Università degli studi di Firenze

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCES DIMAI
Master's degree in Applied Mathematics

Master's thesis

The Quadratic Assignment Problem

Metaheuristic approaches

Candidate:
Tommaso Mannelli Mazzoli
Matricola 6462050

Thesis advisor:
Stefania Bellavia
Università degli studi di Firenze

Research supervisor:
Angel Felipe Ortega
Universidad Complutense de Madrid

Thesis submitted in 2020

COLOPHON

This document was typeset using LaTeX document processing system originally developed by Leslie Lamport, based on TeX typesetting system created by Donald Knuth.

The typographic package `classicthesis` was used. The bibliography was processed by `Biblatex`.

The LaTeX code of this document can be found on GitHub at `https://github.com/Tommaso-Mannelli-Mazzoli/masters-thesis`.

To Michelangelo.

Abstract

This thesis deals with the Quadratic Assignment Problem (QAP).

The QAP is an **NP**-hard combinatorial optimization problem.

The goal of this thesis is to describe the problem, some of its reformulations and applications and to investigate a number of heuristic and metaheuristic methods.

We implemented greedy and local search heuristic algorithms, then, we use them to construct more advanced metaheuristic methods which provide better solutions. We studied and implemented metaheuristics such as Ant Colony Optimization, Tabu search, Variable Neighborhood Search.

These methods have been implemented in Fortran language and the codes have been made available in the GitHub repository. Numerical results obtained on several instances from the QAPLIB library are also shown.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

INTRODUCTION

This dissertation deals with the numerical solution of the Quadratic Assignment Problem (QAP). It is one of the most studied and complex problem in the field of optimization, and it has lots of applications. The first formulation of the problem was introduced by Koopmans and Beckmann [23] in 1957 and can described as follows:

> Given n facilities and n possible locations, one wants to assign each facility to one location in order to minimize a prescribed cost function. This cost function depends on the known flow f_{ij} from facility i to facility j and on the distance d_{rs} between location r and location s.

The goal of the thesis is to describe the problem, its formulations, to implement heuristic and metaheuristic algorithms to obtain an approximated solution and to compare them.

A first reformulation of the problem employing permutation is the following. Consider the locations set as a vector

$$v = (1, 2, \dots, n),$$

therefore the solution is a permutation of the entries of v. That is, it is a permutation $\pi\colon \{1, \dots, n\} \to \{1, \dots, n\}$ such that $\pi(i) = r$ means to assign the facility i to location r.

We will use *one-note* notation. Thus, π will be denoted as follows:

$$\pi = \big[\pi(1), \pi(2), \dots, \pi(n)\big].$$

This problem, even if it does not look so difficult at first sight, is actually pretty hard. First of all the optimal solution we are looking for is integer, and the exact algorithms (i.e. algorithms designed to provide an optimal solution) are extremely expensive for large scale problems. Moreover, the problem is not linear (as the name suggests, is quadratic). Therefore, for $n > 30$ the computational time of exact algorithms is prohibitive [5, p. 210].

There, we chose to follow a *metaheuristic* approach. Heuristic algorithms do not guarantee to find an optimal solution and generally they return a sub-optimal solution. However, they are problem-dependent and may get trapped in local optima. Note that this is a trouble, since our goal is to achieve a global optimum.

As stated in the book [16, p. ix], metaheuristics are "solution methods that orchestrate an interaction between local improvement procedures and higher level strategies to create a process capable of escaping from local optima and performing a robust search of a solution space". Metaheuristic algorithms are less problem dependent than heuristic methods and usually reach a sub-optimal solution with a reasonable computational cost, but it is not possible to assess the quality of the provided approximation due to the lack of an optimality measure.

Finally, note that the QAP is **NP**-hard [30, Theorem 2.1].

