property (b). By the induction hypothesis, the claim holds with $h = i-1$, then:

$$(1+\epsilon)^{(i-1)/n}\ell_w \geq q.$$

Hence we have:

$$\ell_w + p_i \geq q/(1+\epsilon)^{(i-1)/n} + p_i \geq (q+p_i)/(1+\epsilon)^{(i-1)/n}$$

and as ℓ_r and $\ell_w + p_i$ are in the same interval, with ℓ_r as lower bound, it holds that

$$(1+\epsilon)^{1/n}\ell_r \geq \ell_w + p_i \geq (q+p_i)/(1+\epsilon)^{(i-1)/n}.$$

From these relations property (b) follows immediately:

$$(1+\epsilon)^{i/n}\ell_r \geq q + p_i$$

Case $s > 0$**.** In this case it holds that $h \geq i$. From (5.2) it follows that $C[i,s,q+p_i] = C[i,s-1,q+p_i]$ or $C[i,s,q+p_i] = C[j,d(j),q+p_i]$. By the induction hypothesis we have $C[i,s-1,q+p_i] \geq \widetilde{C}[i,s-1,\ell_r]$ and $C[j,d(j),q+p_i] \geq \widetilde{C}[j,d(j),\ell_r]$. Property (a) follows then immediately:

$$C[i,s,q+p_i] \quad \geq \quad \min\{\widetilde{C}[i,s-1,\ell_r], \widetilde{C}[j,d(j),\ell_r]\} = \widetilde{C}[i,s,\ell_r].$$

Property (b) is proven as in Case $s = 0$.

From Theorem 5.3 and the bound on the number of different values of ℓ_w it follows that the complexity of Scaled-LR-DP is $O\big(n^2(1/\epsilon)\log(P_{\text{tot}})\big)$. □

5.2.4 Some special cases

In this section we analyze some special cases of trees or cost/profit structures where Problem 1 is polynomially solvable.

Trees with equal profits or equal costs

We have proven that Problem 1 is hard, even on a star. However, Theorem 5.3 implies that in the special case where the sum of the all the profits P_{tot} is polynomial in n, Problem 1 is polynomially solvable.

If all node profits are equal (but the edge costs are arbitrary), we may assume $p_i = 1$ for all $i \in V \setminus \{0\}$, so that $P_{\text{tot}} = n$. In this case, algorithm LR-DP solves Problem 1 in $O(n^2)$ time.

If all edge costs are equal (but the node profits are arbitrary), we may assume $c_{ij} = 1$ for all $(i, j) \in E$. In this case, the cost of a feasible subtour is twice the number of visited nodes. Hence there are at most $n + 1$ efficient points (exactly $n + 1$ if all node profits are strictly positive). For all $i \in V \setminus \{0\}$, let i^* denote the parent of i along the unique path from i to 0. Let $M = \max_i\{p_i\} + 1$. We modify the edge costs and the node profits as follows:

$$\begin{array}{ll} c'_{i^*i} = -p_i + M & \text{for all } i \in V \setminus \{0\}, \\ p'_i = 1 & \text{for all } i \in V \setminus \{0\}. \end{array}$$

With the modified costs and profits we are back to the case of general costs and equal profits considered above. It easy to see that there is a one-to-one correspondence between the efficient points (and corresponding feasible subtours) in the modified problem and the original one. Thus Problem 1 in the case of equal costs and arbitrary profits on a tree can still be solved in $O(n^2)$ time by algorithm LR-DP.

The line

Let $T = (V, E)$ be a path, let the edge costs be arbitrary positive and let the node profits be arbitrary positive (except $p_0 = 0$). Results for the line with a latency objective are found in Coene and Spieksma [36]. We distinguish between two cases.

The source is an extreme node If the source is an extreme point, we may assume that the nodes are numbered from left to right, so that the depot 0 is the leftmost node. Then it is easy to see that there are exactly $n+1$ feasible subtours (from the void subpath to the whole path) and they are all Pareto optimal. Furthermore, every feasible subtour corresponds to a different efficient point. A feasible subtour is identified by its rightmost node i.

In this situation, both OP and PCTSP can be solved in $O(n)$ time. And in fact, the more general Problem 1 can also be solved in $O(n^2)$ by Mono-Bi-Objective algorithm. However, the following algorithm solves Problem 1 in $O(n)$ time.

Algorithm 5 Extreme Path algorithm

1. set $\mathcal{E} = \{(\gamma_0, \pi_0)\} = \{(0, 0)\}$;
2. for all $i = 1, \ldots, n$ do: set $\gamma_i = \gamma_{i-1} + 2c_{i-1,i}$ and $\pi_i = \pi_{i-1} + p_i$; append (γ_i, π_i) to $\mathcal{E}$.

The Extreme Path algorithm returns the ordered list of efficient points

$$\mathcal{E} = \{(\gamma_0, \pi_0), (\gamma_1, \pi_1), \ldots, (\gamma_n, \pi_n)\},$$

where the Pareto-optimal subtour associated with (γ_i, π_i) is simply the subpath from 0 to i and back ($i = 0, 1, \ldots, n$).

The source is an internal node If the source is not an extreme point then Problem 1 is a little less straightforward to solve. We start by defining a lower bound on any algorithm solving this problem:

Theorem 5.5 *Problem 1 on the line with the source as an internal node can have $O(n^2)$ efficient points.*

Proof. Consider the following instance of Problem 1 on the line. We are given a set of n nodes on the line and one server positioned at the origin. $n/2$ nodes are positioned to the left of the origin and $n/2$ nodes are positioned to the right of the origin. For all consecutive nodes i and j to the left of the origin it holds that $c_{ij} = 1$ and $p_i = 1$. Let us define d_i as the distance from a right client i to the furthest left client. For all clients j to the right of the origin it holds that $c_{ij} > d_i$, where i is an immediate predecessor of j. The profit of client j, p_j, is larger than the sum of the profits of all clients on the left side of j. In this instance, every combination of a left client and a right client yields a pareto optimal solution. □

We now describe an algorithm solving this problem. Let the source 0 be positioned in any point of the path, so in general there are n_L nodes on the left of the depot and there are n_R nodes on the right of the source, with $n_L + n_R = n$. In this case, any feasible subtour S is just a subpath, containing node 0, and traversed twice, from 0 to a left node i, then from i to a right node j, and finally from j back to 0.

The total number of such subpaths becomes $n_L \cdot n_R \leq n^2/4 = O(n^2)$. We may distinguish three types of feasible subtours: those where the source is the leftmost node, those where the source is the rightmost node, and those where the source is an internal node. Every subtour of the third type is obtained by gluing a subtour of the first type and a subtour of the second type.

In order to solve Problem 1, we then proceed as follows. In a first step we evaluate the costs and profits of all feasible subtours of the first type and of the second type. In a second step, we use this information to evaluate all feasible subtours of the third type. In a third and final step, we build up the set of efficient points by sorting the feasible subtours and eliminating non-Pareto ones.

The first step can be carried out by applying twice the Extreme Path algorithm and thus it can be done in $O(n_L + n_R) = O(n)$ time. The second step is carried out by summing up the evaluation of every subtour of the first type with the evaluation of every subtour of second type. Thus the second step requires $O(n_L \cdot n_R) = O(n^2)$ elementary operations.

We propose to solve Problem 1 by the following algorithm: Step 1 requires $O(n_R) = O(n)$ time, step 2 requires $O(n_L) = O(n)$ time, step 4 requires $O(n_R \cdot n_L) = O(n^2)$ time; step 5 requires $O(n^2 \log n^2) = O(n^2 \log n)$ time. Hence algorithm Internal Path solves Problem 1 in $O(n^2 \log n)$ time. Recall that any algorithm solving Problem 1 on the line needs at least $O(n^2)$ time.

Algorithm 6 Internal Path algorithm

1. call the Extreme Path algorithm to generate the list $\mathcal{E}_R$ of points corresponding to feasible subtours of the first type;
2. call the Extreme Path algorithm to generate the list $\mathcal{E}_L$ of points corresponding to feasible subtours of the second type;
3. set $\mathcal{P} = \emptyset$;
4. for all (γ', π') in $\mathcal{E}_R$ and (γ'', π'') in $\mathcal{E}_L$: insert in $\mathcal{P}$ points (γ', π'), (γ'', π''), and $(\gamma' + \gamma'', \pi' + \pi'')$ (the subtour correponding to the third point is obtained by gluing the subtours corresponding to the first and second points);
5. sort all the points in increasing order of γ and go through this list, eliminating non-efficient points (as explained in the end of Section 3.1).

Problem 1 on a cycle

In case $G = (V, E)$ is a cycle, we assume that $E = \{(0,1), (1,2), \ldots, (n-1, n), (n, 0)\}$. We traverse the cycle *clockwise* if we go from 0 to 1, then from 1 to 2, and so on. There are four types of feasible subtours:

1. tours going from 0 to i (≥ 0) clockwise and then coming back counter-clockwise;
2. tours going from 0 to i (> 0) counter-clockwise and then coming back clockwise;
3. tours going from 0 to i (> 0) clockwise, coming back counter-clockwise beyond 0 up to j ($> i$) and finally going back to 0 again clockwise;
4. the whole cycle.

Note that in cases 1, 2 and 3 it is not necessary to travel further than half the total cost of the cycle, otherwise it would be better to visit the whole cycle. It is then possible to evaluate all tours and to build up the efficient set by modifying algorithm Internal Path.

5.3 Problem 2 on trees

In this section, we consider the identification of extreme supported efficient points. Problem 3 (i.e., finding just one supported efficient point corresponding to a given weighted sum of the objectives) can be solved in $O(n)$ time. We show here that Problem 2 (i.e., finding all extreme supported efficient points) can be done in $O(n^2)$ time by solving a parametric linear program with a very special structure. A similar result is obtained by Hoogeveen [95] who gives an example of a bicriteria problem where the number of efficient points is not polynomially bounded, but the number of supported solutions is. We also show that Problem 2 contains sorting. This implies that *any* algorithm for Problem 2 needs at least $O(n \log n)$ operations.

Let $T = (V, E)$ be a tree, rooted at node 0 (the depot). For all $i \in V \setminus \{0\}$, let $i^\star$ denote the parent of i along the unique path from i to 0. For all $i \in V$, let $\delta(i) \subseteq V$ be the set of children of i; if i is a leaf then clearly $\delta(i) = \emptyset$. We associate with every node i a binary variable x_i, which equals 1 if and only if i belongs to a subtour. Notice that $x_i = 1$

implies $x_{i^\star} = 1$. Problem 2 on a tree corresponds to the following parametric program, where $\lambda \in [0,1]$ is the parameter:

$$\begin{array}{rll} \max & \sum_{i \in V \setminus \{0\}} (\lambda p_i - 2(1-\lambda) c_{ii^\star}) x_i & \\ \text{subject to} & x_0 = 1 & \\ & x_i - x_{i^\star} \leq 0 & (i \in V \setminus \{0\}) \\ & x_i \in \{0,1\} & (i \in V) \end{array} \tag{5.5}$$

nodes are selected such that the sum of the weighted differences between the profits and costs is maximized. A feasible solution consists in a set of *connected* nodes which are also connected to the depot. This is enforced in the constraints in (5.5). The coefficient matrix of problem (5.5) is TUM, since it is the transpose of the node-arc incidence matrix of tree T, where all edges are oriented to the root. Thus we may relax the integrality constraints to nonnegativity constraints. The dual of the relaxed problem is the following:

$$\begin{array}{rll} \min & y_0 & \\ \text{subject to} & y_0 \geq \sum_{j \in \delta(0)} y_j & \\ & y_i \geq (\lambda p_i - 2(1-\lambda) c_{ii^\star}) + \sum_{j \in \delta(i)} y_j & (i \in V \setminus \{0\}) \\ & y_i \geq 0 & (i \in V \setminus \{0\}) \end{array} \tag{5.6}$$

Notice that minimizing y_0 is equivalent to minimizing the y_i. Thus, an optimal solution of problem (5.6) can be described as follows:

$$y_i(\lambda) = \max\{0; (\lambda p_i - 2(1-\lambda) c_{ii^\star}) + \sum_{j \in \delta(i)} y_j(\lambda)\} \quad (i \in V \setminus \{0\}) \tag{5.7}$$

For any fixed value of λ, the unique solution of equations (5.7) can be computed in $O(n)$ time going backward from the leaves to the root. By fixing $\lambda = 1/2$ we get an algorithm for PTP. However, we need to enumerate the solutions for all $\lambda \in [0,1]$.

If $y_0(\lambda) = 0$ then the optimal subtour is empty. Otherwise, by complementary slackness, an optimal subtour visits all nodes i where $y_i(\lambda) > 0$ and $y_j(\lambda) > 0$ for all ancestors of i. More precisely, consider the set-valued function $E^\star : [0,1] \mapsto 2^E$, where $E^\star(\lambda) = \{(i, i^\star) \in E : y_i(\lambda) > 0\}$, and let $T(\lambda) = (V, E^\star(\lambda))$. For any given λ, an optimal subtour visits the nodes belonging to the connected component of $T(\lambda)$ containing the depot (node 0). Thus, in order to enumerate all supported subtours, it is sufficient to record the different values of function $E^\star$.

Suppose to increase parameter λ continuously from 0 to 1. By expanding equation (5.7) recursively, one can see that $y_i(\lambda)$ is a nondecreasing, piecewise linear and convex function of the parameter λ, for all i. As a consequence, if $\lambda' < \lambda''$ then $E^\star(\lambda') \subseteq E^\star(\lambda'')$. Furthermore, $E^\star(0) = \emptyset$ (corresponding to the trivial empty tour) and $E^\star(1) = E$ (corresponding to the complete tour of all nodes). Thus, $E^\star$ changes its value in only $K \leq n-1$ breakpoints, $0 < \lambda_1 < \lambda_2 < \ldots < \lambda_K < 1$.

In order to compute the breakpoints of $E^\star$, we proceed as follows. For any given $\lambda' \in (0,1)$, let $R(\lambda') \subset V \setminus \{0\}$ be the set of the roots of the connected components of $T(\lambda')$, excluding the depot. For all $i \in R(\lambda')$, let $V_i(\lambda')$ be the nodes of the corresponding connected component. Finally, let $\lambda'' > \lambda'$ be sufficiently close to λ'. By expanding equation (5.7) recursively, we see that

$$y_i(\lambda) = \max\{0; \sum_{j \in V_i(\lambda')} (\lambda p_j - 2(1-\lambda) c_{jj^\star})\} \qquad \text{for all } \lambda \in [\lambda', \lambda''] \tag{5.8}$$

If we set $\alpha_i(\lambda') = \sum_{j \in V_i(\lambda')} p_j$ and $\beta_i(\lambda') = 2 \sum_{j \in V_i(\lambda')} c_{jj^\star}$ then we may write (5.8) as

$$y_i(\lambda) = \max\{0; [\alpha_i(\lambda') + \beta_i(\lambda')]\lambda - \beta_i(\lambda')\} \qquad \text{for all } \lambda \in [\lambda', \lambda''] \tag{5.9}$$

It follows that equations (5.9) are valid as long as

$$\lambda'' \leq \min_{i \in R(\lambda')} \left\{ \frac{\beta_i(\lambda')}{\alpha_i(\lambda') + \beta_i(\lambda')} \right\}$$

The right hand side of the above inequality is the next breakpoint.

We search for all breakpoints and the corresponding supported subtours by the following algorithm.

Algorithm Extreme Supported Efficient Points

1. (Initialization)
   (a) $\mathcal{SE} = \{(0,0)\}$ (list of extreme supported efficient points), $\mathcal{S} = \{\{0\}\}$ (list of the node sets covered by the supported subtours), $R = V \setminus \{0\}$ (set of roots), flag = FALSE (if flag = TRUE then a new supported subtour has been found);
   (b) for all $i \in R$: set $\alpha_i = p_i$ and $\beta_i = 2c_{ii^\star}$, and set $V_i = \{i\}$;
2. (Breakpoint computation) for all $i \in R$:
   (a) set $\lambda_i = \beta_i/(\alpha_i + \beta_i)$;
   (b) let $\lambda_{\min} = \min_{i \in R}\{\lambda_i\}$ and $R_{\min} = \{i \in R : \lambda_i = \lambda_{\min}\}$;
3. (Connected components updating) for all $i \in R_{\min}$:
   (a) set $\alpha_{i^\star} = \alpha_{i^\star} + \alpha_i$, $\beta_{i^\star} = \beta_{i^\star} + \beta_i$, and $V_{i^\star} = V_{i^\star} \cup V_i$;
   (b) for all $j \in \delta(i)$: set $j^\star = i^\star$;
   (c) remove i from R and from $\delta(i^\star)$;
   (d) if $i^\star = 0$ then set flag = TRUE;
4. (Optimal subtour storing) if flag = TRUE then
   (a) append (β_0, α_0) to $\mathcal{SE}$ and append V_0 to $\mathcal{S}$;
   (b) flag = FALSE;
5. (Termination test) if $R \neq \emptyset$ then go to Step 3, else return $\mathcal{SE}$ and $\mathcal{S}$.

Theorem 5.6 *Algorithm Extreme Supported Efficient Subtours solves Problem 2 on a tree in $O(n^2)$ time.*

Proof. Correctness follows from the previous discussion. Indeed, it holds that (a) nodes are labeled with an increasing value for λ and (b) if a subtour is feasible for a value λ, it is also feasible for any $\lambda' \geq \lambda$ (see above). In particular, let $\lambda'_{\min}$ and $\lambda''_{\min}$ be two values of $\lambda_{\min}$ computed in two successive iterations of the algorithm. By construction, the node set V'_0 appended to $\mathcal{S}$ after the computation of $\lambda'_{\min}$ corresponds to an optimal subtour for all $\lambda \in [\lambda'_{\min}, \lambda''_{\min}]$, where $\lambda'_{\min} < \lambda''_{\min}$. Hence, V'_0 corresponds to a node of the convex hull of the efficient points.

Concerning complexity, at every iteration at least one node is removed from the set of roots, hence the iterations are at most n. The complexity of every iteration is $O(n)$. □

Theorem 5.7 *Problem 2 is at least as hard as sorting.*

Proof. We will prove this theorem by showing that an algorithm solving Problem 2 can be used to sort a set of n numbers. Given n (distinct) numbers v_i to be sorted, we now build an instance of Problem 2. Consider an instance of Problem 2 on a star. There are n spokes and on each spoke $i \in \{1, \ldots, n\}$ a client is positioned at a distance $c_{0i} = v_i$ from the origin. Each client has a profit $p_i = 1$. A solution to this instance of Problem 2 consists of a set of supported efficient points. Each supported efficient point represents a set, say S, of visited clients corresponding to a certain value $\lambda_{\max} = \max\{i \in S | \lambda_i\}$, with $\lambda_i = \frac{2c_{0i}}{1+2c_{0i}}$. Note that $c_{0i} > c_{0j}$ if $\lambda_i > \lambda_j$. Now for each pair of supported efficient points S_1 and S_2, with $|S_1| < |S_2|$, it must hold that $S_1 \subset S_2$. There is one point i in S_2 that is not in any $S_1, S_1 \subset S_2$, and for such a point i it holds that $\lambda_i > \lambda_j$ for all $j \in S_1$. Hence, since all elements in S_1 have a smaller value for λ, and thus for c, and since no larger set S will contain a new element with lower c, it holds that the index of i in the ordered set must be equal to $|S_2|$. Thus, given a solution of Problem 2, find for each supported efficient point S the maximal element in the set of clients visited and set the index of this element equal to $|S|$. In this way, all numbers v_i can be ordered, which proves the theorem. □

Notice that sorting, in general, takes at least $O(n \log n)$ (Cormen et al. [40]), but if values to be ordered are integers, it can be done in linear time. Thus Theorem 7 implies that any algorithm solving Problem 2 in which costs and profits are strictly positive values takes at least $O(n \log n)$. Notice also that we can solve Problem 2 on the line in $O(n)$. On a line it holds that if you visit a node i, you also visited all clients between i and the origin. Thus, λ_i must be at least equal to $\frac{2c_{0i}}{\sum_{0 \leq j \leq i} p_j + 2c_{0i}}$, for all $0 \leq i \leq n$. Now, at each side of the origin it holds that if $c_{0i} > c_{0j}$ and $\lambda_i < \lambda_j$, then client j will only be visited in an efficient subtour if client i is visited as well. Thus, those clients can be merged into one client with $\lambda = \lambda_i$. Finally, select the remaining clients in increasing order of λ in order to form the supported efficient subtours.

5.4 Conclusions

In this chapter we studied the traveling salesman problem with profits from a bi-objective point of view on graphs with a tree metric. We have considered three problems: finding all efficient points (Problem 1); finding all extreme supported efficient points (Problem 2); finding one efficient point, corresponding to a given combination of the two objectives (Problem 3). For every problem, we have developed efficient algorithms. Moreover, we have analyzed some special cases, including problems on a path.

The following table summarizes our results:

	Tree		Line	
	LB	UB	LB	UB
Problem 1	$> n^k$(unless P=NP)	FPTAS	$O(n^2)$	$O(n^2 \log n)$
Problem 2	$O(n \log n)$	$O(n^2)$	$O(n)$	$O(n)$
Problem 3	$O(n)$	$O(n)$	$O(n)$	$O(n)$

There are some interesting extensions that could be studied in future research. When service times are added to the clients, it no longer holds that all predecessors of a client i

are visited when i is visited. It follows that Algorithm LR-DP can not be applied directly. A similar argument holds when a visit consists in a trip for the client visited (pickup and delivery), or when time windows are added. In these cases it can happen that a client is not visited the first time the server passes by, but it is visited later on in the tour.

Chapter 6

The bi-objective traveling salesman problem with profits: a Branch-and-Cut approach

6.1 Introduction

The purpose of this chapter is to describe a new type of procedure to solve the *Bi-Objective Traveling Salesman Problem with Profits* (BOTSPP).

We introduce an algorithm based on cutting planes within a branch-and-bound approach, typically named branch-and-cut in single-objective optimization, that generates all (supported and non-supported) efficient points in the objective space with respect to both criteria.