DOCUMENT ORGANIZATION The remainder of this document is organized in the following manner:

- Chapter 1 provides background knowledge on the Quadratic Assignment Problem. We describe the original problem, two variants and many equivalent formulations.
- Chapter 2 describes some applications of QAP to the real world. The hospital Layout problem is described in Section 2.1, while the problem of organizing guests around a table is discussed in Section 2.2. The choice of assigning letters in a keyboard presented in Section 2.4. Finally, in Section 2.5 we give an overview of the problem of arranging 20 numbers around a dartboard, which can be expressed as a QAP instance.
- Chapter 3 introduces heuristic algorithms. In Section 3.1 we give preliminary definitions of neighborhood of a permutation; then, we study and compare local search algorithms, used to improve the current solution in order to obtain a local optimum. In Section 3.2 constructive algorithms are described and compared.
- Chapter 4 introduces metaheuristic algorithms. In Section 4.1 the Tabu Search algorithm is described, in Section 4.2 the "bio-inspired" Ant Colony Optimization algorithm is studied, while in Section 4.3 the Variable Neighborhood Search algorithm is analyzed.
- In Chapter 5 a brief introduction of the instances used is presented to evaluate the performance of metaheuristic algorithms presented in Chapter 4. Then, the performance of such methods are discussed.
- In Chapter 6 we propose a few possible enhancements and future developments. Finally, we sum up what we achieved with this work.

NOTATIONS

In this text vectors and matrices are denoted by boldface, italic symbols (like $\boldsymbol{w}$ and $\boldsymbol{M}$) while sets by capital italic (like I). The following notation is used through the text.

Simbol	Meaning	Note
I	generic set	
$\lvert I \rvert$	cardinality of set I	
$\mathbb{N}$	set of natural numbers	$\mathbb{N} = \{1, 2, \dots\}$
$[n]$	set of first n natural numbers	$[n] = \{1, 2, 3, \dots, n\}$
π	permutation	
S_n	set of permutation of n elements	
s.t.	subject to	restrictions of the problem
w.r.t.	with respect to	
PD	percentage deviation	
$\boldsymbol{v}$	vector	
$\boldsymbol{A}$	matrix	
$\boldsymbol{A}^{\mathrm{T}}$	transpose of a matrix	
$\langle \boldsymbol{A}, \boldsymbol{B} \rangle$	inner product	
$\boldsymbol{A} \otimes \boldsymbol{B}$	Kronecker product	
$\operatorname{tr} \boldsymbol{A}$	trace of the matrix $\boldsymbol{A}$	$\operatorname{tr} \boldsymbol{A} = \sum_{i=1}^{n} a_{ii}$

PREFACE

This thesis was written within the Agreement between Universidad Complutense de Madrid (Madrid, Spain) and Università degli Studi di Firenze (Florence, Italy). As prescribed by the agreement, I was a visiting scholar for a period of 6 months, from September 2019 to February 2020.

During this period, I approached the field of Operative Research. I had the opportunity to study in Prof. Angel Felipe Ortega's *Advanced Optimization Techniques* course at UCM. Within this course, I became interested in heuristic methods and Prof. Ortega agreed to supervise my Master's thesis project on Quadratic Assignment Problem.

My thesis advisor at University of Florence is Prof. Stefania Bellavia. She helped me to place my work in a broader context, making it more organic and to improve the presentation, with a constant, scrupulous and meticulous *labor limae*.

I would like to thank Prof. Bellavia and Prof. Ortega for their invaluable guidance, for their dedication and their patience.

Part I

THEORY

1 THE QUADRATIC ASSIGNMENT PROBLEM

In this chapter we will describe formulations and variants of the Quadratic Assignment Problem. There are several equivalent formulations of the QAP, each one exploiting a different feature of the structure of the problem. Different formulations lead to different solution approaches. Other formulations and variants can be found in [5, Ch. 7].