The developed procedure recalls the algorithm proposed by Ledesma and Salazar [116] for the bi-objective Traveling Purchaser Problem. Our method embeds a cutting plane generation in a branch-and-bound scheme to find a Pareto optimal solution in the decision space for each efficient point in the objective space of the bi-objective combinatorial optimization problem. As in the method presented by Alves and Clímaco in [1], our approach makes use of cutting planes to help in solving each single-objective problem, taking advantage of computations previously performed that produced other Pareto-optimal solutions in the decision space.

To our knowledge, only Bérubé *et al.* [21] studied the BOTSPP from an exact point of view. They proposed an efficient variant of the ϵ-constraint method for bi-objective combinatorial problems, where exactly one ϵ-constraint problem is solved for each point on the Pareto front. In other words, they solved a series of ϵ-constraint single-objective subproblems, obtained by transforming one of the objectives in a constraint, generating in this way a set of feasible points that can be dominated or not. For this reason, at each step they check if the computed solution belongs to the efficient set. They also provide some improvement heuristics based on the exploitation on information gathered from previous problems, devised to speed up the resolution process. The instances solved derive from a TSP library and are with up to 150 nodes. An advantage of our approach with respect to this one is that each solution found by our algorithm is non-dominated, hence the Pareto frontier can be directly built up, step by step.

The chapter is organized as follows. In Section 6.2 and 6.3 we describe the algorithm and report the *cut pool* procedures used to solve the problem. In the section 6.4 we describe how

the Lin-Kernighan heuristic can be used to improve the performances of our algorithm. In Section 6.5 we introduce some procedures based on the exact branch-and-cut approach that compute an approximation of the Pareto frontier. Finally, Section 6.6 reports information about the tools chosen to implement the algorithm, and gives computational results obtained by running the procedure on several TSPP instances.

6.2 The TSPP Branch-And-Cut Algorithm

In this section, we want to give a general description of the algorithm developed to generate supported and non-supported solutions for BOTSPP(see Section 3.1 for the mathematical formulation of the problem).

The underlying idea is based on an approach that uses the weighting method with additional constraints. It combines linearly the two objective functions introducing a weighting factor for each of them. If these weights, here w_1 and w_2, are considered as parameters, this method can be seen as a linear weighting method. To work with a unique parameter w, we make the normalization $w_1 + w_2 = 1$. This procedure is contained in a binary search that explores specific regions of the decision space, through the use of constraints that restrict the objective space to defined sub-areas. This allows us to generate both supported and non-supported points.

The initial step of the algorithm computes the ideal point (f_1^I, f_2^I) defined by

$$f_1^I = \min_{\sigma \in \mathcal{F}} f_1(\sigma)$$
$$f_2^I = \max_{\sigma \in \mathcal{F}} f_2(\sigma)$$

and the Nadir point (f_1^N, f_2^N) defined as

$$f_1^N = \min_{\sigma \in \mathcal{F}} \{ f_1(\sigma) \mid f_2 = f_2^I \}$$
$$f_2^N = \max_{\sigma \in \mathcal{F}} \{ f_2(\sigma) \mid f_1 = f_1^I \}$$

They define lower and upper bounds on the value of efficient solutions, respectively. From these points we derive the first two efficient points of the problem:

$$(f_1^I, f_2^N),\ (f_1^N, f_2^I)$$

that delimit the criterion space area where Pareto-efficient solutions have to be searched. The algorithm *TSPP Branch-And-Cut* initializes the efficient set S_E with these points, defining in the meantime the first zone in $\mathbb{R}^2$ that has to be explored. Then, it stores these points in the list of pending intervals L. The algorithm iterates on intervals contained in L, searching efficient points in the criterion area delimited by the selected data. To do this, it solves the related mono-objective problem and gives (if it exists) a new efficient point to put in the efficient set S_E and two new intervals to explore. The *TSPP Branch-And-Cut* algorithm continues until all intervals in L are successfully explored. Figure 6.1 shows an example of intervals obtained after the first optimization step.

We can notice that each search area is a rectangle defined by two efficient points, the first located in the lowest-left side, the second in the uppest-right side of the area. In the following sections we will name with $[(f_1^1, f_2^1) \dots (f_1^2, f_2^2)]$ the area in the objective space delimited by the points of coordinates (f_1^1, f_2^1) and (f_1^2, f_2^2).

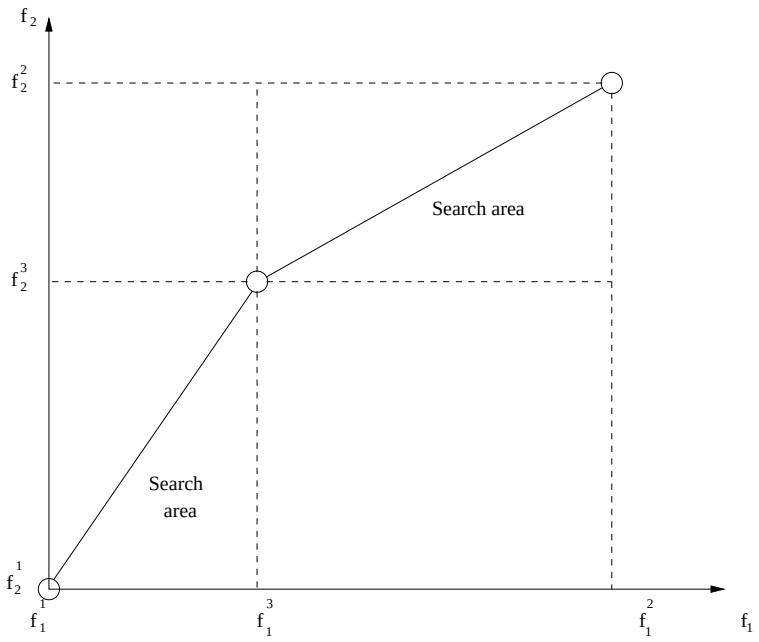

Figure 6.1: First step of the TSPP branch-and-cut algorithm.

The single-objective problem, that here we call TSPP1, related to the area $[(f_1^1, f_2^1) \dots (f_1^2, f_2^2)]$ has the following formulation:

$$\min\ f_1(\sigma) - w f_2(\sigma) \tag{6.1a}$$
$$\text{subject to } \sigma \in P \tag{6.1b}$$
$$f_1(\sigma) > f_1^1 \tag{6.1c}$$
$$f_2(\sigma) < f_2^2 \tag{6.1d}$$

where $w = \left(f_1^1 - f_1^2\right) / \left(f_2^1 - f_2^2\right)$. We assume that the profits are integer numbers, so we can replace the constraints (6.1c) and (6.1d) with the following:

$$f_1(\sigma) \leq f_1^1 - 1 \tag{6.2}$$
$$f_2(\sigma) \geq f_2^2 + 1 \tag{6.3}$$

The pseudo-code of the TSPP branch-and-cut algorithm is shown in Algorithm 7. In (6.4)-(6.7) we compute starting efficient points, used in (6.8) to initialize the list L of pending intervals, and we store them in the efficient set S_E in (6.9). Then, in (6.10) we iterate on the intervals contained in L: at each step, in (6.12) we remove an interval from L and in (6.14) we solve a TSPP1 on the criterion area defined by it. If a solution does exists, then it is added to the efficient set S_E in (6.16) and the point found is used to generate two new intervals to study in (6.17).

In the following sections we analyze the algorithm just described in more detail. In particular, we briefly depict the computation of the starting efficient points, then we linger on the branch-and-cut procedure chosen to solve each TSPP1 subproblem.

Algorithm 7 TSPP branch-and-cut algorithm

$$f_1^1 := \min_{\sigma \in \mathcal{F}} f_1(\sigma) \tag{6.4}$$

$$f_2^1 := \max_{\sigma \in \mathcal{F}} \{ f_2(\sigma) \,|\, f_1(\sigma) \leq f_1^1 \} \tag{6.5}$$

$$f_2^2 := \max_{\sigma \in \mathcal{F}} f_2(\sigma) \tag{6.6}$$

$$f_1^2 := \min_{\sigma \in \mathcal{F}} \{ f_1(\sigma) \,|\, f_2(\sigma) \geq f_2^2 \} \tag{6.7}$$

$$L^I := \left[(f_1^1, f_2^1) \dots (f_1^2, f_2^2) \right] \tag{6.8}$$

$$S_E := \left[(f_1^1, f_2^1),\ (f_1^2, f_2^2) \right] \tag{6.9}$$

$$\text{while } L^I \neq \emptyset \tag{6.10}$$

$$\quad \text{select one interval } \left[(f_1^1, f_2^1), (f_1^2, f_2^2) \right] \text{ from } L^I \tag{6.11}$$

$$\quad L^I = L^I \setminus \left[(f_1^1, f_2^1), (f_1^2, f_2^2) \right] \tag{6.12}$$

$$\quad w = \left(f_1^1 - f_1^2 \right) / \left(f_2^1 - f_2^2 \right) \tag{6.13}$$

$$\quad \sigma^* := \text{TSPP1}(w, f_1^1, f_2^2) \tag{6.14}$$

$$\quad \text{if } \sigma^* \neq \emptyset \tag{6.15}$$

$$\quad\quad S_E := S_E \cup \left(f_1(\sigma^*), f_2(\sigma^*) \right) \tag{6.16}$$

$$\quad\quad L^I := L^I \cup \left\{ \left[(f_1^1, f_2^1) \dots (f_1(\sigma^*), f_2(\sigma^*)) \right],\ \left[(f_1(\sigma^*), f_2(\sigma^*)) \dots (f_1^2, f_2^2) \right] \right\} \tag{6.17}$$

6.2.1 Initial Efficient Points

The computation of the first two efficient points is immediate: this is due to the particular structure of the problem. As explained before, they are computed from the Ideal and the Nadir points. The Ideal values are defined as the maximum criterion values over the efficient set, and they are obtained by simply optimizing each objective function individually over the feasible region. Thus, considering that there are no restrictions about the visit of any node, the minimum cost corresponds to the visit of no nodes and for this reason its value is 0, while the maximum profit corresponds to the visit of all nodes, and its value is the sum of all profit nodes:

$$\left(f_1^I, f_2^I \right) = \left(0, \sum_{i=1}^{n} p_i \right)$$

The Nadir points are the minimum criterion values over the efficient set. They can be obtained optimizing each objective function with the restriction that the other objective must be constrained by the Ideal value. Also in this case, it is easy to see that the maximum profit reachable when the cost is 0 is equal to 0, while the minimum cost needed to visit all the nodes is the solution of a TSP on the whole graph. Thus:

$$\left(f_1^N, f_2^N \right) = \left(\text{TSP}, 0 \right).$$

Hence, for each instance of the BOTSPP problem, we need only to find a TSP solution to obtain the two starting points

$$(f_1^1, f_2^1) = (0, 0) \qquad \text{and} \qquad (f_1^2, f_2^2) = \left(\text{TSP}, \sum_{i=1}^{n} p_i \right)$$

that define the criterion area where the entire Pareto-efficient set is contained. It could be possible that the TSPP branch-and-cut algorithm stops with no feasible tours: the meaning in this case is that the solutions are the two starting points only.

6.3 Branch-and-Cut procedure

To generate cuts for solving the TSPP1, we chose to use the common procedures for cuts generation (see Algorithm 8). So, we relax the subtour elimination constraints from the BOTSPP problem and then we generate at each iteration the cuts needed to eliminate all subtours.

We start with an ILP model for the TSPP:

$$\min \sum_{e \in E} c_e x_e - w \sum_{v_i \in V} p_i y_i \tag{6.18a}$$

$$\text{subject to} \sum_{e \in \delta(v_i)} x_e = 2y_i, \text{ for all } v_i \in V \tag{6.18b}$$

$$\sum_{e \in \delta(S)} x(e) \geq 2y_i, \text{ for all } S \subseteq V \text{ with } \emptyset \neq S \neq V, v_0 \in V \setminus S \text{ and } v_i \in S \tag{6.18c}$$

$$x_e \in \{0, 1\} \tag{6.18d}$$

$$y_e \in \{0.1\} \tag{6.18e}$$

Clearly, the presence of the subtour elimination constraints (6.18c) makes the problem difficult to solve: in fact, they are exponential in the number of nodes, so it is not possible to generate and add all of them to the model. Thus, we chose to solve the problem obtained by dropping these constraints. The subtour obtained has the following properties:

- the objective function value is a lower bound for the optimal value of the problem;
- the solution probably contains subcycles.

This type of problem is known as *separation problem*. The input of the separation problem is a solution vector x_e^*. A separation routine checks if the solution vector satisfies all constraints. If it does not, a specific routine returns violated constraints and adds them to the model. Obviously, it is impossible to check one by one all subtour elimination constraints to find those violated by the solution, but there is a property, based on the *max-flow min-cut theorem*, that allows to directly find the violated constraints.

Theorem 6.1 (Max-flow min-cut theorem (Ford-Fulkerson, 1956)) *In any network, the value of a max flow equals the capacity of a minimum cut.*

The theorem states that the maximum flow in a network is dictated by its bottleneck. In other words, the quantity of material flowing between any two nodes cannot be greater than the weakest set of links somewhere among them.

So, if we compute the minimum cut between every couple of nodes in the graph, we obtain the violated cuts for TSPP1. To do this, we set up a capacitated network by giving capacity x_e^* to edge e, where x_e^* is the solution of the relaxed problem. We fix v_0 as the source node and we consider as the destination each other node of the graph, iteratively. Then, we compute the minimum cut either by a network flow or by a min-cut algorithm. If the obtained minimum cut violates one of the subtour elimination constraints (this can be easily checked substituting the cut value inside the constraints (6.18c)), then it must be added to the model. To avoid the repeated computation of the same cut, when a violated cut is found, the nodes it contains cannot be further considered as feasible destinations.

The computation of a minimum cut has a worst-case running time of $O(nm \log n)$.

Algorithm 8 Cut Pool procedure

- Compute the solution x^* of TSPP1 without subtour elimination constraints;
- do
  - construct a complete undirected capacitated graph G' with a number k of nodes equal to the number of nodes visited in the solution found, and capacity of the edge e equal to x_e^*;
  - fix v_0 as the source node s;
  - for each destination node $t = v_1, \ldots, v_k$:
    - solve a min cut problem on graph G', with source s and destination t, and return *cut value*;
    - if ($cut\ value < 2y_t$)
      - add the cut to TSPP1;
      - drop from the destination set the nodes belonging to the cut just found;
    - compute the solution x^* of TSPP1 with the new cut;

  while x^* does not contain subtours;
- Return the optimal solution x^*.

6.4 Improvement methods

The sequence of subproblems that must be solved to obtain the Pareto efficient frontier have a structure similar to the TSP. As explained before, to reach each efficient point it is necessary to optimize several time the same problem, adding at each iteration those cuts needed to avoid subcycles inside the optimal path. This is done by a special macro given by CPLEX (see Section 6.6.1). To handle the creation/addition of new cut, CPLEX creates a *branch-and-cut search tree*. The nodes of the branch-and-cut search tree correspond to intermediate solutions, (*i.e.*, solutions containing subcycles) that need to be further optimized. New cuts generated from a solution belonging to a given tree node give rise to children nodes. Thus, the computational time depends on the number of generated cuts, and consequently on the width of the branch-and-cut search tree. This suggests that a feasible way to improve the performances is to reduce the size of the branch-and-cut search tree, by trying to reduce

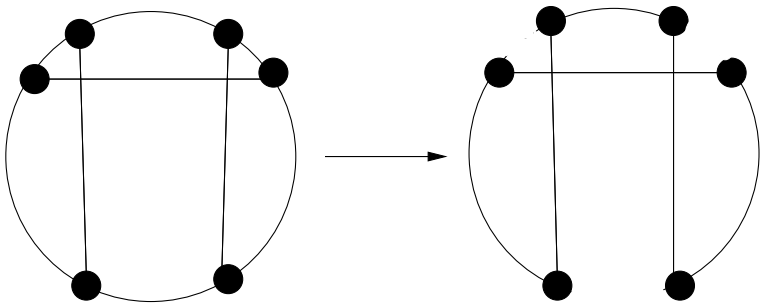

Figure 6.2: A 3-opt move.

the number of cuts needed to reach the solution. CPLEX gives the possibility to suggest a feasible starting solution, which is used to speed up the computational time.

For this reason, we decided to use the Lin-Kernighan heuristic (see the next section) to compute a feasible starting solution, and we suggest it to the branch-and-cut-search tree during the first step of the Cut Pool procedure (Alg. 8).

6.4.1 Lin-Kernighan heuristic

The Lin-Kernighan heuristic [122] is considered to be one of the most successful approaches for generating optimal or near-optimal solutions for the symmetric Traveling Salesman Problem. It can be seen as a generalization of 2-opt and 3-opt methods: it involves swapping pairs of sub-tours to make a new tour. The 2-opt and 3-opt algorithms are special cases of the λ-opt algorithm [121]: here, at each step, λ links of the current tour are replaced by other λ links in such a way that a better tour is achieved. In other words, in each step a shorter tour is obtained by deleting λ links and putting resulting paths together in a new way, possibly reversing one or more of them. Figure 6.2 shows an example of 3-opt exchange.

The λ-opt algorithm is based on the λ-*optimality* concept:

Definition 6.1 *A tour is said to be* λ-optimal *if it is impossible to obtain a shorter tour by replacing any* λ *of its links by any other set of* λ *links.*

Unfortunately, the number of operations to test all λ-exchanges increases rapidly as the number of nodes increases. In a naive implementation, the testing of a λ-exchange has a time complexity of $O(n^\lambda)$. Furthermore, there is no nontrivial upper bound on the number of λ-exchanges. Thus, it is difficult to know what λ to use to achieve the best compromise among running time and quality of the solution.

To overcome this problem, Lin and Kernighan introduced the variable λ_{opt} in their approach. The value of λ_{opt} changes during the execution of the algorithm, each time getting its best value. Thus, given a feasible tour, the algorithm repeatedly performs λ-exchanges that reduce the length of the current tour, until a tour is reached for which no exchange yields an improvement. This process may be repeated many times from initial tours generated in some randomized way.

The solution found by the Lin-Kernighan heuristic is 1.3% above the optimum [92].

In our algorithm, we use a modified and extended version of Lin-Kernighan algorithm, presented by Helsgaun in [92]. The new algorithm is a considerable improvement of the original one. The most noticeable difference is found in the search strategy: the new heuristic

uses larger, and more complex, search steps. Also, new is the use of the sensitivity analysis to direct and restrict the search.

Computational experiments have shown that this new version of the heuristic is highly effective. Even though the algorithm is approximate, optimal solutions are produced with an impressively high frequency.

6.4.2 Improving the starting solution

We choose to use the Lin-Kernighan heuristic to suggest a feasible starting solution to each branch-and-cut search tree. This step has to be performed before the *do-while* iteration in the Cut Pool procedure. All the needed operations are reported in Alg. 9.

Algorithm 9 Starting Solution procedure

- Compute the solution (x^*) of TSPP1 relaxed of subtour elimination constraints;
- if solution (x^*) contains subtours
  - select from starting graph only nodes visited by the solution found, and create a new graph G';
  - invoke Lin-Kernighan on G', obtaining a solution (z^*);
- suggest (z^*) as starting feasible solution to the branch-and-cut procedure.

Hence, each time that we approach to solve an instance of TSPP1 we apply the Lin-Kernighan heuristic to the graph G' obtained with the set of nodes visited in the solution of the relaxed problem. This gives us a feasible tour containing no cycles that minimizes the cost to visit all the nodes of G'. Thus, supposing that the final optimal efficient solution associated with TSPP1 contains most of the nodes visited in the solution of the relaxed instance, we hope to reduce the branch-and-cut tree, getting to the optimal solution in a shorter time.

6.5 Approximate Pareto front

The sequence of TSPP1 problems solved by our procedure leads to the computation of the exact efficient Pareto frontier. As explained in Section 3.2.1, the cardinality of efficient frontier can be exponential in the number of nodes, thus the Pareto set can be composed by points very close to each other, and the computation of all of them can take a considerable time.

To improve the performances, we developed two different approximation algorithms for the TSPP. The first, that we called *Equal Distance Approximation*, computes each new solution starting from the previous computed one. Thus, starting from a fixed approximation value, we are able to find the minimum number of efficient points that give an ϵ approximation of the efficient frontier.

The second type of approach, that we called *Sub-Area search*, divides the objective space in sub-areas. Each sub-area is explored at most one time, in order to find one efficient point contained in it, if it exists. Also in this case, starting from a fixed approximation value, we know in which way the objective space must be divided in order to obtain an $\epsilon-$approximation

of the efficient frontier. For both procedures, we gave a lower and an upper bound on the number fo iterations needed to reach the solution.

Finally, we briefly describe some simple approaches that return particular subsets of the Pareto efficient frontier.

6.5.1 Equal Distance Approximation

This type of approach derives directly from the branch-and-cut procedure just described. The idea arises from some real-life applications, when it is often not useful to have hundreds of solutions very close to each other, obtained after a long computational time: but it is preferable to get a representative sample of them in a reasonable time, instead. In fact, as we mentioned before, the cardinality of the efficient frontier grows exponentially with the number of graph nodes, and for large graph instances most of the efficient points differ from each other for few units only, even if both the cost and the profit have values in the order of thousands.

For these reasons, we decided to generate only a representative subset of the efficient frontier, so that each point in the frontier is computed starting at a distance δ from the others. We then obtain an homogeneous set of exact solutions, each one representing a particular area of the criterion space.

To get an efficient set of this kind, we use the branch-and-cut procedure developed for the exact frontier, incrementing the cost and the profit gaps among adjacent efficient points. This can be easily done by modifying those TSPP constraints that define the delimitation intervals in which the solutions are searched. Thus, each time we compute a new point we need the cost and the profit values of the two points delimiting the current search area, in order to set the constraints needed to model the new search area. Then, we apply the branch-and-cut procedure to find the solutions inside this new area.

The set of solutions obtained in this way is an ϵ-approximation of the exact Pareto efficient frontier. Before to proceed, we recall the definition of ϵ-approximation:

Definition 6.2 *A couple* (γ, π) *is an* ϵ*-approximation of* (γ^*, π^*) *if:*

i) $\gamma \leq (1+\epsilon)\gamma^*$;

ii) $\pi \geq \dfrac{\pi^*}{1+\epsilon}$.

See section 5.2.3 for a deeper description of this topic.

Thus, if we set the value of ϵ *a priori*, *i.e.*, the maximum error that we want to obtain on the solution set, we can compute at each iteration the maximum gap δ that can be set between each couple of solutions.

The pseudocode of the procedure differs from that of Alg. 7 only in the instructions from (6.10) to (6.17), that are replaced as it is shown in Alg. 10.

Theorem 6.2 *The* Heuristic Equal Distance *algorithm returns an* ϵ*-approximation of the exact efficient frontier for the Traveling Salesman Problem with Profits.*

Proof. Let us consider the first two efficient points computed by the algorithm: (γ_0, π_0) and (γ_1, π_1) (see Section 6.2.1). The new criterion-space area where a new efficient solution

Algorithm 10 Heuristic Equal Distance

Same lines (6.4)–(6.9) as in Alg. 7
while $L^I \neq \emptyset$
  select one interval $\left[(f_1^1, f_2^1), (f_1^2, f_2^2)\right]$ from L^I
  $L^I = L^I \setminus \left[(f_1^1, f_2^1), (f_1^2, f_2^2)\right]$
  compute $\delta_1 = \epsilon f_2^1$
  compute $\delta_2 = \epsilon f_1^2/(1+\epsilon)$
  if $\left((f_1^2 - f_1^1) > \delta_2 \text{ and } (f_2^2 - f_2^1) > \delta_1\right)$
    $w = \dfrac{f_1^1 - f_1^2}{f_2^1 - f_2^2}$
    $\sigma^* :=$ TSPP1$(w, f_1^2 - \delta_2, f_2^1 + \delta_1)$
    if $\sigma^* \neq \emptyset$
      $S_E := S_E \cup (f_1(\sigma^*), f_2(\sigma^*))$
      $L^I := L^I \cup \left\{ \left[(f_1^1, f_2^1) \ldots (f_1(\sigma^*), f_2(\sigma^*))\right],\ \left[(f_1(\sigma^*), f_2(\sigma^*)) \ldots (f_1^2, f_2^2)\right] \right\}$

must be searched is obtained from these points, adding a fixed constant δ_1 to the profit value π_0 and subtracting δ_2 from the cost value γ_1, where (see Figure 6.3):

$$\delta_1 = \pi_0 \epsilon \qquad \text{and} \qquad \delta_2 = \frac{\gamma_1 \epsilon}{1+\epsilon}\,.$$

New efficient solutions are recursively searched until one of the following conditions happens:

Case 1: the difference between the profit values of points delimiting the new searching area is less than δ_1;

Case 2: the difference between the cost values of points delimiting the new searching area is less than δ_2;

Case 3: there are no efficient points in the new search area of the criterion space.

To prove that the efficient set found by the *Heuristic Equal Distance* algorithm is an ϵ-approximation of the exact Pareto-efficient frontier, we must show that for every point (γ^*, π^*) in the dotted areas of Fig. 6.4 (*i.e.*, for every point not found by Alg. 10), it exists a point (γ, π) found by the algorithm for which the following conditions hold true:

i) $\gamma \leq (1+\epsilon)\gamma^*$;

ii) $\pi \geq \dfrac{\pi^*}{1+\epsilon}$.

Consider the case depicted in Fig. 6.4. Let (γ, π) be the point computed by the algorithm. The horizontal line that defines the new search area differ from π by the quantity:

$$\delta_1 = \pi\epsilon\,.$$

Thus all the efficient points (γ^*, π^*) in the dotted area will not be computed. It's easy to show that all these points are an ϵ-approximation of (γ, π):

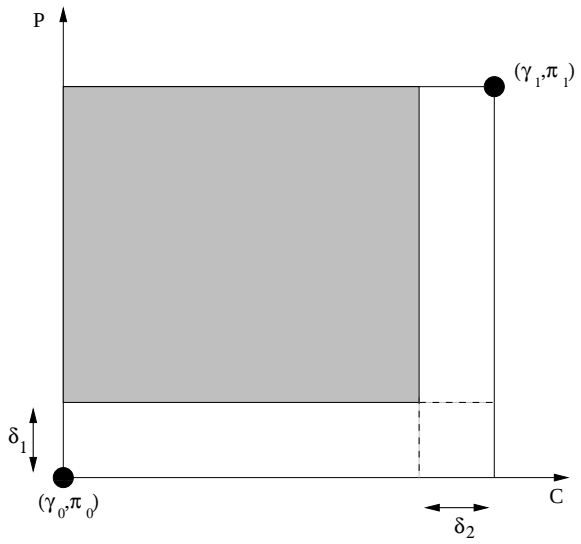

Figure 6.3: the Heuristic Equal Distance algorithm searches new efficient points in the marked area.

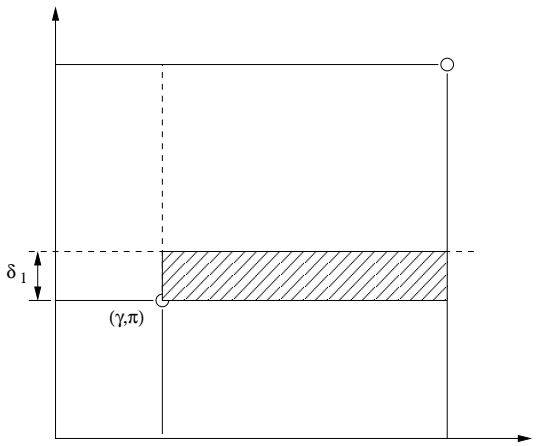

Figure 6.4: The marked area can contain efficient points that are not computed by the Heuristic Equal Distance algorithm.

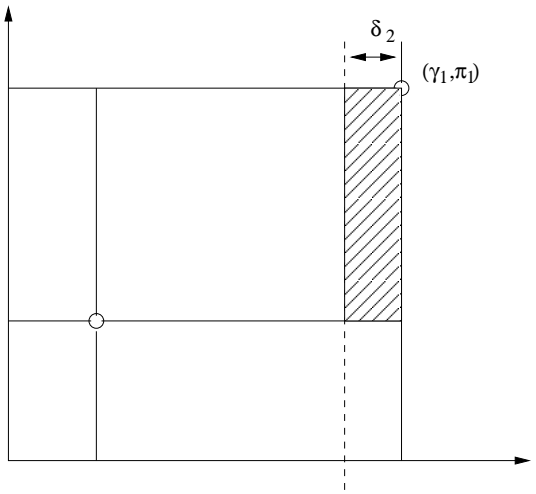

Figure 6.5: The marked area can contains efficient points that are not computed by the *Heuristic Equal Distance* algorithm.

- the property (i) follows immediately from $\gamma \leq \gamma^*$;
- for property (ii) we have

$$\pi^* - \pi \leq \delta_1 \quad \Rightarrow \quad \pi^* - \pi \leq \pi\epsilon \quad \Rightarrow \quad \pi \geq \frac{\pi^*}{1+\epsilon}\,.$$

Thus, (γ, π) is an ϵ-approximation of each efficient point (γ^*, π^*) belonging to the dotted area of Fig. 6.4.

Consider now the case depicted in Fig 6.5. Let (γ^1, π^1) be the point computed by the algorithm. The vertical line that defines the new search area differs from γ^1 by

$$\delta_2 = \frac{\gamma^1\epsilon}{1+\epsilon}\,.$$

All Pareto-efficient points (γ^*, π^*) belonging to the dotted area will not be computed by the algorithm. Again, we can easily show that all these points are an ϵ-approximation of (γ^1, π^1):

- property (ii) follows immediately from

$$\pi^1 \geq \pi^* \quad \Rightarrow \quad \pi^1 \geq \frac{\pi^*}{1+\epsilon};$$

- for property (i) we have

$$\gamma^1 - \gamma^* \leq \delta_2 \quad \Rightarrow \quad \gamma^1 - \gamma^* \leq \frac{\gamma^1\epsilon}{1+\epsilon} \quad \Rightarrow \quad \gamma^1 \leq (1+\epsilon)\gamma^*\,.$$

Thus, it follows that the efficient frontier computed by *Heuristic Equal Distance* algorithm is an ϵ-approximation of the exact efficient frontier of the TSPP. □

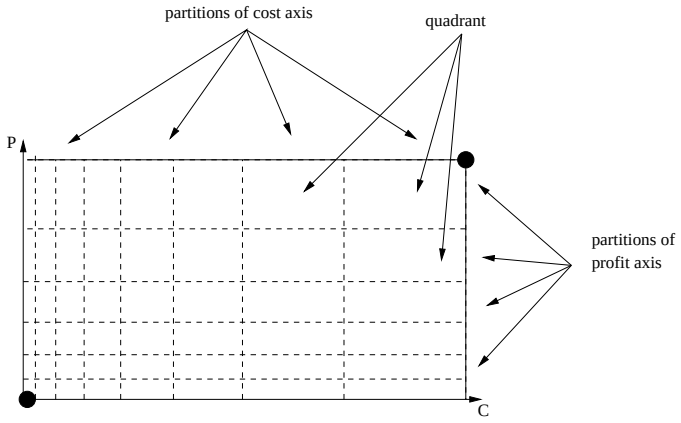

Figure 6.6: Fixed the starting efficient points, the grid is computed by dividing the area in sub-areas.

6.5.2 Sub-Area search

The idea behind the procedure that we describe here comes from the *Equal Distance Approximation* heuristic. In this approach the decision maker decides *a priori* the approximation error on the efficient frontier that will be generated. The difference with respect to the Equal Distance Approximation algorithm is that in the sub-area procedure each solution is searched in a specific part of the objective space, defined by the beginning phase of the algorithm.

The first step is the computation of the starting efficient points $(0,0)$ and (γ_N, π_N) (see Section 6.2.1). Then, the x-axis is divided in k intervals

$$\left[0, \frac{\gamma_N}{(1+\epsilon)^k}\right), \left[\frac{\gamma_N}{(1+\epsilon)^k}, \frac{\gamma_N}{(1+\epsilon)^{k-1}}\right), \ldots, \left[\frac{\gamma_N}{(1+\epsilon)}, \gamma_N\right) \tag{6.19}$$

with $k = \log_{(1+\epsilon)} \gamma_N$ and, analogously, the y-axis is divided in u intervals

$$[0,1), [1,(1+\epsilon)), [(1+\epsilon),(1+\epsilon)^2), \ldots, [(1+\epsilon)^{u-1},(1+\epsilon)^u) \tag{6.20}$$

with $u = \log_{1+\epsilon}(\pi_N)$. Note that the union of all intervals, both in the x-axis and the y-axis, gives the whole objective space delimited by the efficient points $(0,0)$ and (γ_N, π_N). Note further that k is of the order $O\big(\epsilon^{-1}\log(\gamma_N)\big)$ which is polynomial in $1/\epsilon$ and in the starting solutions values. The same happens for u, which is of order $O\big(\epsilon^{-1}\log(\pi_N)\big)$, thus it is polynomial in $1/\epsilon$ and in the starting solutions values. We will call *quadrants* the new sub-areas obtained intersecting all these intervals. In Fig. 6.6 we show an objective space partitioned in this way.

To find one efficient point for each sub-area, we create and solve the single objective problem described in (6.1a)–(6.1d), where f_1^1 and f_2^2 are the lower and the upper endpoints of intervals defining the selected sub-area, respectively. Thus,

$$f_1^1 = \frac{\gamma_N}{(1+\epsilon)^r} \tag{6.21}$$

$$f_2^2 = (1+\epsilon)^s \tag{6.22}$$

with $0 \le r \le k$ and $0 \le s \le u$. For convenience, in the rest of this section we will refer to lower endpoints of 6.19 as ℓ_r and to upper endpoints of 6.20 as u_s, with $\ell_0 = 0$ and $u_0 = 1$.

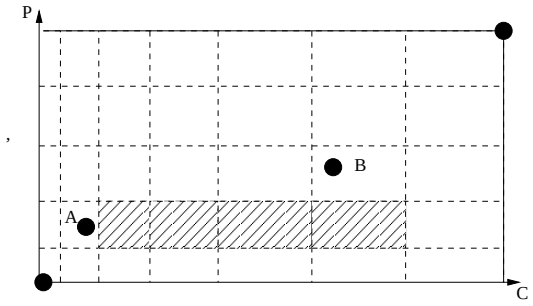

Figure 6.7: Example of Case 2

The algorithm can be obtained from Alg. 7 by replacing the lines from 6.10 to 6.17 with the new instructions shown in Alg. 11, where the *EPHorLayer* procedure is outlined in Alg. 12.

The *Sub-Area Searching* algorithm starts searching a solution in the first vertical partition, in the area immediately on top of that containing the point $(1, 1)$. We do not consider the quadrant containing the point $(0, 0)$ because there are no other solutions between $(0, 0)$ and $(1, 1)$ for the integrality hypothesis on cost and profit values. Thus, we set the number of intervals in (6.25)–(6.26) and we iterate on them (see the *while-do* loops at (6.29) and (6.31)). When a solution is found, either it can coincide with a solution previously found or it can belong to one of the quadrants of the horizontal layer defined by ℓ_r and u_s. In the first case (6.48) no solution is found in the quadrant, so a new search begins in the same vertical layer, but in the quadrant immediately on top of that just analyzed (see (6.49)). In the second case, we must distinguish two possibilities:

1) the solution belongs to the first quadrant defined by ℓ_r and u_s (6.38);

2) the solution belongs to an other quadrant of the horizontal layer defined by ℓ_r and u_s (6.42).

In the situation (1) the algorithm searches new efficient points in the quadrants belonging to the horizontal layer immediately on top of that containing the solution just found (6.41). In the situation (2), we must explore quadrants between those containing the current solution and the last computed solution. In Fig. 6.7 we show with dotted lines those quadrants between efficient points A and B: they represent the region of the objective space that is explored by the *EPHorLayer* procedure in Alg. 12.
When an entire vertical layer is explored, then the search is moved to the quadrant placed in the right-hand side of that containing the last solution found.

In Fig. 6.8 we numbered the criterion space quadrants in the order they are analyzed by the algorithm in a particular instance.

We now prove a relevant result.

Theorem 6.3 *The* Sub-Areas Searching *algorithm generates an ϵ-approximation of the Pareto-efficient frontier for the TSPP.*

Proof. To prove this statement we must show that, in each sub-area of the objective space, the efficient point computed is an ϵ-approximation of all other efficient points belonging to that area and not computed by the algorithm. Let us consider a generic sub-area, as the one represented in Fig. 6.9.

Algorithm 11 Sub-Areas Searching

Same lines (6.4)–(6.9) as in Alg. 7 (6.23)
compute (γ_N, π_N); (6.24)
$u = \log_{(1+\epsilon)}(\pi_N)$; (6.25)
$k = \log_{(1+\epsilon)}(\gamma_N)$; (6.26)
$r = 0; s = 1;$ (6.27)
$LastS = s; LastR = 1;$ (6.28)
while $(r \leq k)$ (6.29)
  $r = r + 1;$ (6.30)
  while $(s \leq u)$ (6.31)
    $f_1^1 = \ell_r; f_2^2 = u_s;$ (6.32)
    $w = \dfrac{\ell_{r+1} - \ell_r}{u_s - u_{s-1}};$ (6.33)
    $\sigma^* = \text{TSPP1}(f_1^1, f_2^2, w);$ (6.34)
    if $\sigma^* \neq \emptyset$ (6.35)
      let (ℓ_{r^*}) be the greater endpoint lower than σ^*; (6.36)
      let (u_{s^*}) be the lower endpoint greater than σ^*; (6.37)
      if $\big((\ell_{r^*} = f_1^1)$ and $(u_{s^*} = f_2^2)\big)$ (6.38)
        $S_E := S_E \cup \sigma^*;$ (6.39)
        $LastS = s^*; \quad LastR = r^*;$ (6.40)
        $s = s + 1;$ (6.41)
      else if $(\ell_{r^*} > f_1^1)$ (6.42)
        $S_E := S_E \cup \sigma^*;$ (6.43)
        EPHorLayer$(r^*, LastS, LastR, S_E)$ (6.44)
        $r = r^*;$ (6.45)
        $s = s^* + 1;$ (6.46)
        $LastR = r^*; LastS = s^*;$ (6.47)
      else if $(u_s^* < u_s)$ (6.48)
        $s = s + 1;$ (6.49)
    if $(\sigma^* = \emptyset)$ (6.50)
      $s = s + 1;$ (6.51)

Algorithm 12 EPHorLayer($r^*, LastS, LastR, S_E$)

$f_2^2 = u_{LastS}$; $\quad r = LastR + 1$;
do
  $f_1^1 = \ell_r$;
  $w = \dfrac{\ell_{r+1} - \ell_r}{u_s - u_{s-1}}$;
  $\sigma^* = \text{TSPP1}(f_1^1, f_2^2, w)$;
  if $\sigma^* \neq \emptyset$
    let ℓ_k be the greater endpoint lower than σ^*;
    $r = k + 1$;
while $\left(\sigma^* \neq \emptyset \text{ and } r \leq r^*\right)$

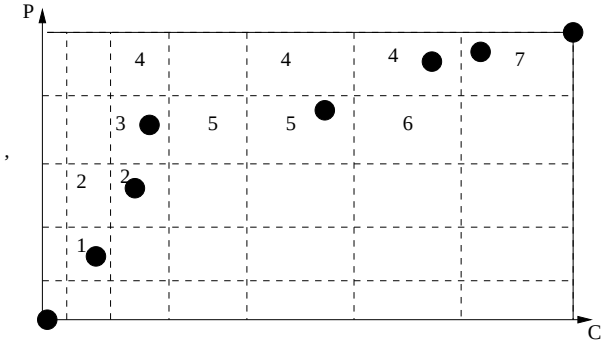

Figure 6.8: Exploration order of the quadrants

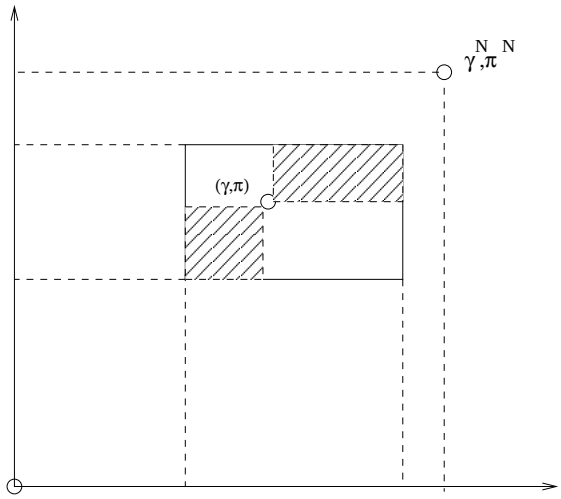

Figure 6.9: Efficient solution in a specific quadrant of the objective space.

Let us suppose that the lower-left endpoint of the quadrant in Fig. 6.9 has coordinates $\left(\gamma_N/(1+\epsilon)^\ell,\ (1+\epsilon)^s\right)$ and the upper-right endpoint has coordinates $\left(\gamma^N/(1+\epsilon)^{\ell-1},\ (1+\epsilon)^{s+1}\right)$. Let (γ, π) be the efficient point found by the Alg. 11 in this area. Then:

$$\frac{\gamma_N}{(1+\epsilon)^\ell} \leq \gamma \leq \frac{\gamma_N}{(1+\epsilon)^{\ell-1}} \tag{6.52}$$

$$(1+\epsilon)^s \leq \pi \leq (1+\epsilon)^{s+1} \tag{6.53}$$

Let (γ^*, π^*) be an efficient point belonging to this sub-area and not computed by Alg. 11. This point can only be in one of the two dropped areas depicted in Fig. 6.9. Indeed, if it would not belong to these areas, then (γ^*, π^*) would dominates (γ, π), which is not possible because (γ, π) is efficient. We then look at what happens in the two dotted areas of Fig. 6.9:

Case 1: let us suppose that (γ^*, π^*) belong to the lower-left dropped area. Thus:

$$\frac{\gamma_N}{(1+\epsilon)^\ell} \leq \gamma^* < \gamma \qquad \text{and} \qquad (1+\epsilon)^s \leq \pi^* < \pi\,.$$

To prove that (γ^*, π^*) is an ϵ-approximation of the efficient point (γ, π), we show that the properties of Def. 6.2 hold true. For the property (i) we have

$$\gamma \leq \frac{\gamma_N}{(1+\epsilon)^{\ell-1}} \quad \Rightarrow \quad \gamma \leq \frac{\gamma_N}{(1+\epsilon)^{\ell-1}} \cdot \frac{1+\epsilon}{1+\epsilon} \quad \Rightarrow \quad \gamma \leq \frac{\gamma^N(1+\epsilon)}{(1+\epsilon)^\ell} \leq \gamma^*(1+\epsilon)\,.$$

Then the property (ii) follows immediately:

$$\frac{\pi^*}{1+\epsilon} \leq \pi^* \leq \pi\,.$$

Case 2: let us suppose that (γ^*, π^*) belong to the upper-right dropped area. Thus:

$$\gamma \leq \gamma^* \leq \frac{\gamma_N}{(1+\epsilon)^{\ell-1}} \qquad \text{and} \qquad \pi \leq \pi^* \leq (1+\epsilon)^{s+1}\,.$$

Again, the property (i) follows immediately by

$$\gamma \leq \gamma^* \leq (1+\epsilon)\gamma^*\,.$$

Hence, the property (ii) holds true because

$$\pi < (1+\epsilon)^{s+1} \quad \Rightarrow \quad \pi < (1+\epsilon)^s(1+\epsilon) \quad \Rightarrow \quad \pi \leq \pi^*(1+\epsilon)\,.$$

We conclude that (γ, π) is an ϵ-approximation of (γ^*, π^*). □

We can give a lower and an upper bound on the number of quadrants that we have to analyze in order to find an ϵ-approximation of the efficient frontier. This number corresponds to the amount of TSPP1 problems that have to be solved to cover the entire criterion space. The lower bound is obtained when all solutions belong to the last vertical layer. In this case we must iterate only on the $u-1$ horizontal layers.