1.1 DESCRIPTION AND FORMULATIONS

A set of n facilities has to be allocated to a set of n locations. We are given three matrices:

- $\boldsymbol{F} \in \mathbb{R}^{n\times n}$, where $\boldsymbol{F} = (f_{ij})$ and f_{ij} is the *flow* between facility i and facility j. Therefore, f_{ij} could be viewed as the amount of supplies transported between the two facilities. Hence, it is the cost per unit length.
- $\boldsymbol{D} \in \mathbb{R}^{n\times n}$, where $\boldsymbol{D} = (d_{rs})$ and d_{rs} is the *distance* between location r and location s. Note that d_{rs} is a length.
- $\boldsymbol{C} \in \mathbb{R}^{n\times n}$, where $\boldsymbol{C} = (c_{ir})$ and c_{ir} is the *cost* of placing facility i at location r. Note that $c_{i\pi(i)}$ is a cost.

1.1.1 *Combinatorial formulation*

The feasible set is the set of permutation of n elements S_n.

If $\pi \in S_n$, the product $f_{ij}\, d_{\pi(i)\pi(j)}$ is the transportation cost associated to assigning facility i to location $\pi(i)$ and facility j to location $\pi(j)$. That is, the transportation cost is given by the product flow times distances.

Each term $c_{i\pi(i)} + \sum_{j=1}^{n} f_{ij} d_{\pi(i)\pi(j)}$ represents the total cost, related to facility i given by the cost for installing it at location $\pi(i)$ plus the transportation costs to all facilities j, if installed at locations $\pi(1), \pi(2), \ldots, \pi(n)$.

Hence, the QAP can be written as

Combinatorial formulation

$$\min_{\pi\in S_n} \left[\sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} d_{\pi(i)\pi(j)} + \sum_{i=1}^{n} c_{i\pi(i)} \right] \tag{1}$$

In future, $z(\pi)$ will denote the quadratic term of the objective function

$$z(\pi) = \sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} d_{\pi(i)\pi(j)}. \tag{2}$$

An instance of the QAP with input matrices $\boldsymbol{F}, \boldsymbol{D}, \boldsymbol{C}$ is denoted by $QAP(\boldsymbol{F}, \boldsymbol{D}, \boldsymbol{C})$. If there is no linear term (hence, $\boldsymbol{C}$ is the null matrix), we just write $QAP(\boldsymbol{F}, \boldsymbol{D})$.

1.1.2 *Lawler's general formulation*

A more general version of the QAP was considered by Lawler [25], who introduced a four-index cost array[1] $\mathsf{K} = (k_{irjs})$ instead of the three matrices $\boldsymbol{F}$, $\boldsymbol{D}$ and $\boldsymbol{C}$. The relationship between the three matrices and K is the following:

$$k_{irjs} = f_{ij}d_{rs} \qquad \text{for } i \neq j \text{ or } r \neq s, \tag{3}$$

$$k_{irir} = f_{ii}d_{rr} + c_{ir} \qquad \text{for } i, r \in \{1, \dots, n\}. \tag{4}$$

Moreover, note that the objective function can be re-written as follows:

$$\begin{aligned}\sum_{i=1}^{n}\left(f_{ii}d_{\pi(i)\pi(i)} + c_{i\pi(i)} + \sum_{\substack{j=1\\ j\neq i}}^{n} f_{ij}d_{\pi(i)\pi(j)}\right) &= \sum_{i=1}^{n}\left(k_{i\pi(i)i\pi(i)} + \sum_{\substack{j=1\\ j\neq i}}^{n} k_{i\pi(i)j\pi(j)}\right)\\ &= \sum_{i=1}^{n}\left(\sum_{j=1}^{n} k_{i\pi(i)j\pi(j)}\right).\end{aligned}$$

According to this notation, the general form of the QAP is

Lawler's general formulation

$$\min_{\pi\in S_n}\left[\sum_{i=1}^{n}\sum_{j=1}^{n} k_{i\pi(i)j\pi(j)}\right] \tag{5}$$

The combinatorial formulation (1) is the most known, but there are others, totally equivalent, that we are going to describe.