The worst case happens when it is necessary to analyze at least two quadrants of each layer. In Fig. 6.10 we show two examples of feasible efficient frontiers that need the minimum and the maximum number of TSPP1, respectively.

Thus, it is easy to see that, when the number of the vertical layers is k and the number of horizontal layers is u, the number Q of quadrants to visit is bounded by:

$$u - 1 \leq Q \leq k + u - 3$$

where k and u are polynomial. It is interesting to see that, even if the starting number of quadrants is given by the product ku, the effective number of quadrants that must be explored is bounded by the sum $k+u$ (up to a constant term).

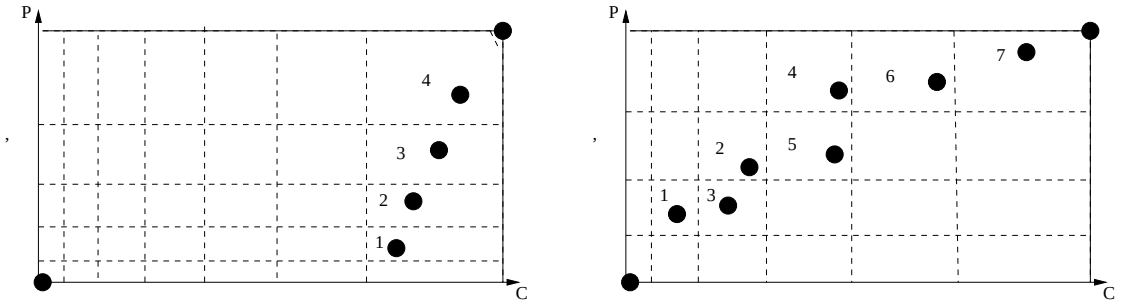

Figure 6.10: Example of the minimum and the maximum number of quadrants that must be explored to find the efficient frontier.

6.5.3 Other approximation approaches

In this section we describe some additional simple approaches, built on the Branch-and-cut approach just described, that compute an approximation of the efficient frontier. These methods were implemeted, but the results are not analyzed in this work.

Random generation of exact solutions

This kind of approach generates only a representative sample of exact solutions belonging to the efficient frontier. The difference with respect to the approximated algorithm just described is that here we decide *a priori* the number of points to generate. The reason of this choice is strictly related to the computational time. In fact, if we have an evaluation of the average time needed to solve a single instance of TSPP starting from a specific set of profit values, we can know the cardinality of the efficient frontier generated for each time interval. So, if we suppose that the available computational time is fixed, then we can know, a priori, the cardinality of the efficient frontier that will be generated, that is the maximum number of solutions that can be computed in the specified time. This can be useful when we need of only a representative sample of the efficient solution set: it is possible to build a clear corrispondence between the cardinality of the efficient frontier and the time needed to compute it.

To do this, each time a new efficient point is generated we store in a vector the two searching areas of the criterion space it defines, and we randomly select in which one to search for the new solution. In other words, we generate points belonging to the Pareto frontier by randomly analyzing those intervals delimiting the searching area in the criterion space generated by the just computed efficient points. Solutions are computed by the branch-and-cut procedure described in previous sections, hence the subset of the Pareto frontier will contain exact points.

Local Search

This type of procedure aims to generate efficient points in a specific area of the criterion space. This could be useful when a traveller needs to know all optimal ways to accumulate at least a specific profit P by spending no more than a given cost C. In Fig. 6.11 we show a typical frontier that can be obtained with this procedure.

The *local search* approach searches all possible exact solutions inside a profit and cost range specified by the user, through the branch-and-cut procedure described in the sections

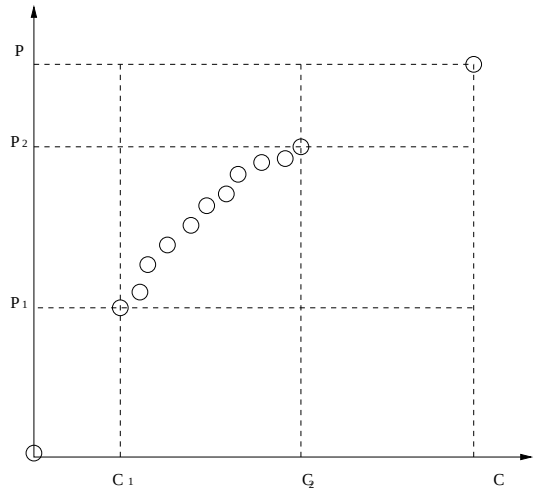

Figure 6.11: Scheme of efficient frontier given by the local search approach.

6.2 and 6.3. It is interesting to investigate how the times can differ if we change the searching area where to generate exact efficient points. Also in this case, it is possible to evaluate, a priori, the average time needed to obtain the efficient set knowing the starting data values.

6.6 Computational results

To assess the performance of our algorithms, we implemented a C++ code and ran it on a AMD Opteron 2.4 GHz processor equipped with CPLEX v. 12.1. Before to proceed with the analysis of the computational time, we briefly discuss the tools used to develop the code. First of all, we shortly overview the CPLEX library, underlying the key functions we use to implement the branch-and-cut procedure. Then we describe the implementation choices for the minimum-cut problem and for the Lin-Kernighan heuristic. Finally, we describe the instance set chosen for the tests and show the obtained results for both the exact and the approximated approaches.

6.6.1 Branch and Cut implementation

CPLEX by ILOG is an optimization software package that solves integer programming problems, very large linear programming problems, quadratic programming problems, and (recently) problems with convex quadratic constraints. It uses the most modern approaches combined with the state-of-the-art solution methods, to give a powerful tool able to solve a wide range of instances.

In the MIP context CPLEX offers *control callbacks*, that are tools that allow to control the branch-and-cut solution search during the optimization of a problem. In particular, they allow to add problem-specific cuts at each node of the branch-and-cut tree search. Callbacks are implemented as an extension of the diagnostic callback class hierarchy. For each one CPLEX offers a *macro* designed to help the programmer during the implementation. In our code we used the following:

```
ILOCUTCALLBACK2(CtCallback, IloExprArray, lhs, IloNumArray, rhs)
```

The parameter `lhs` is an array of expressions, while the parameter `rhs` is an array of values. These parameters are the left-hand side and the right-hand side values of the cuts to be added to model, satisfying

$$\texttt{lhs} \leq \texttt{rhs}\,.$$

In this way, cuts are added when these constraints are violated at a node of the branch-and-cut tree. Thus, this callback takes the list of violated cut as a parameter and uses them to compute the new LP solution.

The cut callback is created and passed to CPLEX by

```
cplex.use(CtCallback(env, lhs, rhs))
```

The method `CtCallback` constructs an instance of the out callback class and returns an handle object for it. This is directly passed to the method `cplex.use`.

To generate the array `lhs` and `rhs` we used the method `makeCuts()`. It receives as input the solution computed by CPLEX in the previous branch-and-cut tree node, and searches the violated constraints, storing them in `lhs` and `rhs`. As we explained in Section 6.3, they derive from the solution of a max-flow min-cut problem, with a fixed source node and $N-1$ feasible destination nodes. To compute the minimum cut we choose to use a function of the *Concorde* package.

Concorde is an ANSI C computer code for the Symmetric Traveling Salesman Problem (STSP) and some related network optimization problems. Its callable library includes over 700 functions allowing the users to solve a wide range of problems related to STSP. In particular, it contains a function that returns minimum s-t cuts in directed and undirected graphs. The function is an implementation of the Push-Relabel Flow algorithm described in Goldberg and Tarjan [80]. The prototype is the following:

```
int CCcut_mincut_st(int ncount, int ecount, int *elist, double *ecap, int s,
                    int t, double* value, int ** cut, int *cutcount)
```

The function receives as input the number `ncount` of nodes in the graph, the number `ecount` of its edges, the list `elist` of edges in node-node format, the edges capacities `ecap`, the source node `s` and the sink node `t`. It returns in `value` the capacity of the minimum cut, in the appropriately sized array `cut` the list of nodes in the minimum cut and in `cutcount` the number of nodes contained in the cut. So, in the implementation of our `makeCuts()` method we invoke the `CCcut_mincut_st` function iteratively by passing as sink node each graph node except the source, thus generating the needed subtour-eliminating cuts.

6.6.2 Computational Results

To analyze the performances of our algorithms we used the same instances used by Bérubé *et al.* in [21]. Thus, we took TSP instances from the TSPLIB [147] and we generated profits in the following three different ways:

- type 1: $p_i = 1$ for each $v_i \in V$;
- type 2: $p_i = 1 + (7141 v_i + 73) \mod 100$;
- type 3: $p_i = 1 + \lceil 99 \frac{c_{1,v_i}}{\theta} \rceil$, where $\theta = \max_{w \in V'} c_{1,w}$.

Instances with profits of type 1 are generally the easier: the cardinality of the efficient frontier generated is equal to the number of starting points. Instances of type 2 have profits values between 1 and 100, while instances of type 3 produces hard problems, where the profit values become larger as their distance from the source increases.

The performances comparison between our approach and the approach developed by Bérubé et al. was initially very difficult to evaluate. In fact, the time employed by our procedure to compute all efficient solutions was entirely spent in the generation of cuts inside the Branch-and-Cut procedure, needed to solve each linear programming model of the algorithm.

If we analyze the algorithmic structure of the two procedures, we can notice that the difference between the approaches lies essentially in the structure of the model that must be solved at each iteration of the algorithm *i.e.*, in the objective function and in the constraints set, and in the modality of exploration of the objective space. Instead, the optimization models are solved in both approaches through a branch-and-cut procedure. For this reason, in order to have a real comparison between performances of our procedure and the approach of Bétubé *et al.*, we implemented the two methods using the same libraries, the same functions and the same hardware resources. In this way, we obtained two codes that differ each other only for the description of each optimization model and the modality of exploration of the objective space.

Tables 6.1, 6.2 and 6.3 show, respectively, the results obtained by running the *TSPP Branch-and-Cut* algorithm with and without the Lin-Kernighan starting procedure on the TSPLIB instances with profits generated in the three ways just described. In the last two columns of the tables we reported the time results obtained running the Bérubé *et al.* approach in our machine with the same structures and resources used for the implementation of our algorithm. In the columns the following information are reported:

- $|V|$: number of nodes considered, including the source.
- EP: number of efficient points in the Pareto front.
- Time: average CPU time in seconds to compute the entire Pareto front.
- Time_A: average CPU time in seconds to solve each TSPP1.

Cells marked by "t.l.e." indicates that the instance was still unsolved after a time limit of 72 hours (259200 seconds).

If we compare the performances of the two approaches, we can notice that the computation time of the Bérubé *et al.* approach is better than the time needed by our procedure to compute the solution set. This is due to the difference between the optimization models that must be solved at each iteration by the two procedures. Indeed, while Bérubé solves a PCTSP, in our approach we build a model with an objective function given by a weighted sum of the profit and the cost functions, and a constraints set that contains all constraints of PCTSP and new ones. This leads to an increment of the average number of cuts needed to eliminate all subtours, and so to an increasing of the time required to CPLEX to solve each single model. In fact, from a detailed study of the times, we noticed that in both approaches, the 95% of the time employed to return the set of solutions is spent by CPLEX in solving optimization models. For this reason, assuming that the number of models solved on average by two methods is the same, we can say that the decisive factor that determinates the effectiveness of the method lies in the time needed to CPLEX to generate the cuts.

If we analyze the performances of our approach when a starting solution computed with the Lin-Kernighan approach is suggested to CPLEX, we note, on average, an improvement of 10%. This suggests that, perhaps, with an effective heuristic that give a good starting solution, and a optimized cut generation, our approach could be margin for improvements.

6.7 Conclusions

In this chapter we described a new algorithm based on cutting planes within a branch-and-bound approach, that computes the entire Pareto efficient frontier for the Traveling Salesman Problem with Profits. The approach explores iteratively the objective space computing, at each iteration, a new efficient point. Further, we introduced two approximation algorithms, built on the same idea of the exact approach, that generate a representive set of the efficient frontier.

To analyze the performaces of our algorithm, we used the same TSP instances tested with the ϵ-constraint method of Bérubé *et al.* [21], and we compare the results. In order to have a effective comparison between the real performances of the two methods, we implemented the Bérybé *et al.* algorithm employing the same structures and libraries used in our approach, and we run the instances on the both implementations. We notice that, the times obtained from the resolution of TSPP by the Berubé *et al.* are, overall, better that ours. This derives basically from the difference between the two optimization models that must be solved at each iteration. Anyway, in our opinion, it could be already possible to obtain an improvement of the performances of our algorithm with a further optimization of the code and better use of the software resource.

If we consider the two procedure from a practical point of view, we can notice that the main difference resides in the modality of generation of the efficient frontier. Indeed, while in the Bérubé approach each efficient point is searched starting from the previous computed, and for this reason the efficient frontier is created onwards starting from the point $(0, 0)$, in our approach the generation of new efficient solutions is made through a uniformed distribution throughout the objective space. This can be useful in the development of heuristics or approximation schemes in fact, let us suppose to have an instance that needs of a too high computational time to be solved. In this case, a user can stop the search of solutions at each moment, consistent with its needs, obtaining always a subset of solutions that describe the entire Pareto set and with a fixed approximation error respect to the exact efficient frontier. Obviously, with the increment of the computation time would increase the density distribution of efficient solutions and descrease the approximation error respect to the exact solutions. For these reasons, even if the computational time is not competitive, the contribution of this procedure resides in its intrinsic structure that can be widely used in the development of approximation algorithms for a large class of bi-objective combinatorial optimization problems.

			Standard procedure		Lin-Kernighan		Bérubé *et al.*	
Instance	$\|V\|$	EP	Time	Time$_A$	Time	Time$_A$	Time	Time$_A$
burma14	14	13	1	0.083	0.7	0.053	0.2	0.015
ulysses16	16	15	1	0.0714	0.68	0.0453	0.4	0.0266
ulysses22	22	21	2	0.0952	1.6	0.0761	2	0.0952381
att48	48	47	65	1.354	60	1.276	53	1.125
eil51	51	50	45	0.89	39	0.78	33	0.647059
berlin52	52	51	68	1.33	61	1.196	54	1.05882
st70	70	69	730	10.57	675	0.782	600	8.69505
eil76	76	75	4696	62.613	4486	59.813	3680	48.5921
pr76	76	75	t.l.e.	-	t.l.e.	-	t.l.e.	-
rat99	99	98	453	4.6224	405	4.132	365	3.6534
kroA100	100	99	14253	141.5	13354	135.34	12619	126.19
kroB100	100	99	18501	186.54	16354	167.4	13667	138.051
kroC100	100	99	4964.3	49.3	4011.3	39.9	3865.3	38.62
kroD100	100	99	27631	276.3	25423	234.1	23509	237.465
kroE100	100	99	14721	148.697	12967	130.01	11648	118.02
rd100	100	99	9121	91.11	8129	82.1	6435.7	64.31
eil101	101	100	29362	299.612	28132	282.45	27942	279.1
lin105	105	104	71536	716.32	66845	669.34	63425	641.8
pr107	107	106	1002.4	10.99	912.4	92.3	835.4	8.34
pr124	124	123	39698	397.67	37566	378.6	32665	324.7
bier127	127	126	9442	74.7778	8112	82.2	7770	61.6667
ch130	130	129	8012	81.01	7341	74.5	6879	65.7
pr136	136	135	t.l.e.	-	t.l.e.	-	t.l.e.	-
gr137	137	136	31247	313.6	28991	291.2	27684	276.1
pr144	144	143	92045	921.56	89941	900.1	86722	873.1
ch150	150	149	26767	268.34	24775	248.5	21475.3	205.4

Table 6.1: Computational results for TSPLIB instances with profits of type 1. Times are in seconds. Positions marked with t.l.e. mean that the time for the solution of the corresponding problem exceeded the lime limit allowed.

			Standard procedure		Lin-Kernighan		Bérubé *et al.*	
Instance	$\|V\|$	EP	Time	Time_A	Time	Time_A	Time	Time_A
burma14	14	59	4.1	0.069	3.5	0.0593	2	0.033898
ulysses16	16	102	6.2	0.0607	5.6	0.0549	5	0.04901
ulysses22	22	130	25	0.1923	20	0.153	15	0.147
att48	48	435	3476	7.2416	2997	6.889	1425	3.2758
eil51	51	225	1951	8.761	1798	7.991	1567	6.964
berlin52	52	406	2980	7.339	2672	6.581	1092	2.69067
st70	70	503	20445	40.64	18995	37.763	15500	30.815
eil76	76	386	41410	107.27	36845	95.45	24446.4	63.3329
pr76	76	-	t.l.e.	-	t.l	-	t.l.e.	-
rat99	99	662	56399	85.194	51005	77.046	45243	68.342
kroA100	100	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
kroB100	100	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
kroC100	100	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
kroD100	100	1128	112855	100.04	95762	959.3	92455	926.5
kroE100	100	1068	170088	159.25	159675	159.9	147994	149.1
rd100	100	920	85647	93.09	79967	802.4	72554	736.6
eil101	101	515	71132	138.12	67221	673.4	60034	601.4
lin105	105	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
pr107	107	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
pr124	124	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
bier127	127	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
ch130	130	-	t.l.e.	-	t.l.e.	-	t.l.e.	-

Table 6.2: Computational results for TSPLIB instances with profits of type 2. Times are in seconds. ("t.l.e." = time limit exceeded)

			Standard procedure		Lin-Kernighan		Bérubé *et al.*	
Instance	$\|V\|$	EP	Time	Time_A	Time	Time_A	Time	Time_A
burma14	14	70	3.8	0.055	3.5	0.05	3	0.04285
ulysses16	16	92	6.1	0.0663	5.6	0.0608	3.72	0.04054
ulysses22	22	128	40.4	0.31	35.7	0.278	18	0.1406
att48	48	438	7336	16.748	6451	14.728	3733	8.522
eil51	51	267	5655	21.17	4665	17.471	3762.03	14.0972
berlin52	52	439	6614	15.066	5742	13.079	3090.56	7.0404
st70	70	452	28769	63.6	22576	49.946	14538	32.163
eil76	76	383	8554	22.33	8012	20.919	7211.9	18.83
pr76	76	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
rat99	99	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
kroA100	100	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
kroB100	100	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
kroC100	100	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
kroD100	100	1063	142365	1339.2	135254	1375.3	127998	1299.1
kroE100	100	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
rd100	100	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
eil101	101	499	79995	160.3	71223	714.3	69003	698.4
lin105	105	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
pr107	107	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
pr124	124	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
bier127	127	-	t.l.e.	-	t.l.e.	-	t.l.e.	-
ch130	130	-	t.l.e.	-	t.l.e.	-	t.l.e.	-

Table 6.3: Computational results for TSPLIB instances with profits of type 3. Times are in seconds.("t.l.e." = time limit exceeded)

Chapter 7

TSPP with Time Windows on trees

The Travelling Salesman Problem with Profits and Time Windows (TSPPTW) deals with finding optimal routes for a traveller that must serve a set of locations, each within a specified time interval.

Time windows arise in problems faced by business organization which work on fixed time schedules. TSPPTW can have several practical applications, such as in vehicle routing problems, bank or postal deliveries, or automated manufacturing environment.

While time constrained routing problems were well studied in last years, research on the Traveling Salesman Problems with Time Windows (TSPTW) has been scan. Pesant *et al.* [136] adapt a constraint programming algorithm for the TSPTW that can handle multiple time windows, while Gendreau *et al.* [73] develop a two-phases method based on an elementary shortest path algorithm to solve a routing problem where the same vehicle performs several routes to serve a set of customers with time windows.

An integration of local search algorithms within a constraint programming framework for combinatorial optimization problems was presented by Nuijten *et al.* [135], in an attempt to gain both the efficiency of local search methods and the flexibility of constraint programming. They apply this approach in a TSPTW framework.

A procedure that finds exact solutions for TSPTW was proposed by Christofides, Migozzi and Toth [32]. They developed a branch-and-bound approach where lower bounds are derived from state-space relaxation of dynamic programming. Their algorithm solved 50-node problems with moderately large time windows. Dumas *et al.* [53] proposed a dynamic-programming approach for the TSPTW that extensively exploits elimination tests to reduce the state space. More recently, Mingozzi *et al.* [125] presented a dynamic-programming approach that embeds bounding functions able to reduce the state space (derived by a generalization of the state-space-relaxation technique) and can also be applied to TSPTW problems with precedence constraints. Balas and Simonetti [17] proposed a special dynamic-programming method that finds an optimal solution if the nodes follow a exact initial order, otherwise it can be used as a linear-time heuristic to exactly explore an exponentially-sized neighborhood. Finally, Ascheuer *et al.* [4] considered several formulations for the asymmetric version of the problem, comparing them within a branch-and-cut scheme. Two complete surveys of the state-of-the-art can be found in [18] and [167].

The aim of this chapter is to give an overview on polynomially solvable cases for TSPPTW, describing some algorithms that find the exact solution set. The chapter is organized as follows. In Section 7.1 we give a TSPPTW formulation. In Section 7.2 we analyze the hypothesis under which the problem is polynomially solvable on a line metric. In the Sections 7.3 and 7.4 we study the TSPPTW complexity when the graph is a cycle or a star, respectively. We

will see how little changes in the starting requirements can take to different computational complexity.

7.1 TSPPTW formulation

Let $G = (V, E)$ be a graph, where $V = \{v_0, v_1, \ldots, v_n\}$ represents a set of locations and E is the set of edges. We associate with each edge $e \in E$ a cost value c_e, that here stands for the travel time needed to pass it, and to each location $v_i \in V$ a profit p_i. Each location $v_i \in V$ is characterized by a *time window* $I = [a_i, b_i]$, where a_i is the *release time* and b_i is the *deadline*. The meaning of these values is the following: the "service" at the node v_i starts at (or after) its release time and ends at (or before) its deadline. Thus the profit can be collected during that time interval only. The objective is to find a route that allows the traveller to accumulate the maximum profit in a minimum travel time, serving each location of V at most once and satisfying the time windows constraints.

To state a feasible formulation of the TSPPTW let us first recall the following notations:

- $t_{i,j}$:= the time needed to travel from node v_i to node v_j;
- T_i := the visit time at node v_i;
- $x_{i,j}$:= the binary decision variable associated with the edge (v_i, v_j). It holds 1 if the edge from node v_i to node v_j is visited, 0 otherwise;
- y_i := the binary variable associated to the node v_i. It holds 1 if node v_i is visited, 0 otherwise.

We can then give the following **TSPPTW formulation**:

$$\max \sum_{v_i \in V} p_i y_i, \quad \min \sum_{(v_i,v_j) \in V \times V} c_{i,j} x_{i,j} \tag{7.1}$$

subject to (7.2)

$$\sum_{e \in \delta(v_i)} x_e = 2y_i, \quad \forall v_i \in V \tag{7.3}$$

$$\sum_{e \in \delta(S)} x_e \geq 2y_i \quad \forall S \subseteq V \text{ with } \emptyset \neq S \neq V, v_0 \in S \text{ and } v_i \notin S \tag{7.4}$$

$$T_j \geq T_i + t_e - M(1 - x_e), \quad \forall e = (v_i, v_j) \in E \tag{7.5}$$

$$a_i \leq T_i \leq b_i \tag{7.6}$$

$$T_0 = 0, \tag{7.7}$$

$$x_e \in \{0, 1\} \quad \forall e \in E \tag{7.8}$$

$$y_i \in \{0, 1\} \quad \forall v_i \in V \tag{7.9}$$

The equation (7.3) and (7.4) are the usual TSPP constraints (see Section 3.1). The constraints (7.5) enforce the temporal relationship among consecutive nodes, while constraints (7.6) specify the interval in which it is possible to visit each node.

The main difference between TSPPTW and TSPP is the restriction on the time in which the traveller can collect the profit at each node. In fact, while in the TSPP we can pick up the profit during the first visit of the node, in the TSPPTW it can happens that the traveller passes a node v_k, collects profits associated to other nodes and picks up p_k during the return. This makes the search of non-dominated solutions more difficult.

7.2 The line

Let $V = \{v_0, v_1, \ldots, v_n\}$ be a set of nodes on a line. We suppose that a traveller leaves the node v_0, representing the source, and must reach the node v_n, that is the destination. To each node $v_i \in V \setminus \{v_0\}$ is associated a time window $[a_i, b_i]$ that describes the time frame during which the service at node v_i must be executed, and a profit p_i that can be taken if the node is visited within its time window. Each pair of nodes (v_i, v_j) is connected by an edge with capacity $c_{i,j}$, that defines the cost to cross it. We can assume, without loss of generality, that the profit, travel time and time windows values are all positive.

Lemma 7.1 *The TSPP and Time Windows on a line is solvable in polynomial time under the following hypotheses:*

- *the traveller cannot pass each edge more than once;*
- *waiting times inside the nodes are allowed.*

This kind of hypotheses can happen, for example, when the sequences of nodes that must be visited in a general graph is already defined for technical-organizational reasons.

To prove the statement we describe a *dynamic programming* approach that finds the exact Pareto frontier for an instance with those features.

We can depict an instance of TSPPTW on a line using a bi-dimensional diagram, as shown in Fig. 7.1, where the x-axis corresponds to time values and the y-axis represents the line nodes. In other words, each horizontal lines intersecting the y-axis corresponds to a node in the line.

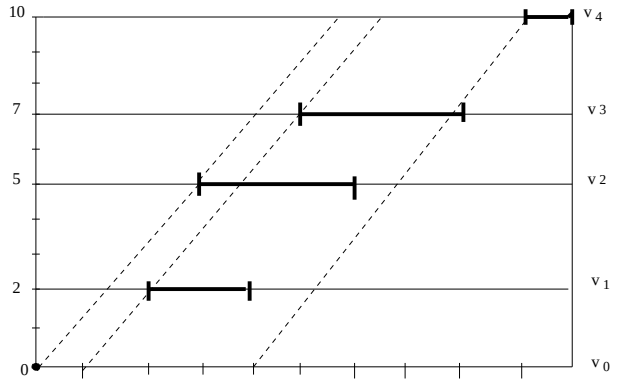

Figure 7.1: Pictorial representation of a TSPP with time windows on a line.

The distance between two horizontal lines is the time needed to cross the edges linking the couple of nodes that they represent, while the highlighted segment of each line indicates the temporal interval in which it is possible to visit the corresponding node, *i.e.*, the time window associated to the node. Each feasible subtour in the line can be depicted as an x-monotone curve starting at the origin and intersecting as many segments as possible. The outlined lines of Fig. 7.1 are feasible subpaths that go through the nodes in the release time a_i, $i = 0, \ldots, n$.

The intersections between the horizontal line corresponding to node v_i and each outlined line correspond to the instant times t a traveller can get to the node v_i, starting from nodes $v_0, \ldots, v_{i-1}$. To save these information, we construct for each node v_i a time array T_i of $i+1$

elements, containing the t times values needed by the traveller to reach the node v_i starting from the nodes $v_0, \ldots, v_{i-1}$. The computation of all T_i arrays is performed as follows:

- for each node index $i = n, \ldots, 0$:
  - for each node $j = i, \ldots, 0$:

$$T_i[j] = a_j + \sum_{k=j}^{i-1} c_{k,k+1}$$

For each $v_i \in V$ and $0 \leq t \leq T_i$ we define:

$P[i,t] :=$ the maximum profit that can be obtained on the nodes $\{v_{i+1}, \ldots, v_n\}$ when at the time t the current position is the node v_i.

The algorithm iteratively builds the solutions starting from the destination node v_n and returning back to the source v_0.

Algorithm 13 TSPPTW on a line

1. (Initialization) for all $T_i[n], i = 0, \ldots, n$:
   (a) if $t \leq b_n$ then $P[n,t] = p_n$,
   (b) if $t > b_n$ then $P[n,t] = 0$.
2. (Nodes Recursion) for all node indices $i = n-1, \ldots, 1$:
   3. (Time Recursion) for all elements in the array $T_i[j]$, $j = i, \ldots, 0$, taken in decreasing order:
      (a) if $t = T_i[j] > b_i$ then

$$P[i,t] = P[i+1, t + c(i,i+1)] \tag{7.10}$$

      (b) if $a_i \leq t \leq b_i$ then

$$P[i,t] = p_i + P[i+1, t + c(i,i+1)] \tag{7.11}$$

      (c) if $t < a_i$

$$P[i,t] = \max\{P[i,a_i], P[i+1, t + d(i,i+1)]\} \tag{7.12}$$

The step (7.10) concerns the case where we arrive in node v_i too late: then the profit of this state is equal to the maximum profit of a tour in the line from node v_{i+1} to node v_n when we arrive at node v_{i+1} at the time $t + c_{i,i+1}$.

The step (7.11) is for the case where we arrive in the node v_i according to its time window, so the profit will be the sum of the maximum profit accumulated in the line from node v_{i+1} to node v_n and the profit of node v_i.

The step (7.12) is for the case where we arrive in the node v_i before the lower value of its time window: thus, either we can wait to take its profit, or we cannot wait. If we decide to wait, then the profit will be equal to the profit of the same line when we arrive in node v_i in the instant a_i, otherwise the profit will be the profit gained in the line from node v_{i+1}

to node v_n from the instant $t + c_{i,i+1}$. The maximum of both determines the profit of this state.

To analyze the number of steps performed by the algorithm we must focus on the two recursions: the first one iterates on the number of nodes, while the second one iterates on the elements of the time array T_i, that contains at most n values. Thus, the complexity of the algorithm is $O(n^2)$, *i.e.*, quadratic in the number of nodes.

7.2.1 TSPP on a line: NP-hardness

In the last section we will show a polynomially solvable case of TSPPTW on a line (see Lemma 7.1), describing a dynamic programming approach able to find the entire Pareto efficient frontier. Now, we want to show how, by modifying the starting hypothesis, the complexity of the problem changes. Thus, let us suppose that there are no bounds on the number of times that each edge can be crossed, and suppose that waiting times on nodes are not allowed. This means that the traveller can pick up the profit associated to a node only if he arrives at the node in its time window. In other words, a traveller can choose to pass through each edge an unlimited number of times, but without possibility to stop until the arrive to destination.

Theorem 7.1 *The TSPPTW on a line, under the hypotheses that:*

- *the traveller can cross each edge an unlimited number of times;*
- *the traveller cannot wait the release time of a node, if he arrives too early;*

is NP-hard.

Proof. Let $KP = (N, C)$ be an instance of the unbounded knapsack problem, in which we have n different item types with weights c_i, $i = 1, \ldots, n$, and profits p_i, $i = 1, \ldots, n$, each one available an unlimited number of times, and let C be the capacity of the knapsack. We have to find a combination of items so that the total weight is less than or equal to C and the total profit is maximum. This problem is NP-hard, as proved in Lueker (1975) by transformation from subset-sum.

We want to show how an instance of KP can be converted to a particular instance of TSPPTW. Let $T = (V, E)$ be a line and let the edge costs and the node profits be arbitrary positive. We can suppose, without loss of generality, that:

- each item i of the KP corresponds to a node v_i of the line for TSPPTW;
- each item profit p_i of the KP corresponds to the node profit p_i for TSPPTW;
- each item weight c_i of the KP correspond to the time needed to cross and return the edge $c_{i-1,i}$ (*i.e.*, $2c_{i-1,i}$), for TSPPTW.

In this way, we build an instance of TSPPTW starting from an instance of KP.

In particular, we can suppose that, to each node v_i of the line is associated the time window $[a_i, a_i]$: this means that, to take the profit associated to the node v_i, we must find a tour that arrives in it exactly at the instant time a_i. So, assuming that the minimum time needed to reach node v_i from the source is c, only one of the following can happen:

- $c > a$, then we cannot take the profit associated to the node,

- $c < a$, then to take the profit it is needed to travel through line edges connecting the source to node v_i until the instant a_i.

Thus, we must find a possible combination of edge costs values that covers exactly the remaining $a_i - c$ times.

If we set C as the time $a_i - c$ for each node v_i, we have to find a feasible combination of edge-cost values that covers exactly C to take the profit p_i associated to node v_i. For this reason, an approach able to find the solution of the unbounded KP can find the solution of TSPPTW. This proves the equivalence of the two problems. Hence, TSPPTW is NP-hard. □

7.2.2 TSPPTW on a line: return to the source

In the previous sections we considered cases in which the origin and destination nodes are different. If we suppose that origin and destination nodes coincide, then if a traveller passes an edge he must pass through the same edge to return to the origin: hence, the profit p_i associated to node v_i can be taken in two different instant times. This makes the problem difficult to solve. In fact, for each node v_i in which the traveller arrives too early, *i.e.*, before the release time, he has to choose whether to wait or not. This choice will bias all future times, keeping in consideration the possibility that the node can be visited also during the return to the source. Hence, through a simple computation, it seems that to find all feasible points we have to consider as solutions for these types of nodes, both possibilities. So, in the worst case, *i.e.*, when the traveller arrives too early in each node of the line, the number of all feasible solutions amounts to $O(2^n)$ items. By these reasons one could apparently conclude that the problem is NP-hard.

If we drop the condition allowing to choose whether to wait or not the release time of each node (that implies that it is not possible to stop during the travel), and we force that a node can be passed at most 2 times, then the problem becomes solvable in polynomial time.

In this section we describe two simple algorithms that compute the entire efficient frontier for TSPPTW on a line under these hypotheses. In the first one we consider the source to be at an endpoint node, in the second one we consider the source to be an internal node of the line.

Before to proceed, we describe an algorithm needed to solve a preliminary problem:

> Given a set of points $\mathcal{P} = \{(\gamma_h, \pi_h) \,|\, h = 1, \ldots, k\} \subset \mathbb{R}^2$ whose coordinates are the cost and the profit of feasible subtours, determine the points that are efficient and sort them according to the following relationships:
>
> $$0 = \gamma_0 \leq \gamma_1 \leq \ldots \leq \gamma_n$$
> $$0 = \pi_0 \leq \pi_1 \leq \ldots \leq \pi_n$$

An algorithm solving this problem is needed to select non-dominated pairs among feasible solutions returned by the procedures that we will describe in the following sections. Such an algorithm is given in Alg. 14. If the list $\mathcal{E}$ is maintained as a binary search tree then the step 3 can be carried out in $O\big(\log(|\mathcal{E}|)\big) = O\big(\log(|\mathcal{P}|)\big)$ time. The other steps require constant time. Thus the complexity of the whole Algorithm 14 is $O\big(|\mathcal{P}| \log(|\mathcal{P}|)\big)$.

Algorithm 14 Efficient Points Algorithm

1. Initialize the list $\mathcal{E}$ with the two points

$$(\gamma', \pi') = (0, 0) \quad \text{and} \quad (\gamma'', \pi'') = \left(2 \sum_{(i,j)\in E} c_{ij}, \sum_{v_i \in V} p_i \right),$$

   in this order;

2. remove a point (γ, π) from $\mathcal{P}$;

3. let (γ', π') be the last point in the list $\mathcal{E}$ with $\gamma' \leq \gamma$ and let (γ'', π'') be the first point in the list $\mathcal{E}$ with $\pi'' \geq \pi$;

4. if $(\gamma', \pi') \neq (\gamma'', \pi'')$, go to Step 5, else go to Step 7.

5. if $\gamma \neq \gamma'$ and $\pi \neq \pi''$ in $\mathcal{E}$, then remove from $\mathcal{E}$ all points between (γ', π') and (γ'', π'') and insert (γ, π) between (γ', π') and (γ'', π'');

6. if $\gamma = \gamma'$, then replace (γ', π') with (γ, π); if $\pi = \pi''$, then replace (γ'', π'') with (γ, π);

7. if $\mathcal{P}$ is not empty then go to step 2.

The source is an extremal node

If the source is an endpoint of the line, we may assume that nodes are numbered from left to right, so the source v_0 is the leftmost node. It is easy to see that there are at most $(n+1)$ feasible subtours. We chose to compute optimal paths in two different steps: the first one computes subpaths from the source to the node v_i, the second one takes subpaths generated in the first phase as starting solutions and returns to the source. Finally, the Efficient Points algorithm extracts solutions belonging to the Pareto-efficient frontier from the feasible ones returned by the procedure.

To keep track of the nodes visited in the first phase of the algorithm, we use the array `VN` storing the n-dimensional vector VN whose i-th component is set to 1 if the node v_i is visited, 0 otherwise.

Let us also define the structure array `T` storing the values of the function $T(i, VN, c)$ giving the maximum profit gained in the path from the origin to the node v_i, with cost c, where VN contains the list of the visited nodes. Finally, let S_F be the set of feasible points. The resolution algorithm, that we call *Extreme Path Time Windows* Algorithm, is outlined in Alg. 15. The first part of the algorithm iterates on the number n of graph nodes to compute the starting values of T, and takes $O(n)$ time. The second part iterates on the values of T and, for each one, on the nodes between the source and node defined by the current value of T itself. It follows that for each T at most n steps are needed to reach a feasible solution. Thus, in the worst case the second phase takes $O(n^2)$ time. We then conclude that the Extreme Path Time Windows (Alg. 15) needs at most $O(n^2)$ time to compute the entire efficient solutions set.

Algorithm 15 Extreme Path Time Windows algorithm

Initialization: $VN_i = 0$, $i = 0, \ldots, n$, $c = 0$, $T(0, VN, 0) = 0$;

First Phase: for $i = 1, \ldots, n$

if $(a_i \leq c_{0,i} \leq b_i)$

$VN_i = 1$;

$T(i, VN, c_{0,i}) = T(i-1, VN, c_{0,i-1}) + p_i$;

else

$T(i, VN, c_{0,i}) = T(i-1, VN, c_{0,i-1})$;

endfor

Second Phase : for each $T(i, VN, c)$ computed in the first phase with $VN_i = 1$:

$\widetilde{VN} = VN$;

for $k = i-1, \ldots, 1$

if $(\widetilde{VN}_k = 0$ and $a_i \leq c + c_{i,k} \leq b_i)$

$\widetilde{VN}_k = 1$;

$T(k, \widetilde{VN}, c + c_{i,k}) = T(k+1, \widetilde{VN}, c + c_{i,k+1}) + p_k$;

else

$T(k, \widetilde{VN}, c + c_{i,k}) = T(k+1, \widetilde{VN}, c + c_{i,k+1})$;

$S_F = S_F \cup \big(c, T(0, \widetilde{VN}, c)\big)$;

endfor

endfor

call the Efficient Point algorithm on S_F;

Source is an internal node

If the source is not an endpoint of the line, then feasible solutions can be obtained in several ways: visiting only the right-hand side of the line, or only the left-hand side, or both. We then need to extend Alg. 15 just described to compute all feasible paths: the resulting algorithm is outlined in Alg. 16.

Algorithm 16 Internal Source algorithm

1. invoke Alg. 15 on the right-hand side of the line and put in S_F all the solutions found;
2. invoke Alg. 15 on the left-hand side of the line and put in S_F all the solutions found;
3. consider each feasible point (c, p) computed in step 1 as the starting solution and invoke Alg. 15 on the left-hand side of the line, then add the found solutions to S_F;
4. consider each feasible point (c, p) computed in step 2 as the starting solution and invoke Alg. 15 on the right-hand side of the line, then add the found solutions to S_F;
5. invoke the Efficient Point algorithm on S_F.

The complexity of this algorithm is determined by steps 3 and 4, where it is necessary to invoke two times the Extreme Path Time Windows (Algorithm 15) starting from the same point. Thus, the Internal Source algorithm takes at most $O(n^4)$ time to find the efficient solutions set.

7.3 TSPP with Time Windows on a cycle

If $G = (V, E)$ is a cycle, let $E = \{(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n), (v_n, v_0)\}$ be the edge set. Let assume also that we traverse the cycle *clockwise* if we go from node 0 to node 1, then from node 1 to node 2 and so on. We now show that the TSPPTW on a cycle can be lead to the case of the TSPPTW on a line.

If we assume that:

- the traveller cannot wait in a node if he arrives before its release time;
- the traveller can visit each node at most two times;

then it exists a polynomial procedure to find the entire efficient Pareto set. Before to describe such a procedure, we notice that in a cycle we have four types of feasible subtours:

1. tours going from v_0 to v_i $(i \geq 0)$ clockwise and then coming back counter-clockwise;
2. tours going from v_0 to v_i $(i > 0)$ counter-clockwise and then coming back clockwise;
3. tours going from v_0 to v_i $(i > 0)$ clockwise, coming back counter-clockwise beyond 0 up to v_j $(j > i)$ and finally going back to v_0 again clockwise;
4. the whole cycle, only if the maximum value of the time windows is greater than the sum of all costs.

To compute the entire Pareto efficient set we have to find all feasible solutions for each set of tours, and then invoke the Efficient Point algorithm to select the non-dominated ones among them. Then, to compute the tours of type 1, 2 and 3 we can simply apply the Internal Path algorithm, while for paths of type 4 it is sufficient to apply only the first part of Alg. 15, considering the cycle as a line with an source and a destination node. Obviously, in the latter case the algorithm has to be run in both the right- and the left-wise of the cycle. Even if the computation of type 4 paths takes $O(n)$ time, the complexity for the other types of subtours is $O(n^4)$, thus computing the whole Pareto-efficient frontier for a cycle takes at most $O(n^4)$ time.

7.4 TSPP with Time Windows on a star

In this section we analyze the difficulty in searching Pareto-efficient solutions for TSPPTW when the metric is a star. Also in this case, it is interesting to notice how the complexity changes by modifying the starting hypothesis on the graph.

Let us consider the TSPPTW on a star, with arbitrary costs, profits and time windows. Then the following statement holds true:

Theorem 7.2 *The TSPPTW on a star is NP-hard.*

Proof. To prove the result, it is enough to notice that the TSPP can be seen as a particular instance of the TSPPTW, where each time window $[a_i, b_i]$ associated to node v_i has release time equal to 0 and deadline greater than or equal to the longest feasible tour. Then the thesis directly follows from Theorem 7.1. □

This result changes if we modify the starting assumptions. In fact, if we assume that all the edge costs $c_{0,i}$ and all the time windows $[a_i, b_i]$ are equal for all $v_i \in V$, then the search of Pareto-efficient solutions becomes easier.

We may assume, for simplicity, that the time windows are normalized to $[a_i/c_{0,i}, b_i/c_{0,i}]$ and the costs are set to 1, *i.e.*, $c_{0,i} = 1$ for all $(v_i, v_j) \in E$. In this case, if we create lists with all possible permutations of nodes, it follows that in each list the traveller can take the profit associated to nodes with the k index satisfying the relation:

$$(2k+1) \in [a, b]$$

or, equivalently,

$$\left\lceil \frac{a-1}{2} \right\rceil \leq k \leq \left\lfloor \frac{b-1}{2} \right\rfloor .$$

Hence, the efficient solutions can be obtained by visiting those nodes with the largest values of profit. For this reason, it is needed to sort the nodes in decreasing order of profit values, and the efficient points can be easily obtained in the following way:

$$\mathcal{E} = \left\{ (0,0), \left(p_1,\, 2\left(1 + \left\lceil \frac{a-1}{2} \right\rceil\right)\right), \left(p_1 + p_2,\, 2\left(2 + \left\lceil \frac{a-1}{2} \right\rceil\right)\right), \ldots \right\} .$$

The complexity of the procedure is $O\big(\lceil (b-a)/2 \rceil\big)$, while the complexity of the ordering phase is $O(n)$ if the numbers to be ordered are integer. Then, it is possible to compute the entire efficient solution set in $O\big(\max\{n, \lceil (b-a)/2 \rceil\}\big)$ time.

We can apply the same procedure on a star with equal profits and equal time windows, but arbitrary costs. In fact, if we modify the edge costs and the node profits as described in Section 5.2.4, the instance of the problem can be driven to the case just explained. Hence:

Lemma 7.2 *Let us consider the TSPPTW on a graph for which one the following properties holds true:*

- *all the edges have the same costs and all nodes have the same time window;*
- *all the nodes have the same profit and the same time window.*

Then the TSPPTW can be solved in $O\big(\max\{n, \lceil (b-a)/2 \rceil\}\big)$ *time.*

7.5 Conclusions

In this chapter we analyzed the complexity in the search of Pareto efficient solutions for the TSPPTW on simple metrics. We showed that the difficulty of the problem changes significantly with the starting hypothesis. In particular, we showed that the TSPPTW on a line is polynomially solvable if both the source and the destination nodes do not coincide and if each of the nodes in the graph can be visited at most once.