1.1.3 *Algebraic formulation*

An other formulation is the algebraic one, which contains binary variables.

Let x_{ir} such that

$$x_{ir} = \begin{cases} 1 & \text{if the facility } i \text{ is assigned to location } r \\ 0 & \text{otherwise.}\end{cases}$$

Hence the QAP can be formulated as

Algebraic formulation

$$\begin{aligned}\min\ & \left[\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{r=1}^{n}\sum_{s=1}^{n} f_{ij}d_{rs}x_{ir}x_{js} + \sum_{i=1}^{n}\sum_{r=1}^{n} c_{ir}x_{ir}\right]\\ \text{s.t}\quad & \sum_{i=1}^{n} x_{ir} = 1 \qquad \forall r \in \{1, \dots, n\}\\ & \sum_{r=1}^{n} x_{ir} = 1 \qquad \forall i \in \{1, \dots, n\}\\ & x_{ir} \in \{0, 1\} \qquad \forall i, j \in \{1, \dots, n\}\end{aligned} \tag{6}$$

Using Lawler's general form (5) this formulation can be written as

[1] We denoted the 4-dimensional array K by sans-serif font in order to distinguish it from 2-dimensional matrices.

Algebraic formulation in Lawler's form

$$\begin{aligned} \min \quad & \left[\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{r=1}^{n} \sum_{s=1}^{n} k_{irjs} x_{ir} x_{js} \right] \\ \text{s.t} \quad & \sum_{i=1}^{n} x_{ir} = 1 \qquad \forall r \in \{1, \dots, n\} \\ & \sum_{r=1}^{n} x_{ir} = 1 \qquad \forall i \in \{1, \dots, n\} \\ & x_{ir} \in \{0, 1\} \qquad \forall i, j \in \{1, \dots, n\} \end{aligned} \tag{7}$$

1.1.4 *Inner product formulation*

The combinatorial formulation (1) can be written in a more compact way using the inner product of permutation matrices.

Definition 1.1 (Permutation Matrix)**.** Let $n \in \mathbb{N}$ and $\boldsymbol{P}$ be an $n \times n$ binary matrix. $\boldsymbol{P}$ is called a *permutation matrix* if

$$\sum_{i=1}^{n} p_{ij} = 1 \text{ for every } j = 1, \dots, n \quad \text{and} \quad \sum_{j=1}^{n} p_{ij} = 1 \text{ for every } i = 1, \dots, n^2.$$

Therefore, every permutation matrix is a matrix with one (and only one) 1 for every row and column. This suggest its name: for every permutation $\pi \in S_n$ exists one and only one permutation matrix $\boldsymbol{X}_\pi = (x_{ir})$ such that:

$$x_{ir} = \begin{cases} 1 & \text{if } \pi(i) = r; \\ 0 & \text{otherwise.} \end{cases}$$

For example, if $n = 4$ the permutation $\pi = [4, 2, 3, 1]$ is associated to the permutation matrix:

$$\boldsymbol{X}_\pi = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}. \tag{8}$$

Hence, given a vector $\boldsymbol{v} \in \mathbb{R}^n$, the vector $\boldsymbol{w} := \boldsymbol{X}_\pi \cdot \boldsymbol{v}$ is obtained permuting the entries of $\boldsymbol{v}$ according to the permutation π, i.e.,

$$w_i \;=\; v_{\pi(i)} \quad \forall i \in \{1, \dots, n\}.$$

In fact, if we consider the permutation matrix described in (8), we obtain

$$\boldsymbol{X}_\pi \cdot \boldsymbol{v} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} v_4 \\ v_2 \\ v_3 \\ v_1 \end{pmatrix}.$$

If $\boldsymbol{A}$ and $\boldsymbol{B}$ are two matrices $n \times n$, we can define their inner product $\langle \boldsymbol{A}, \boldsymbol{B} \rangle$.