The complexity changes if the traveller starts and ends its travel to the source node. In this case, we develop an algorithm that compute the efficient frontier in polynomial time under the hypothesis that the traveller cannot wait the release time of a node if he arrives there too early and he can visit each node at most 2 times. If we relax the restriction on the maximum number of times that it is possible to visit a node, the problem becomes NP-hard.

If the metric is a cycle, then we showed that the problem can be driven to the line and the complexity is the same. Finally, in the case of a star we showed that the problem is NP-hard in general but, under some assumptions on the profits, the costs and the time windows, the efficient frontier can be computed in pseudo-polynomial time. All the results are summarized in the following tables:

	Origin node $\neq$ Destination Node
Line, internal source	$O(n^2)$
Line, external source	$O(n^2)$

	Origin Node = Destination Node	
	No Waiting Times	No Waiting Times Unlimited Time Passing
Line, internal source	$O(n^2)$	NP-hard
Line, external source	$O(n^4)$	NP-hard
Cycle	$O(n^4)$	NP-hard
Star, same TW and Costs	$O\big(\max\{\lceil (b-a)/2 \rceil, n\}\big)$	$O\big(\max\{\lceil (b-a)/2 \rceil, n\}\big)$
Star, same TW	NP-hard	NP-hard
Star	NP-hard	NP-hard

Chapter 8

A dynamic programming approach to the TSPP solution

Dynamic programming is a general approach used to solve problems in many areas of Mathematics and Computer Science. Basically, it can be applied whenever a problem has an optimal substructure, so solutions to the whole problem can be obtained from solutions of its subproblems. The book by Bellman [20] gives an extensive introduction into the field and can still be seen as the most useful general reference.

In this chapter we describe an exact approach to build the efficient Pareto frontier for the TSPP. The algorithm proposed derives from a dynamic programming approach for the *cycle problem*. It recursively searches efficient solutions by exploring subsets of nodes in the graph G, and associates each found subtour to a node in a search tree T. In this way, the search of non-dominated solutions can be simply done by comparing values associated to tree nodes.

The chapter is organized as follows. We begin by describing the Cycle Problem and the procedure used to construct the recursion formula. Then, we depict the code developed lingering on the basic data structures chosen to model the problem. Finally, we show the obtained results, and propose some possible future works.

8.1 The Dynamic Programming Approach

The aim of this section is to give a brief description about the recursion procedure developed to solve the problem. The main idea under the construction of the method derives from a dynamic programming approach developed for the cycle problem.

The *Cycle Problem* consists in finding a cycle of minimum cost that passes through every node of a graph G at most once. A deeper description of the problem can be found in Section 1.4.4. We will show that a procedure that solve the Cycle Problem can be modified to find solutions for the TSPP.

Therefore, we start with a description of a general dynamic programming approach for the Cycle Problem, and we extend it to a procedure to solve the TSPP.

8.1.1 Dynamic Programming for Cycle Problem

Let $G = (V, E)$ be a graph, where $V = \{v_0, v_1, \ldots, v_n\}$ is the set of nodes and E the set of edges. Let c_e be the cost associated to each edge $e \in E$. We want to find the cycle of

minimum cost that starts from the source node v_0, passes through some nodes exactly once and returns to the source.

Let S be a node subset in the graph G. We define:

$Z(S, i) :=$ the minimum cost of a path that starts from the source v_0 and passes through each nodes in $S \subseteq V \setminus \{v_0\}$ exactly once, ending in the node $v_i \notin S$.

Thus, the solution of the cycle problem can be reached through the following recursion:

Initialization: $Z(\emptyset, 0) = 0$; $Z(\emptyset, i) = c_{0,i}$ for all $v_i \in V$

Recursion: $Z(S, j) = \min_{i \in S}\{Z(S \setminus \{i\}, i) + c_{i,j}\}$, for each $S \subseteq V$ and for each $v_j \in V \setminus S$.

In the initialization phase the cost to visit an empty set of nodes is set to 0, while the cost to visit one node v_i is set to the cost of the edge linking the source with v_i. The recursion procedure computes the minimum cost to visit a subset S of the nodes where the last node that must be visited is fixed. The cost of the state $Z(S, j)$ is equal to the cost of a path that visits all nodes in S ending in the node v_i, increased by the cost of passing through the edge linking the final node v_i to the new ending node v_j. In other words, to compute the value $Z(S, j)$ we have to find a final node $v_i \in S$ that minimize the sum of $Z(S \setminus \{i\}, i)$ and $c_{i,j}$.

Hence, the optimal solution for the cycle problem can be obtained as follows:

$$\min_S\{Z(S, 0)\} \text{ for all } S \subseteq V \setminus \{v_0\}$$

that is, the minimum cost to visit all nodes of the set S where the last visited node is v_j, added to the cost of returning to the source from v_j.

This procedure needs an exponential number of steps to reach the optimal solution value. The complexity derives from the need to analyze every subset S of the nodes set V. This makes the approach very inefficient, even for small values of n.

8.1.2 Dynamic Programming for TSPP

The procedure just described can be extended to find all feasible solutions of the TSPP. To better understand the iterative step and the dominance relationships, we save the information on the subtours computed during several algorithm iterations in a *state*. Each *state* K contains four parameters:

$$K := (S, j, p, c)$$

whose meaning is the following:

- $S \subseteq V$ is the subset of nodes visited in the state;
- j is the index of the last node visited in the optimal path relative to the node set S;
- p is the profit of S, *i.e.* the sum of profits associated to nodes belonging to S;
- c is the value of the minimum cost needed to visit exactly once all nodes in S.

Thus, each state represents a different path built on a sequence of previous choices. The visit of a new node in the graph G generates a state transition.

We say that a state $K = (S, j, p, c)$ *dominates* a state $K' = (S', j', p', c')$ if:
1) the set of destinations S' is a subset of S;
2) the index j of the last node visited is equal to j';

3) $p \geq p'$;
4) $c \leq c'$;
and one of the two inequalities 3 and 4 holds strictly. This means that one state K' dominates a state K if there is a path visiting a subset of the nodes in K, ending at the same node, but with minor cost and larger profit. The possibility to find dominated solutions is clearly restricted to graph with non-Euclidean metric. Otherwise, the minimum cost associated to a set of nodes is always greater than the cost of visiting any one of its subsets.

The dynamic programming procedure recalls those explained in the previous section for the cycle problem. Let us define:

$Z(S, j, p) :=$ the value of the minimum cost needed to visit the nodes contained in S with the restriction that the last node to be visited must be v_j,

where the value of p is simply the sum of profits of nodes in S. It is clear that there exists a one-to-one relation between each state K and $Z(S, j, p)$. We can describe a state $K = (S, j, p, c)$ as a function of Z by simply substituting the cost value with that returned by the function.

The recursion procedure is:

Initialization: $Z(\emptyset, 0, 0) = 0$;

Recursion: $Z(S, j, p) = \min_{i \in S \setminus \{j\}} \{ Z(S \setminus \{j\}, i, p - p_j) + c_{i,j} \}$

The Pareto set will contain all non-dominated pairs of solutions given by: $\big(Z(S, j, p), p\big)$, computed during the whole search. The algorithm we use is described in Alg. 17.

Algorithm 17 Dynamic Programming algorithm

- (Initialization) $Z(\emptyset, 0, 0) = 0$; card $= 1$;
- while (card $\leq N$)

  (Recursion)
  for each subset $S \subseteq V$ with $|S| =$ card:
  $Z(S, j, p_S) = \min_{i \in S \setminus \{j\}} \{ Z(S \setminus \{j\}, i, p_S - p_j) + c_{i,j} \}$;
  end for
  card = card + 1;

  end while

In the first step the algorithm initializes the value of Z and card, then it iterates on the cardinality of S. So, for each value from 1 to n, where n is the total number of nodes in the graph G, the recursion formula is applied to each subset S of V with cardinality equal to card.

8.2 Implementation

The simplicity of the dynamic programming formulation does not implies an easy implementation. Furthermore, besides the computational complexity of the approach, the algorithm performance is strongly determined by the implementation choices.

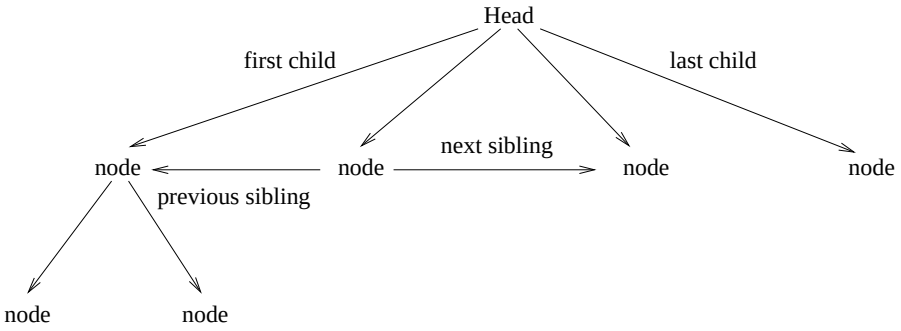

Figure 8.1: Structure of a tree in the `tree.hh` library header.

The generation of a new path depends basically on the solutions previously generated, so the main difficulty arises from the necessity of handling the relation among the subtours generated during the several phases of the algorithm.

It is easy to note that the simplicity in computing the pair (cost, profit) associated to a subtour P derives from the knowledge of the better (cost, profit) values of each subtour that visits every subset of permuted nodes of P. Hence, the difficulty in the implementation phase arises from the difficulty of maintaining a link among optimal (cost, profit) values associated to each subtour and the best pair of solutions $(\text{cost}', \text{profit}')$ related to its subset of nodes.

To simplify the modeling of the problem, we have chosen to describe the several states generated in the construction of the efficient frontier through a tree structure, where a one-to-one relation between states and tree nodes is immediately clear. In the next subsections we accurately describe the `tree` and the `node` classes we have developed, their fields, and the basic methods implemented to efficiently visit the search tree.

8.2.1 The `Tree` library

In this section we give a brief overview of the C++ library used to create the search tree. See [133] for a deeper description about the library functionalities.

The `tree` class in `tree.hh` is a templated container class in the spirit of the C++ Standard Template Library (STL). The essential difference between a container with the structure of a tree and the STL containers is that the latter are *linear*. Thus, while in the STL containers one has essentially one possible way to iterate over their elements, this is no longer true for trees. The `tree` class organizes the data in form of so-called n-ary trees. In Fig. 8.1 we can see a tree that can be built through this library. Nodes at the same level of the tree are called *siblings*, while nodes that are below a given node and are linked to it through an arc are called its *children*. At the top of the tree, there is a set of nodes characterized by having no parents. The collection of these nodes is called the *head* of the tree. In our work we consider a unique node in the tree head, called the *root* node.

The `tree.hh` library provides four different iteration schemes, giving the possibility to visit the tree nodes in five different orderings. To describe these types of ordering, we consider the tree scheme in Figure 8.2. Possible ways to visit the nodes of this tree are the following:

pre-order: root, A, C, D, B, E, F;

post-order: C, D, A, E, F, B, root;

sibling: for example A, B;

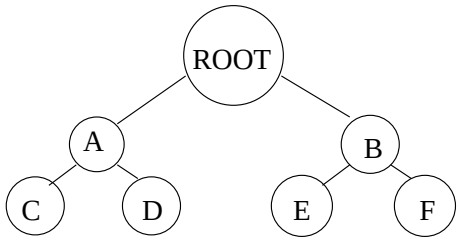

Figure 8.2: Tree scheme

fixed-depth: for example A, B;

leaf: C, D, E, F;

The *pre-order* and *post-order* iterators are the default ones, usually defined as `Iterator`. The *fixed-depth* iterator iterates on all nodes at a fixed depth of the tree, while the *sibling* iterator iterates on the children of a given node on a fixed depth of the tree. Finally, the *leaf* iterator iterates over all leaves (bottom-most) nodes of the tree.

One can also convert an iterator of any type into one of every other kind through a copy constructor. Moreover, a set of very useful methods allow to append child nodes, or to insert a node at a given depth of the tree, or to determine indices in a sibling range. These methods add many functionalities to our algorithms.

8.2.2 The `myNode` class

To maintain a clear structure of the several states in the dynamic programming approach, we chose to directly map each subtour visiting a subset of nodes of the graph G, to each node of the search tree T. In this way, it is easy to maintain a link connecting each state with those generated from it. The `myNode` class contains the following fields:

```
class myNode {
      private:
          bool   removed;
          int    level;
          double cost, profit;
          int    *S;
          int    N;
}
```

The flag `removed` indicates whether the node is dominated or not by other ones. If the node is not dominated, then `removed` is set to `false` and it must be considered in the search of new states, otherwise the flag `removed` is set to `true`: in this case, the node in the tree T is *fathomed* and it will no longer be considered during the search of new solutions belonging to the efficient frontier. The `level` field indicates the node depth in the tree T. The field `S` is a pointer to an integer array containing the list of nodes belonging to the starting graph G visited in the current solution. It is important to notice that the number of nodes contained in the array `S` is equal to the value contained in the `level` field; this fact happens because each level of the search tree corresponds to subtours of the starting graph G with a prefixed length. The `cost` and `profit` values indicate to best reachable values one can get by visiting

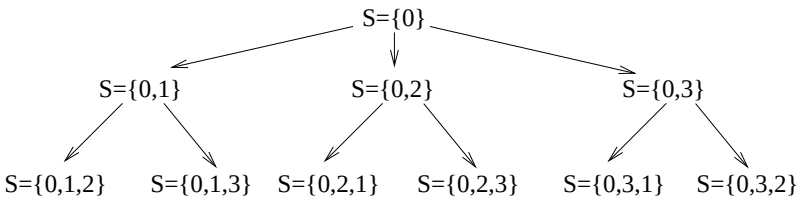

Figure 8.3: field structure `S` for a tree T.

the nodes contained in the `S` array. Finally, `N` represents the total number of nodes in the starting graph G.

The *root* node is the first point generated; its fields are initialized in the following way:

```
removed = false;
level   = 0;
cost    = 0;
profit  = 0;
S       = v[0];
```

where `v[0]` $= v_0$. The root node corresponds to the first efficient point belonging to the Pareto frontier. The other nodes in the tree T are recursively generated from it by the following constructor:

```
myNode(int level, int *Sparent, int N, double profit, double cost)
```

The fields of a node K in the tree T are obtained from the fields of its father K'. Thus, the level of K is computed by adding 1 to the level of K', the `S` array of node K is initialized with the `S` values of K', and the same happens with the `profit` and `cost` fields. The new node represents the best way to visit the subgraph considered by the father node plus a new node not yet visited. Thus, children of a given node in the search tree T cover all nodes of graph G not yet visited by father node. In Fig. 8.3 we show the field `S` in the first three levels of a search tree associated to a complete graph G with $N = 4$. It is easy to note that the field `S` of each node is equal to the field `S` of its father node plus a new node of graph G not yet considered in the subpath.

8.2.3 Handling the comparison among subcycles

The main problem in the implementation of the algorithm is the difficulty to maintain a certain relationship among nodes of the tree T corresponding to the same set of nodes of the graph G. In fact, at each step of the algorithm, when we must compute the cost associated to a new node, we have to find the minimum cost of a path visiting those nodes of the graph G contained in the field `S` of the considered node. To do this, we need to know the costs associated to each subpath visiting all subsets of S, to update their cost with the new added node, and then to compare them to find the cheapest one. This means that we must find, in the previous level of the tree, the nodes of the search tree corresponding to these subsets of S. This operation can take exponential time because the number of non-fathomed nodes in level k can be, in the worst case:

$$n(n-1)(n-2)\cdots(n-k+1).$$

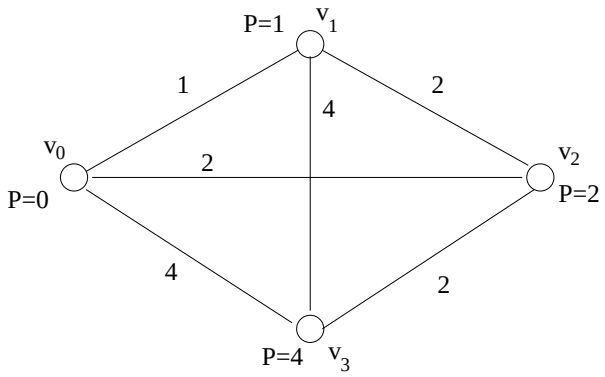

Figure 8.4: the graph G.

Hence, in building the level $(k+1)$ of the tree T, we should perform:

$$(k+1)n(n-1)\cdots(n-k+1)$$

comparisons just to find the nodes corresponding to the subsets of S. The following example shows what happens.

Example

Consider the graph G in Fig. 8.4 with $N = 4$. We proceed with the construction of the tree T associated to the graph G, through the execution of the dynamic programming algorithm of section 8.1.2. The root node of tree T corresponds to $Z(\emptyset, 0, 0)$ and its value is 0. From this node, we can build the first level of the tree T. Nodes of first level correspond to paths in graph G connecting the source to all the other nodes:

- $Z(\{0,1\},1,1) = 1$
- $Z(\{0,2\},2,2) = 2$
- $Z(\{0,3\},4,4) = 4$

The nodes in the second level of the tree can be easily obtained from the nodes of the first level:

- $Z(\{0,1,2\},2,3) = Z(\{0,1\},1,1) + 2 = 3$
- $Z(\{0,1,3\},3,5) = Z(\{0,1\},1,1) + 4 = 5$
- $Z(\{0,2,1\},1,3) = Z(\{0,2\},2,2) + 2 = 4$
- $Z(\{0,2,3\},3,6) = Z(\{0,2\},2,2) + 2 = 4$
- $Z(\{0,3,1\},1,5) = Z(\{0,3\},4,4) + 4 = 8$
- $Z(\{0,3,2\},1,6) = Z(\{0,3\},4,4) + 2 = 6$

We cannot fathom any node at this level: in fact, there are no sets S in Level 1 that are also subsets of S' at Level 2 with the same ending node. At this point the tree T has the structure shown in Fig. 8.5.

To build the third level of the tree T we need to compute the following values of the function Z:

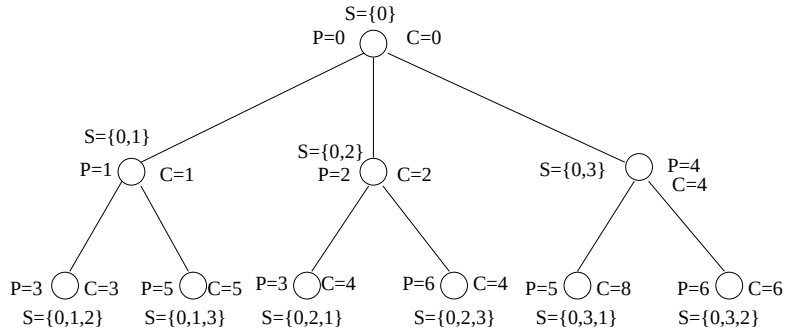

Figure 8.5: Fields of the first two levels in the tree T.

- $Z(\{0,1,2,3\},3,6) = \min\{Z(\{0,1,2\},2,3)+2,\ Z(\{0,2,1\},1,3)+4\} = 5$
- $Z(\{0,1,3,2\},2,6) = \min\{Z(\{0,1,3\},3,5)+2,\ Z(\{0,3,1\},1,5)+4\} = 7$
- $Z(\{0,2,1,3\},3,6) = \min\{Z(\{0,2,1\},1,3)+4,\ Z(\{0,1,2\},2,3)+2\} = 5$
- $Z(\{0,2,3,1\},1,6) = \min\{Z(\{0,2,3\},3,6)+4,\ Z(\{0,3,2\},2,6)+2\} = 8$
- $Z(\{0,3,1,2\},2,6) = \min\{Z(\{0,3,1\},1,5)+2,\ Z(\{0,1,3\},3,5)+2\} = 7$
- $Z(\{0,3,2,1\},1,6) = \min\{Z(\{0,3,2\},2,6)+2,\ Z(\{0,2,3\},3,6)+4\} = 8$

Thus, to compute the Z-value of each node in Level 3 we found the tree nodes in Level 2 corresponding to the subsets of their `S` field, we updated them with the new node and we compared the obtained costs. The final tree is represented in Fig. 8.6.

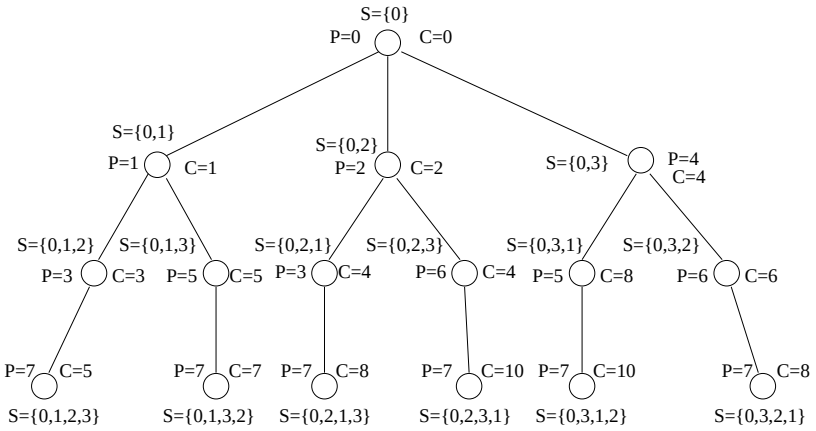

Figure 8.6: Fields of the first three levels in the tree T.

At this point it is possible to compare the set of feasible pairs (cost, profit) computed in Level 3 and the corresponding ones in Level 2, in order to remove those nodes corresponding to dominated solutions. In this specific case, we can remove two nodes in Level 2: in fact, checking the (profit,cost) solutions corresponding to the following S:

- $S = \{0,1,3\}$ compared to $S = \{0,1,2,3\}$;
- $S = \{0,3,1\}$ compared to $S = \{0,3,2,1\}$.

We can notice how a different cost is related to an equal value of the profit. So, it is convenient to choose the path that, visiting the same nodes, gains better results. Even in this small problem instance, a high number of comparison are needed in the several phases of the algorithm. It is clear that they grows exponentially with the size of the problem. To limit them, we developed an approach to generate and classify tree nodes.

In the next section we will describe this approach, showing also the great difference in terms of performances against the basic procedure.

8.2.4 Architectural choices

As we mentioned before, the complexity in the implementation of the algorithm depends, among other things, on the difficulty to find tree nodes corresponding to different permutations of the nodes contained in S. A good improvement of the performances can be obtained if we are able to find such a connection during the tree building phase. So, to keep track of the `S` fields during the construction of the tree, we decided to handle the different permutations of the nodes belonging to a specific path through a matrix, called `PermInd`. The role of this matrix is to make easier the computing of the recursion procedure for each subset S of the graph nodes. Each row of `PermInd` contains pointers to nodes of the search tree T that correspond to paths visiting the same subset S of the graph nodes. The use of this matrix allows to compute the optimal value of $Z(S, j, p)$ in linear time. In fact, it is enough to consider the corresponding row of `PermInd` just once, computing the costs associated to each possible solution and then take the minimum.

A new `PermInd` matrix is allocated at each level of the search tree T and deallocated at the end of its construction. When a new tree node is built, *i.e.*, when the constructor is invoked, the pointer to the node is stored in `PermInd`. This operation takes a large amount of time when the number of children in a given level becomes considerable, but the time required to fill this matrix is anyway shorter than the time needed to search each time the nodes of the tree T corresponding to a subset of the given set S.

The total number of rows of the `PermInd` matrix will be the number of possible different permutations of k nodes in the graph G, where k represents the level of the search tree that must be analyzed. So, in each level k the number of `PermInd` rows will be:

$$\binom{N-1}{k} k$$

where $N-1$ counts all nodes in the graph G except the source. Thus, each row of `PermInd` will contain a pointer to the tree nodes corresponding to permutations of the nodes in subsets of $S \setminus \{j\}$ with ending node v_j, while the number of columns of the `PermInd` matrix will be $k-1$.

The simpler way to fill `PermInd` is the following: when a new tree node is generated, a search in the `PermInd` matrix begins. The first element of each row in `PermInd` is checked and values contained in S relative to the newly created node are compared to the node list S pointed by the entry of the matrix. Then, if the two subsets are different permutations of the same graph nodes with an equal final node, then the pointer to the node is copied in the first free column position of the `PermInd` row selected; otherwise, the search through the matrix continues. If there are no rows containing permutations of the given set of nodes, then the pointer to the node is copied in the first available free row.

We can notice that the time needed to fill the matrix with pointers to the children at level k in the search tree T can be exponential. An upper bound to the number of comparisons

to make before to fill the entire matrix can be computed by assuming to visit each time the entire matrix before to find the exact permutation of the values in S. The maximum number of children that could be created at level k is:

$$\binom{n}{k} k!$$

and so, the maximum number of comparisons becomes:

$$\binom{N-1}{k} kn(n-1)\cdots(n-k+1)$$

that is exponential. It must be observed that a mathematical law that would give an exact connection among the several ways to visit the nodes could avoid the direct search of these correspondences in the `PermInd` matrix, thus reducing the computational time.

We haven't found such a type of law, yet, but we noticed that by organizing the pointers insertions in the `PermInd` matrix, the number of comparisons reduces considerably. For that, we worked on the main characteristic of nodes contained in S, *i.e.* the final index; in fact, in each row of the structure `S` we find the permutation of the same nodes, with the same ending node. Hence, if the graph G contains N nodes, ($N-1$ if we do not count the root), we have at most $N-1$ possible final nodes. So, we decided to sort the `PermInd` rows by the final node of S. In this way, we have pointers to subpaths ending with node 1 in the first rows, followed by pointers to subpaths ending with node 2, and so on. Thus, when we encounter subpaths S ending with node v_k, and we must search `PermInd` rows corresponding to it, we have to look for the possible permutations only in the subset of rows pointing to fields `S` ending with the node v_k. The number of `PermInd` rows corresponding to each possible final node at level z is:

$$\frac{n!}{(n-z)!}\,.$$

Another little trick consists in keeping track of the number of nonzero elements in each row of `PermInd`. This can be important to reduce the time needed to find the exact column position where to copy the pointer once the row is found. So, in place of scanning the elements of each `PermInd` row until the first free position is found, we create a matrix `IndRowCol`, with $N-1$ rows and as many columns as those of `PermInd`: it gives the column position where a new element can be inserted once the row is selected and we update it when it happens.

Moreover, we observed that all children of a given tree node are inserted in the same column of the `PermInd` matrix. This fact derives from a specific rule that correlates the permutation number. We show this in the next example.

Example

Let $G = (V, E)$ be a complete graph with 5 nodes. To build the Pareto-efficient frontier we have to construct and explore the tree T, that at the end will be composed of four levels. Following what is explained in the previous discussion, to each tree level is associated a different `PermInd` matrix, assembled during the initialization of the nodes. In Fig. 8.7 we can see the four `PermInd` matrices generated for a particular instance of the problem.

In each row of the four matrices we can find the S fields of the pointed nodes, that represent the node paths in graph G associated to tree nodes. Thus, each row contains feasible candidates for the efficient frontier. Once `PermInd` is created, the computation of the values of Z can be easily performed by a search of the better path in each row of the matrix.

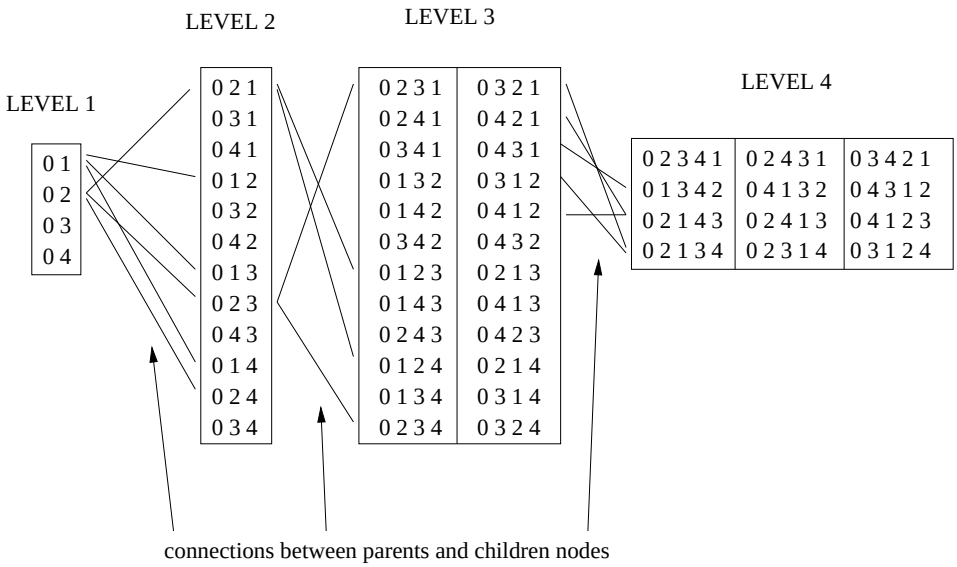

Figure 8.7: Example of `S` fields pointed by the `PermInd` matrix in a graph with 5 nodes.

Let us suppose that the first two levels of the search tree T have been created. Then let us consider the tree node corresponding to the set $S = \{0, 1, 3\}$. The children of this node, stored in the fields `S`, are the following:

- 0, 1, 3, 2;
- 0, 1, 3, 4.

We noticed that, once the column position of the first child in the `PermInd` matrix is found, the position of the other children follows easily. For example, the pointer to the child corresponding to $S = \{0, 1, 3, 4\}$ will be in the same column of the child pointer corresponding to $\{0, 1, 3, 2\}$, but in the row containing permutations ending with node 4. This behaviour can be observed in Fig. 8.7: in the third level of `PermInd`, we can see that children associated to each node of level 2 are always placed in the same column. This considerably reduces the computational time: in fact, if we know that the pointer corresponding to $S = \{0, 1, 3, 4\}$ is in column 2 and that the first w rows corresponding to permutations with final node 4 are already filled, then we can begin the search of the position where to store the new datum from the row $w + 1$.

With these tricks, the computational time is reduced by factor of n.

8.3 Computational Results

To evaluate the performances of the dynamic programming approach, we implemented the algorithm in C++ and we ran it on a AMD Opteron processor at 2.4 GHz.

We chose to analyze randomly generated instances with different numbers of nodes and the smaller instances analyzed in section 6.6.2. This choice is because the CPU time needed to solve a particular instance depends only on the dimension of the starting graph and not on the costs and profits values. This happens because the computations required inside the procedure consist in elementary operations (sums, differences, comparisons) among numerical values, that take constant time, and the number of these operations depends linearly on the

dimension of the graph. Hence, the key factor is not the order of magnitude of costs and profits, but rather the dimension of the graph G.

The first type of instances considered in our tests have the following features:

1. point coordinates are generated in the square $[0, 500] \times [0, 400]$ according to a uniform distribution, and the routing cost are defined by the Euclidean distances among these points;

2. profits are randomly generated in the interval $[1, 50]$.

The same instances are studied in the case of unitary profits. We consider graphs with number of nodes ranging from 10 to 20. In the tables we report the average time to solve the problem instances. We computed the average over 10 instances. In the columns we report the following information:

N: the number of nodes in the graph;

$|S_E|$: the number of computed efficient points;

CPU time: the average CPU time needed to compute the entire Pareto frontier, with the procedures described in section 8.2;

CPU time basic: the average CPU time needed to find the entire Pareto frontier, computed in the simplest way, *i.e.*, without any tricks during the filling of the `PermInd` matrix.

Times are expressed in CPU seconds.

	Unitary Profits		
N	$\|S_E\|$	CPU time	CPU time basic
10	10	0	1.8
12	12	3	7
14	14	95	157
16	16	1815	3583
18	18	21402	40623
20	20	42347	69935

	Random Profits		
N	$\|S_E\|$	CPU time	CPU time basic
10	25	0	2
12	30	3	8
14	40	92	156
16	58	1806	3540
18	80	21315	40750
20	126	42430	70024

In the first table we show the results computed when the profits are unitary, in the second one we solved the same instances with random profits. It is interesting to note the meaningful difference between the time needed to solve the instances when the accesses to the `PermInd` matrix are optimized and when they are not. This is important because it shows how much little devices in the procedure implementation can improve the performances. We can also notice that profits values do not overload the CPU time and thus the computational time

depends only on the cardinality of the graph nodes set. This derives from the structure of the procedure. This is important because it allows to estimate in advance the time needed to find the entire Pareto-efficient frontier for every instance, independently on both the profits and the costs values.

In the following table we report the results related to some TSP library instances analyzed in Chapter 6.

instance	profit type	$\|S_E\|$	CPU time	CPU time basic
burma14	1	14	103	165
	2	59	101	164
	3	70	102	165
ulysses16	1	16	2001	3602
	2	102	2005	3601
	3	92	1998	2999

It is important to see how the computational time remains invariant with all profit sets, even if the performances are considerably worse with respect to those obtained with the branch-and-cut approach. Thus, each improvement to the procedure takes to a constant reduction of the computational time for every instance of the same size.

8.4 Conclusions

In this chapter we presented a dynamic programming approach to find the entire Pareto-efficient frontier for the TSPP. If we analyze the procedure just described from a computational point of view, we can notice that the performances are considerably worse with respect to the branch-and-cut approach. The benefit of this type of method consists in the simplicity of the iteration formulas. In fact, as we showed before, all operations required inside the code are elementary. The long time needed to reach the solution is motivated by the large number of elementary operations to do: thus, the performances are proportional to the number of nodes of the starting graph. Anyway, the time improvement we get by optimizing the data handling is evident and allows us to expect that additional code optimization can further improved the procedure performances.

Moreover, the proposed approach can be modified to be used as a *heuristic approach*. For example, if we drop the dominated nodes at each level of the search tree, comparing only cost and profit fields and not considering the set S, we obtain an upper bound of the Pareto-efficient frontier. On the contrary, if we create tree nodes by always choosing the shortest path in the graph, thus allowing to visit each node more than once, and at the end we drop the cycles inside the paths found, we obtain a lower bound of the Pareto-efficient frontier. In this way, it could be possible to exploit the simplicity of the recursion formula and develop a procedure that find an approximated solution subset in a reasonable time.

Conclusions

In this dissertation we analyzed, under several points of view, the Traveling Salesman Problem with Profits, that is a generalization of the well known Travelling Salesman Problem. The inspiring idea behind this problem was the search of flexible and advanced algorithmic approaches to model increasingly complex network configurations. In particular this problem can be immediately applied to the problem of optimizing the ways to send medical data through a local network (see Chapter 4).

We decided to study the problem from a bi-objective point of view. At the time this work is written, only very few approaches are known to solve the TSPP from a bi-objective point of view, in particular only one work solves exactly the problem, while the remaining literature adresses the topic through different types of heuristic.

We started by giving in the first three Chapters, an overview of the state-of-the-art about multiobjective optimization and on the available resolution approaches for the TSPP. Then, in the chapters from 4 to 8 we analyzed the TSPP under several points of view, developing exact and approximated algorithms.

In particular, in Chapter 4 we describe a feasible application of this problem in the Medical field. In fact, thanks to modern information technology, the need to share medical informations about clinical cases between different hospitals is emerging. For this reason, if we consider the network that links together the various hospitals' departments as a graph, then an optimal solution of the Traveling Salesman Problem with Profits is the best path on which to send informations requests to get the best medical responsed as soon as possible. This type of study could lead to the development of new techniques for consultations and medical diagnoses.

In Chapter 5, the Traveling Salesman Problem with Profits is studied from a bi-objective point of view on graphs with a tree metric. We considered three problems: finding all efficient points; finding all extreme supported efficient points; finding one efficient point, corresponding to a given combination of the two objectives. For each problem, we developed efficient algorithms. Moreover, we analyzed the problem on some simple metrics.

In Chapter 6 we developed an algorithm that compute the entire exact efficient frontier for the TSPP problem using cutting planes inside a branch-and-cut approach. We implemented this algorithm and we analyzed its performances. The feature of our approch is in the way the points are generated. In fact, we compute the efficient frontier in a distributed way, and it can be useful in the development of approaches that search an approximation of the efficient frontier. In the last part of the chapter we introduced some algorithms of this type, built on the same idea of the exact approach, that return an ϵ-approximation of the efficient frontier.

In Chapter 7 we analyzed the complexity of the search of Pareto-efficient solutions for the TSPPTW on simple metrics. In particular we studied how little changes in the starting hypotesys of the problem can significantly change its complexity. For all studied metrics we

developed algorithms that find the exact efficient frontier.

Finally, we presented a Dynamic Programming approach to find the entire Pareto-efficient frontier for the Traveling Salesman Problem with Profits. If we analyze the procedure just described from a computational point of view, we can notice that the performances are considerably worse with respect to the branch-and-cut approach. The benefit of this type of method consists in the simplicity of the iteration formulas. The interest that can arise in this kind of procedure is on its feasible extensions. In fact, the simplicity of the iteration formula lends itself to be used, with some modifications, as a *heuristic approach* or as an internal part of an another procedure.

Bibliography

[1] M.J. Alves and J. Clímaco. Using cutting planes in an interactive reference point approach for multi-objective integer linear programming problems. *European Journal of Operational Research*, 117:565–577, 1999.

[2] C. Archetti, D. Feillet, A. Hertz, and M.G. Speranza. The capacitated team orienteering and profitable tour problems. *Journal of the Operational Research Society*, 60:831–842, 2009.

[3] Br. Arnd, T. Rheinisch-Westfalische, and A. Hochschule. Solving the orienteering problem through branch-and-cut. *INFORMS Journal on Computing*, 1998.

[4] N. Ascheuer, M. Fischetti, and M. Grötschel. Solving asymmetric travelling salesman problem with time windows by branch-and-cut. *Mathematical Programming*, 90:475–506, 2001.

[5] I. Averbakh and O. Berman. A heuristic with worst-case analysis for minimax routing of two traveling salesmen on a tree. *Discrete Applied Mathematics*, 68:17–32, 1996.

[6] B. Awerbuch, Y. Azar, A. Blum, and S. Vempala. New approximation guarantees for minimum-weight k-trees and prize collecting salesmen. *SIAM J. Comput.*, 28(1):254–262, 1998.

[7] A. Bachem and M. Grotschel. New aspects of polyhedral theory. *Modern Applied Mathematics,Optimization and Operations Research*, pages 51–106, 1981.

[8] T. Bagchi, J. Gupta, and C. Sriskandarajah. A review of TSP based approaches for flowshop scheduling. *European Journal of Operational Research*, 169:816–854, 2006.

[9] K.R. Baker. Introduction to sequencing and scheduling. *Addison-Wesley, Reading, MA*, 1974.

[10] K.R. Baker. Elements of sequencing and scheduling. *Amos Tuck School of Business Administration, Dartmouth College, Hanover, NH*, 1992.

[11] E. Balas. The prize collecting traveling salesman problem. *Networks*, 19:621–636, 1989.

[12] E. Balas. On the cycle polytope of a directed graph. *GSIA, Carnegie Mellon University, Pittsburg*, 1993.

[13] E. Balas. The prize collecting traveling salesman problem: II. Polyhedral results. *Networks*, 17:1001–1018, 1995.

[14] E. Balas. New classes of efficiently solvable generalized traveling salesman problems. *Annals of Opererations Research*, 86:529–558, 1999.

[15] E. Balas and G. Martin. Roll-a-round: Software package for scheduling the rounds of a rolling mill. *Balas and Martin Associates*, 1985.

[16] E. Balas and M. Oosten. On the cycle polytope of a directed graph. *Networks*, 36:34–46, 2000.

[17] E. Balas and N. Simonetti. Linear time dynamic programming algorithms for new classes of restricted TSPs: a computational study. *INFORMS Journal on Computing*, 13:56–75, 2001.

[18] M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser. Network routing. *Handbooks in Operations Research and Management Science*, 8, 1995.

[19] P. Bauer. The cycle polytope: Facets. *Mathematics of Operations Research*, 22:110–145, 1997.

[20] R. E. Bellman. Dynamic programming. *Dover Publications*, 2003.

[21] J. Bérubé, M. Gendreau, and J. Potvin. An exact epsilon-constraint method for bi-objective combinatorial optimization problems: Application to the traveling salesman problem with profits. *European Journal of Operational Research*, 194:39–50, 2009.

[22] D. Bienstock, M. Goemans, D. Simchi-Levi, and D. Williamson. A note on the prize collecting traveling salesman problem. *Mathematical Programming*, 59:413–420, 1993.

[23] J. Blazewicz, K. Ecker, G. Schmidt, and J. Weglarz. Scheduling in computer and manufacturing systems. *Springer-Verlag, Berlin*, 1993.

[24] P. Brucker. Scheduling algorithms. *Springer, Berlin, Germany*, 1998.

[25] R. Burkard, M. Dell'Amico, and S. Martello. Assignment problems. *SIAM*, 2009.

[26] A. Campbell and M. Savelsbergh. A decomposition approach for the inventory routing problem. *Transportation Science*, 38:488–502, 2004.

[27] R.L. Carraway, T.L. Morin, and H. Moskovitz. Generalized dynamic programming for multicriteria optimization. *European Journal of Operational Research*, 44:95–104, 1990.

[28] B. Chandran and S. Raghavan. Modeling and solving the capacitated vehicle routing problem on trees. In B. Golden, S. Raghavan, and E. Wasil, editors, *The Vehicle Routing Problem*, pages 239–261. Springer, 2008.

[29] I.M. Chao, B.L. Golden, and E.A. Wasil. A fast and effective heuristic for the orienteering problem. *European Journal of Operational Research*, 88(3):475–489, 1996.

[30] A.A. Chaves and L.A. Nogueira Lorena. Hybrid heuristics with detection of promising areas for the prize collecting travelling salesman problem. *Computer Science*, 4972:123–134, 2008.

[31] K. Chen and S. Har-Peled. The orienteering problem in the plane revisited. *Proceedings of the twenty-second annual symposium on Computational geometry, Sedona, Arizona, USA*, pages 247–254, 2006.

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

[33] J.C.M. Climaco and Martins E.Q.V. A bicriterion shortest path algorithm. *European Journal of Operational Research*, 11:399–404, 1982.

[34] C.A. Coello. List of references on evolutionary multi-objective optimization. *http://www.lania.mx/ ccoello/EMOO/EMOObib.html*, 2000.

[35] S. Coene, F. Spieksma, C. Filippi, and E. Stevanato. The traveling salesman problem on trees: balancing profits and costs. *under submission*, 2009.

[36] S. Coene and F. C. R. Spieksma. Profit-based latency problems on the line. *Operations Research Letters*, 36:333–337, 2007.

[37] E.G. Coffman. Computer and job-shop scheduling theory. *Wiley, New York*, 1976.

[38] E. Coiera. Guida all'informatica medica, internet e telemedicina. *Il Pensiero Scientifico Editore*, 1999.

[39] R.W. Conway, W.L. Maxwell, and L.W. Miller. Theory of scheduling. *Addison-Wesley, Reading, MA*, 1967.

[40] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.

[41] C. Coullard and W.R. Pulleyblank. On cycle cones and polyhedra. *Linear Algebra Applied*, 114/115:613–640, 1989.

[42] P. Cowling. A flexible decision support system for steel hot rolling mill scheduling. *Computers and Industrial Engineering*, 45:307–321, 2003.

[43] P. Czyzak and A. Jaszkiewicz. A multi-objective metaheuristic approach to the localization of a chain of petrol stations by the capital budgeting model. *Control and Cybernetics*, 25:177–187, 1996.

[44] P. Czyzak and A. Jaszkiewicz. Pareto simulated annealing - a metaheuristic technique for multiple objective combinatorial optimization. *Journal of Multi-Criteria Decision Analysis*, 7:34–47, 1998.

[45] H.M. Dathe. Zur lsung des zuordnungsproblems bei zwei zielgr en. zeitschrift fur. *Operations Research*, 22:105–118, 1978.