[2] Hence the matrix $\boldsymbol{X} = (x_{ir})$ in equation (6) is a permutation matrix.

Definition 1.2 (Inner product of two matrices)**.** Let $\boldsymbol{A} = (a_{ij})$ and $\boldsymbol{B} = (b_{ij})$ be two real matrices $n \times n$. We define the *inner product*[3] of $\boldsymbol{A}$ and $\boldsymbol{B}$ as the real number defined by

$$\langle \boldsymbol{A}, \boldsymbol{B} \rangle := \sum_{r=1}^{n} \sum_{s=1}^{n} a_{rs} b_{rs}.$$

Letting $\boldsymbol{X}_\pi$ be a permutation matrix and x_{ij} its entry (i, j), we note that

$$\begin{aligned} \left(\boldsymbol{X}_\pi \boldsymbol{D} \boldsymbol{X}_\pi^{\mathrm{T}} \right)_{ij} &= \sum_{r=1}^{n} x_{ir} \left(\boldsymbol{D} \boldsymbol{X}_\pi^{\mathrm{T}} \right)_{rj} \\ &= \sum_{r=1}^{n} x_{ir} \left(\sum_{s=1}^{n} d_{rs} x_{js} \right) \\ &= \sum_{r=1}^{n} \sum_{s=1}^{n} x_{ir} d_{rs} x_{js}. \end{aligned} \tag{9}$$

By (6) and (9) it follows that the QAP can be rewritten as

Inner product formulation

$$\begin{aligned} \min \ & \left[\langle \boldsymbol{F}, \boldsymbol{X} \boldsymbol{D} \boldsymbol{X}^{\mathrm{T}} \rangle + \langle \boldsymbol{C}, \boldsymbol{X} \rangle \right] \\ \text{s.t} \quad & \boldsymbol{X} \text{ is a permutation matrix} \end{aligned} \tag{10}$$

1.1.5 *Trace formulation*

Definition 1.3 (Trace of a Matrix)**.** Let $\boldsymbol{A} = (a_{ij})_{ij}$ be a matrix $n \times n$. We define *trace* of $\boldsymbol{A}$ as the real number given by the sum of its diagonal elements:

$$\operatorname{tr} \boldsymbol{A} = \sum_{i=1}^{n} a_{ii}$$

Proposition 1.1. *Let $\boldsymbol{A}$, $\boldsymbol{B}$ be two $n \times n$ matrices, then we get some simple properties of the trace:*

1. $\operatorname{tr}(\boldsymbol{A} + \boldsymbol{B}) = \operatorname{tr} \boldsymbol{A} + \operatorname{tr} \boldsymbol{B}$;
2. $\operatorname{tr} \boldsymbol{A}^{\mathrm{T}} = \operatorname{tr} \boldsymbol{A}$;
3. $\operatorname{tr}(\boldsymbol{A}\boldsymbol{B}) = \operatorname{tr}(\boldsymbol{A}^{\mathrm{T}} \boldsymbol{B}^{\mathrm{T}})$;
4. $\langle \boldsymbol{A}, \boldsymbol{B} \rangle = \operatorname{tr}(\boldsymbol{A}^{\mathrm{T}} \boldsymbol{B})$.