[46] Deb, Pratap, Agarwal, and Meyarivan. A fast and elitist multi-objective genetic algorithm. *NSGA-II, IEEE Trans Evol Comput.*, 6(2):182–97, 2002.

[47] M. Dell'Amico, F. Maffioli, and A. Sciomachen. A lagrangian heuristic for the prize collecting travelling salesman problem. *Annals of Operations Research*, 81(0):289–306, 1998.

[48] M. Dell'Amico, F. Maffioli, and P. Värbrand. On prize-collecting tours and the asymmetric traveling salesman problem. *International Transactions in Operational Research*, 2:297–308, 1995.

[49] M. Dell'Amico, F. Maffioli, and P. Värbrand. A lagrangian heuristic for the prize-collecting travelling salesman problem. *Annals of Operation Research*, 81:289–305, 1998.

[50] M. DellAmico, F. Maffioli, and S. Martello. Annotated bibliographies in combinatorial optimization. *Wiley, Chichester*, 1997.

[51] J.A. Diaz. Solving multi-objective transportation problems. *Ekonomicko Mathematicky Obzor*, 14:267–274, 1978.

[52] R. Diestel. Graph theory. *Springer-Verlag Heidelberg, new York*, 2005.

[53] Y. Dumas, J. Desrosiers, E. Gelinas, and M. M. Solomon. An optimal algorithm for the travelling salesman problem with time windows. *Operations Research*, 43:367–371, 1995.

[54] P.F. Dutot, L. Eyraud, G. Mounié, and D. Trystram. Bi-criteria algorithm for scheduling jobs on cluster platforms. *SPAA*, pages 125–132, 2004.

[55] T. E. Easterfield. A combinatorial algorithm. *J. London Math Soc.*, 21:219–226, 1946.

[56] M. Ehrgott. Integer solutions of multicriteria network flow problems. *Investigacao Operacional*, 19:229–240, 1999.

[57] M. Ehrgott. Approximation algorithms for combinatorial multicriteria optimization problems. *International Transactions in Operational Research*, 7:5–31, 2000.

[58] M. Ehrgott. *Multicriteria Optimization*. Springer, 2005.

[59] M. Ehrgott and X. Gandibleux. Multi-objective combinatorial optimization. *Ehrgott, M. Gandibleux, X.(Eds.), Multiple Criteria Optimization:State of the Art Annotated Bibliographic Surveys, Kluwer's International Series in Operations Research and Management Science*, 52:369–444, 2002.

[60] A. El-Houssaine, R. Birger, and H. Van Landeghem. Modeling inventory routing problems in supply chains of high consumption products. *European Journal of Operational Research*, 169:1048–1063, 2006.

[61] Emelichev and Perepelitsa. Complexity of vector optimization problems on graphs. *Optimization*, 22:903–918, 1991.

[62] S. Engevall, M. Göthe-Lundgen, and P. Värband. The heterogeneous vehicle-routing game. *Transportation Science*, 38:71–85, 2004.

[63] T. Erlebach, H. Kellerer, and U. Pferschy. Approximating multi-objective knapsack problems. *Management Science*, 48:1603–1612, 2002.

[64] D. Feillet, P. Dejax, and M. Gendreau. The profitable arc tour problem: Solution with a branch-and-price algorithm. *Transportation Science*, 39:539–552, 2005.

[65] D. Feillet, P. Dejax, and M. Gendreau. Traveling salesman problems with profits. *Transportation Science*, 39:188–205, 2005.

[66] M. Fischetti, J.J. Salazar, and P. Toth. Solving the orienteering problem through branch and cut. *Informs Journal on computing, Spring*, 10(2):133–148, 1998.

[67] M. Fischetti and P. Toth. An additive approach for the optimal solution of the prize collecting traveling salesman problem. *In B. L. Golden, A. A. Assad, (eds.) Vehicle Routing: Methods and Studies, Elsevier Science Publishers*, pages 319–343, 1988.

[68] R.J. Gallagher and O.A. Saleh. Constructing the set of efficient objective values in linear multiple objective transportation problems. *European Journal of Operational Research*, 73:150–163, 1994.

[69] X. Gandibleux, N. Mezdaoui, and A. Freville. A tabu search procedure to solve multiobjective combinatorial optimization problems. *In: Caballero R., Ruiz F. and Steuer R. (eds.), Advances in Multiple Objective and Goal Programming, Lecture Notes in Economics and Mathematical Systems, Springer Verlag*, 455:291–300, 1997.

[70] M. Gendreau, A. Hertz, and G. Laporte. A tabu search heuristic for the vehicle routing problem. *Management Science*, 40:1276–1290, 1994.

[71] M. Gendreau, G. Laporte, and F. Semet. A branch-and-cut algorithm for the undirected selective traveling salesman problem. *Networks*, 32:263–273, 1998.

[72] M. Gendreau, G. Laporte, and F. Semet. A tabu search heuristic for the undirected selective traveling salesman problem. *European Journal of Operational Research*, 106:539–545, 1998.

[73] M. Gendreau, J.Y. Potvin, and A. Nabila. An exact algorithm for a single-vehicle routing problem with time windows and multiple routes. *European Journal of Operational Research*, 178:755–766, 2007.

[74] D. H. Gensch. An industrial application of the traveling salesman subtour problem. *AIIE Trans.*, 10(4):362–370, 1978.

[75] A. M. Geoffrion. Proper efficiency the theory of vector maximization. *Journal of Mathematical Analysis and Application*, 2:618–630, 1968.

[76] F. Glover and M. Laguna. Tabu search. *Kluwer Academic Publishers, Boston, MA*, 1997.

[77] F. Glover, E. Taillard, and D. de Werra. A user guide to tabu search. *Annals of Operations Research*, 41:3–28, 1993.

[78] M. Goemans and D. Williamson. General approximation technique for constrained forest problems. *3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 307–315, 1992.

[79] Golberg and Richardson. Genetic algorithm with sharing for multimodal function optimization. *In: Genetic algorithms and their applications: proceedings of the second international conference on genetic algorithms, Camridge, MA, USA:Lawrence Erlbraum Associates*, 1987.

[80] A. Goldberg and R. Tajan. A new approach to the maximum flow problem. *Journal of the Association for Computing Machinery*, 35(4):921–940, 1988.

[81] B. L. Golden, A. Assad, and R. Dahl. Analysis of a large scale vehicle routing problem with an inventory component. *Large Scale Systems*, 7:181–190, 1984.

[82] B. L. Golden, L. Levy, and R. Vohra. The orienteering problem. *Naval Research Logistics*, 34:307–318, 1987.

[83] B.L. Golden, Q. Wang, and Liu. Multifaceted heuristic for the orienteering problem. *Naval Research Logistic*, 35:359–366, 1988.

[84] M. Göthe Lundgen, K. Jornsten, and P. Värbrand. On the nucleolus of the basic vehicle routing game. *Mathematical programming*, 72:83–100, 1996.

[85] M. Göthe-Lundgen, F. Maffioli, and P. Värbrand. A lagrangian decomposition approach for a prize collecting traveling salesman type problem. *Technical Report LiTH-MATH-R-1995-10, Linköping Institute of Technology, Sweden*, 1995.

[86] B. Grunbaum. Convex polytopes. *Wiley, London*, 1967.

[87] H.W. Hamacher and G. Ruhe. On spanning tree problems with multiple objectives. *Annals of Operations Research*, 52:209–230, 1994.

[88] P. Hansen. Bicriterion path problems. *In: Fandel G, Gal T (eds) Multiple criteria decision making theory and application, Lecture Notes in Economics and Mathematical Systems, Springer, Berlin Heidelberg New York*, 177:109–127, 1979.

[89] M. Haouari, Chaouachi J., and Siala. A hybrid lagrangian genetic algorithm for the prize collecting steiner tree problem. *Computers and Operations Research*, 33:1274–1288, 2006.

[90] R. Hartley. Vector optimal routing by dynamic programming. *In: Serafini P (ed.), Mathematics of multi-objective optimization, CISM International Centre for Mechanical Sciences - Courses and Lectures, Springer, Wien*, 289:215–224, 1985.

[91] C. Helmberg. The m-cost atsp. *Lecture Notes Comput. Sci.*, 1610:242–258, 1999.

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

[93] J.H. Holland. Adaptation in natural and artificial systems. *University of Michigan Press, Ann Arbor*, 1975.

[94] J. A. Hoogeveen. Single-machine bicriteria scheduling. Doctoral Thesis, 1992.

[95] J. A. Hoogeveen. Multicriteria scheduling. *European Journal of Operational Research*, 167:592–623, 2005.

[96] J.P. Ignizio. Goal programming and extensions. *Lexington Books, Lexington, KY*, 1976.

[97] H. Isermann. Proper efficiency and the linear vector maximum problem. *Operations Research*, 22:189–191, 1974.

[98] E. Jeannot, É Saule, and D. Trystram. Bi-objective approximation scheme for makespan and reliability optimization on uniform parallel machines. *EuroPar 2008, Las Palmas, Spain*, pages 877–886, 2008.

[99] D. S. Johnson and K. A. Niemi. On knapsacks, partitions, and a new dynamic programming technique for trees. *Mathematics of Operations Research*, 8:1–14, 1983.

[100] N. Jozefowiez, F. Glover, and M. Laguna. Multi-objective meta-heuristics for the traveling salesman problem with profits. *Journal of Mathematical Modeling and Algorithms*, 7:177–195, 2008.

[101] N. Jozefowiez, F. Semet, and E. Talbi. Multi-objective vehicle routing problems. *European Journal of Operational Research*, 189:293–309, 2008.

[102] S. N. Kabadi and A. Punnen. Prize-collecting traveling salesman problem. *INFORMS Washington Conf., Washington, D.C.*, 1996.

[103] Y. Karuno, H. Nagamochi, and T. Ibaraki. Vehicle scheduling on a tree with release and handling times. *Annals of Operations Research*, 69:193–207, 1997.

[104] S. Kataoka and S. Morito. An algorithm for the single constraint maximum collection problem. *J. Oper. Res. Soc. Japan*, 31(4):515–530, 1988.

[105] S. Kataoka, T. Yamada, and S. Morito. Minimum directed 1 subtree relaxation for score orienteering problem. *European Journal of Opererational Research*, 104:139–153, 1998.

[106] R. Keeney and H. Raiffa. Decision with multiple objectives. *New York: Cambridge University Press*, 1993.

[107] C. P. Keller. multi-objective routing through space and time: The mvp and tdvp problems. *Unpublished doctoral dissertation Department of Geography, The University of Western Ontario, London, Ontario, Canada*, 1985.

[108] C. P. Keller. Algorithms to solve the orienteering problem: a comparison. *European Journal of Operational Research*, 41:224–231, 1989.

[109] C. P. Keller and M. Goodchild. The multi-objective vending problem: A generalization of the traveling salesman problem. *Environ. Planning B: Planning Design*, 15:447–460, 1988.

[110] D. Knuth. The art of computer programming: Fundamental algorithms. *Third Edition. Addison-Wesley*, pages 308–423, 1997.

[111] A.A. Konaka, W. David, A. Coitb, and C. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering and System Safety*, 91:992–1007, 2006.

[112] P. Korhonen, S. Salo, and R.E. Steuer. A heuristic for estimating nadir criterion values in multiple objective linear programming. *Operations Research*, 45(5):751–757, 1997.

[113] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 2:83–97, 1955.

[114] M. Labbé, G. Laporte, and H. Mercure. Capacitated vehicle routing on trees. *Operations Research*, 39:616–622, 1991.

[115] G. Laporte and S. Martello. The selective traveling salesman problem. *Discrete Appl. Math*, 26:193–207, 1990.

[116] J.R. Ledesma and J.J. Salazar. The bi-objective travelling purchaser problem. *European Journal of Operational Research*, 160:599–613, 2005.

[117] H. Lee and P.S. Pulat. Bicriteria network flow problems: Integer case. *European Journal of Operational Research*, 66:148–157, 1993.

[118] S.M. Lee. Goal programming for decision analysis. *Auerbach, Philadelphia, PA*, 1972.

[119] A. C. Leifer and M. B. Rosenwein. Strong linear programming relaxations for the orienteering problem. *European Journal of Opererational Research*, 73:517–523, 1994.

[120] A. Lim, F. Wang, and Z. Xu. The capacitated traveling salesman problem with pickups and deliveries on a tree. *Proceedings of ISAAC05, Lecture Notes in Computer Science*, pages 1061–1070, 2005.

[121] S. Lin. Computer solutions of the traveling salesman problem. *Bell System Tech. J.*, 44:2245–2269, 1965.

[122] S. Lin and Kernighan B. W. An effective heuristic algorithm for the traveling salesman problem. *Operation Research*, 21:498–516, 1973.

[123] J. Malczewski and W. Ogryczak. The multiple criteria location problem. a generalized network model and the set of efficient solutions. *Environment and Planning A*, 27:1931–1960, 1995.

[124] A. Martello and P. Toth. Knapsack problems, algorithms and computer implementation. *John Wiley and Sons*, 1990.

[125] A. Mingozzi, L. Bianco, and S. Ricciardelli. Dynamic programming strategies for the travelling salesman problem with time windows and precedence constraints. *Operations Research*, 45:365–377, 1997.

[126] M. Minoux. Solving combinatorial problems with combined min-max-min-sum objective and applications. *Mathematical Programming*, 45:361–372, 1989.

[127] N. Mladenovic and P. Hansen. Variable neighborhood search. *Computers and Operations Research*, 24:1097–1100, 1997.

[128] I. Muslea. The very offline k-vehicle routing problem on trees. *Proceedings of the International Conference of the Chilean Computer Science Society*, pages 155–163, 1997.

[129] G.L. Nemhauser and L.A. Wolsey. Integer and combinatorial optimization. *John Wiley and Sons*, 1999.

[130] C. E. Noon, J. Mittenthal, and R. Pillai. A TSSP plus 1 decomposition strategy for the vehicle routing problem. *European Journal of Operational Research*, 79:524–536, 1994.

[131] I. Osman and G. Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63:513–623, 1996.

[132] C. H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 86–92, 2000.

[133] Kasper Peeters. tree.h documentation. *http://www.damtp.cam.ac.uk/user/kp229/tree/*, 2008.

[134] J. F. Pekny and D.L. Miller. An exact parallel algorithm for the resource constrained travelling salesman problem with application to scheduling with an aggregate deadline. *ACM 18th Annual Comput. Sci. Conf. ACM Press, New York*, pages 208–214, 1990.

[135] G. Pesant, M. Gendreau, and W. Nuijten. A constraint programming framework for local search methods. *Journal of Heuristics*, 5:255–279, 1999.

[136] G. Pesant, M. Gendreau, J.Y. Potvin, and Rousseau J.M. On the flexibility of constraint programming models: From single to multiple time windows for the traveling salesman problem. *European Journal of Operational Research*, 117:253–263, 1999.

[137] P. Pili, R. Scateni, P. Zanarini, and G. Zanetti. Visualizzazione volumetrica in ambiente medico. *Proc. of the Annual AICA Conference*, 1993.

[138] M. Pinedo. Scheduling: Theory, algorithms and systems. *second ed. Prentice-Hall, Englewood Cliffs, NJ*, 2002.

[139] Pires, Antunes, and Martins. A multi-objective model for var planning in radial distribution networks based on tabu search. *IEEE Trans. Power Systems*, 2004.

[140] SY. Prasad. Approximation error analysis in bicriteria heuristics. *Journal of Multi-Criteria Decision Analysis*, 7(3):155–159, 1998.

[141] Przybylski, Gandibleux, and Ehrgott. Two phase algorithms for the bi-objective assignment problem. *European Journal of Operational Research*, 185(2):509–533, 2003.

[142] A.P. Punnen. On combined minmax-minsum optimization. *Computers and Operations Research*, 21(6):707–716, 1994.

[143] R. Ramesh and K.M. Brown. An efficient four-phase heuristic for the generalized orienteering problem. *Comput. Oper. Res.*, 18(2):151–165, 1991.

[144] T. Ramesh, Y.S. Yoon, and M.H. Karwan. An optimal algorithm for the orienteering tour problem. *ORSA Journal on Computing*, 4:155–165, 1992.

[145] Ramos, Alonso, Sicilia, and Gonzales. The problem of the optimal bi-objective spanning tree. *European Journal of Operational Research*, 111:617–628, 1998.

[146] C. Reeves. Modern heuristic techniques for combinatorial problems. *Advanced topics in computer science. McGrawHill, London*, 1995.

[147] G. Reinelt. A traveling salesman problem library. *ORSA Journal of Computing*, 3:376–384, 1991.

[148] J. Ringuest and D. Rinks. Interactive solutions for the linear multi-objective transportation problem. *European Journal of Operational Research*, 32(1):96–106, 1987.

[149] Y. Robert and F. Vivien. Introduction to scheduling. *CRC Press, Chapman and Hall/CRC Computational Science*, 2009.

[150] R.T. Rockafellar. Convex analysis. *Princeton University Press, New Jersey*, 1970.

[151] B. Roy and D. Bouyssou. Aide multicritére á la décision: Méthodes et cas. *Economica, Paris*, 1993.

[152] J.J. Salazar-Gonzalez. On the cycle polytope of an undirected graph. *Working paper,DEIOC, Universidad de La Laguna*, 1994.

[153] N. Samphaiboon and T. Yamada. Heuristic and exact algorithms for the precedence-constrained knapsack problem. *Journal of Optimization Theory and Applications*, 105:659–676, 2000.

[154] N. Samphaiboon and T. Yamada. Heuristic and exact algorithms for the precedence-constrained knapsack problem. *Journal of Optimization Theory and Applications*, 105:659–676, 2000.

[155] J.D. Schaffer. Multiple objective optimization with vector evaluated. *Genetic Algorithms, Ph.D. Dissertation, Vanderbilt University*, 1984.

[156] A. Schrijver. Theory of linear and integer programming. *John Wiley and Sons*, 1986.

[157] D. Schweigert. Linear extensions and vector-valued spanning trees. *Methods of Operations Research*, 60:219–222, 1990.

[158] A. Sedeño Noda and C. González-Martín. An algorithm for the bi-objective integer minimum cost flow problem. *Computers and Operations Research*, 28(2):139–156, 2001.

[159] P. Serafini. Some considerations about computational complexity for multi-objective combinatorial problems. *In: Jahn J, Krabs W (eds) Recent advances and historical development of vector optimization, vol 294. Lecture Notes in Economics and Mathematical Systems. Springer, Berlin Heidelberg New York*, 294, 1986.

[160] P. Serafini. Simulated annealing for multi-objective optimization problems. *In: Proceedings of the 10th International Conference on Multiple Criteria Decision Making, Taipei-Taiwan*, I:87–96, 1992.

[161] A. Sergienko and V.A. Perepelitsa. Finding the set of alternatives in discrete multicriterion problems. *Cybernetics*, 27(3):673 – 683, 1991.

[162] ACM SIGGRAPH. Three-dimensional visualization using medical data. *Course Notes*, 1993.

[163] F. Sourd and O. Spanjaard. A multi-objective branch-and-bound framework: Application to the bi-objective spanning tree problem. *INFORMS: Journal of Computing*, 20(3):472–484, 2008.

[164] V. Srinivasan and G.L. Thompson. An operator theory of parametric programming for the transportation problem I. *Naval Res. Logist. Quart.*, 19:205–226, 1972.

[165] M. F. Tasgetiren. A genetic algorithm with an adaptive penality function for the orienteering problem. *Journal of Economic and Social Research*, 4(2):1–26, 2001.

[166] M.F. Tasgetiren and A.E. Smith. A genetic algorithm for the orienteering problem. *Evolutionary Computation, Proceedings of the 2000 Congress, La Jolia, CA, USA*, 2:910–915, 2000.

[167] P. Toth and D. Vigo. The vehicle routing problem. *SIAM Monographs on Discrete Mathematics and Applications, Philadelphia*, 2002.

[168] T. Tsiligirides. Heuristic methods applied to orienteering. *Journal of the Operational Research Society*, 35:797–809, 1984.

[169] E.L. Ulungu. Optimisation combinatoire multicritere: Determination de l'ensembledes solutions efficaces et methodes interactives. *PhD thesis, Univesite' de Mons-Hainault, Faculte de Sciences*, 1993.

[170] E.L. Ulungu and J. Teghem. Heuristic for multi-objective combinatorial optimization problems with simulated annealing. *Presented at the EURO XII Conference, Helsinki*, 1992.

[171] E.L. Ulungu and J. Teghem. Application of the two phases method to solve the bi-objective knapsack problem. *Technical report, Faculte' Polytechnique de Mons, Belgium*, 1994.

[172] E.L. Ulungu and J. Teghem. The two-phases method: An efficient procedure to solve bi-objective combinatorial optimization problems. *Foundations of Computing and Decision Sciences*, 20(2):149–165, 1994.

[173] E.L. Ulungu and J. Teghem. Solving multi-objective knapsack problem by a branch-and-bound procedure. *In: Climaco J (ed) Multicriteria analysis, Springer, Berlin Heidelberg New York*, pages 269–278, 1997.

[174] J.H. van Bemmel and M. A. Musen. Handbook of medical informatics. *Springer*, 1997.

[175] M. Visee, J. Teghem, M. Pirlot, and E.L. Ulungu. Two-phases method and branch-and-bound procedures to solve the bi-obective knapsack problem. *Journal of Global Optimization*, 12:139–155, 1998.

[176] Q. Wang, X. Sun, B.L. Golden, and J. Jia. Using artificial neural networks to solve the orienteering problem. *Annals of Operations Research*, 61:111–120, 1995.

[177] G. J. Woeginger. When does a dynamic programming formulation guarantee the existence of a fully polynomial time approximation scheme (FPTAS)? *INFORMS Journal on Computing*, 12:57–74, 1999.

[178] P.L. Yu. Multiple criteria decision making: concepts, techniques and extensions. *Plenum Press, New York, NY*, 1985.

[179] A. L. Yuille and J. J. Kosowsky. Statistical physics algorithms that converge. *Neural Computation*, 6(3):341–356, 1994.

[180] E. Zitzler and L. Thiele. multi-objective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Trans. Evol. Comput.*, 3(4):257–71, 1999.