We can rewrite (10) using the trace operator, in fact

$$\begin{aligned} \langle \boldsymbol{F}, \boldsymbol{X} \boldsymbol{D} \boldsymbol{X}^{\mathrm{T}} \rangle + \langle \boldsymbol{C}, \boldsymbol{X} \rangle &\overset{4}{=} \operatorname{tr}\left(\boldsymbol{F}^{\mathrm{T}} \boldsymbol{X} \boldsymbol{B} \boldsymbol{X}^{\mathrm{T}} \right) + \operatorname{tr}\left(\boldsymbol{C}^{\mathrm{T}} \boldsymbol{X} \right) \\ &\overset{3}{=} \operatorname{tr}\left(\boldsymbol{F} \boldsymbol{X} \boldsymbol{D}^{\mathrm{T}} \boldsymbol{X}^{\mathrm{T}} \right) + \operatorname{tr}\left(\boldsymbol{C} \boldsymbol{X}^{\mathrm{T}} \right) \\ &\overset{1}{=} \operatorname{tr}\left(\boldsymbol{F} \boldsymbol{X} \boldsymbol{D}^{\mathrm{T}} \boldsymbol{X}^{\mathrm{T}} + \boldsymbol{C} \boldsymbol{X}^{\mathrm{T}} \right) \\ &= \operatorname{tr}\left((\boldsymbol{F} \boldsymbol{X} \boldsymbol{D}^{\mathrm{T}} + \boldsymbol{C}) \boldsymbol{X}^{\mathrm{T}} \right) \end{aligned}$$

[3] Sometimes in literature this operation is called *Hadamard product*, from the French mathematician Jacques Hadamard.

Therefore the equation (10) can be rewritten as

Trace formulation

$$\begin{aligned} &\min \left[\operatorname{tr} \left((\boldsymbol{F}^{\mathrm{T}} \boldsymbol{X} \boldsymbol{D} + \boldsymbol{C}) \boldsymbol{X}^{\mathrm{T}} \right) \right] \\ &\text{s.t.} \quad \boldsymbol{X} \text{ is a permutation matrix.} \end{aligned} \tag{11}$$

The trace formulation of the QAP was used by Finke, Burkard e Rendl [14] to introduce eigenvalue bounds for QAP.

1.1.6 *Kronecker product formulation*

A further reformulation can be observed exploiting the Kronecker product of two matrices.

Definition 1.4 (Kronecker product)**.** Let $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{B} \in \mathbb{R}^{r \times s}$ two matrices. We define the Kronecker product $\boldsymbol{A} \otimes \boldsymbol{B} \in \mathbb{R}^{mr \times ns}$ as the matrix formed by all possible products $a_{ij} b_{hk}$:

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{pmatrix} a_{11}B & a_{12}B & \dots & a_{1n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \dots & a_{mn}B \end{pmatrix}.$$

Now, let $\boldsymbol{X}$ be a permutation matrix. We can consider the four-index cost array K introduced in equation (3), as "matrix of matrices", so every $n \times n$ matrix $\boldsymbol{K}^{ir}$ is formed by the elements k_{irjs} with fixed indices i and r and variable indices $j, s = 1, 2, \dots, n$. Using this notation we get

$$\begin{aligned} \langle \mathsf{K}, \boldsymbol{X} \otimes \boldsymbol{X} \rangle &= \left\langle \begin{bmatrix} \boldsymbol{K}^{11} & \dots & \boldsymbol{K}^{1n} \\ \vdots & \ddots & \vdots \\ \boldsymbol{K}^{n1} & \dots & \boldsymbol{K}^{nn} \end{bmatrix} \begin{bmatrix} x_{11}\boldsymbol{X} & \dots & x_{1n}\boldsymbol{X} \\ \vdots & \ddots & \vdots \\ x_{n1}\boldsymbol{X} & \dots & x_{nn}\boldsymbol{X} \end{bmatrix} \right\rangle \\ &= \sum_{i=1}^{n} \sum_{r=1}^{n} x_{ir} \langle \boldsymbol{K}^{ir}, \boldsymbol{X} \rangle \\ &= \sum_{i=1}^{n} \sum_{r=1}^{n} \sum_{j=1}^{n} \sum_{s=1}^{n} k_{irjs} x_{ir} x_{js}, \end{aligned}$$

which is the objective function from (7). This lead us to the Kronecker product formulation of the QAP.

Kronecker product formulation

$$\begin{aligned} &\min \langle \mathsf{K}, \mathsf{Y} \rangle \\ &\text{s.t.} \quad \mathsf{Y} = \boldsymbol{X} \otimes \boldsymbol{X} \\ &\qquad \boldsymbol{X} \text{ is a permutation matrix} \end{aligned} \tag{12}$$

1.2 VARIANTS

There are some variants in literature. We are going to show some of the principal ones. Many others can be found in [5, Ch. 9].

1.2.1 *QBAP*

The most known variants of the QAP is the *Quadratic Bottleneck Assignment Problem* (QBAP), that is obtained by replacing the sums in the objective function of a QAP with the maximum operator.

The QBAP can be formulated as

$$\min_{\pi \in S_n} \left[\max \left(\max_{1 \leq i,j \leq n} f_{ij} d_{\pi(i)\pi(j)}, \max_{1 \leq i \leq n} c_{i\pi(i)} \right) \right]. \tag{13}$$

Basically all QAP applications give rise to a QBAP model as well, because it often makes sense to minimize the largest cost instead of the overall cost incurred by some decisions.

1.2.2 *Quadratic semi-assignment problem*

The *quadratic semi-assignment problem* (semi-QAP) has the same objective function as the QAP but allows the solution not to be permutations: they map the set of integers $N = \{1, 2, \ldots, n\}$ to the set $M = \{1, 2, \ldots, m\}$ with $n > m$ (so there are more facilities than locations and there is no limit to the number of facilities we can assign to the same location)

We can write the semi-QAP in the algebraic form as

$$\begin{aligned} \min \quad & \left[\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{r=1}^{m} \sum_{s=1}^{m} f_{ij} d_{rs} x_{ir} x_{js} + \sum_{i=1}^{n} \sum_{r=1}^{m} c_{ir} x_{ir} \right] \\ \text{s.t} \quad & \sum_{r=1}^{m} x_{ir} = 1 \quad i = 1, \ldots, n \\ & x_{ir} \in \{0, 1\} \quad i = 1, 2, \ldots, n \quad j = 1, 2, \ldots, m \end{aligned} \tag{14}$$

The semi-QAP is **NP**-hard, as an instance of QAP($\boldsymbol{F}, \boldsymbol{D}, \boldsymbol{C}$) can always be transformed into an equivalent instance of semi-QAP by adding slack variables.

From now on, the matrix $\boldsymbol{C}$ will be the null matrix, hence, no linear term will appear in the objective function.

2 APPLICATIONS

As the name said, the Quadratic Assignment Problem was firstly studied to solve the problem involving assignment of n facilities to n locations. Nevertheless. QAP appears in several seemingly unrelated decision problems such as keyboard design [4], scheduling [17], arrangements of micro array chips [7], numerical analysis [3], forest park management [2], Traveling Salesman Problem (TSP) [30, 8] and so on. In this chapter we describe some of these applications; many others can be found on [5].

2.1 HOSPITAL LAYOUT

In 1975 the Ahmed Mahe Hospital of Cairo (Egypt) was composed by six major departments. One of them (the Outpatient) is formed by 19 clinics, listed in table 1.

The problem was to find the optimal layout of the department minimizing the total distance traveled by patients. Alwalid Elshafei [12] modeled this as a QAP. This problem has dimension $n = 19$.

The problem is symmetric, since every patient must return to the first clinic he visited to mark off his card.

The distance matrix $\boldsymbol{D} = (d_{ij})$ contains the distances between the clinics i and j. The flow matrix $\boldsymbol{F} = (f_{ij})$ contains flows between clinics r and s on a yearly basis.

The highest flow is 76687: between Receiving and Recording and General Practitioner. The second one (40951) is between General Practitioner and Pharmacy. The third one (13732) is between Receiving and Recording and Dental clinic. Moreover, the flow between facilities 15 and 16 was set 99999, to force them to be in two adjacent locations in the final solution.

The distances between locations were measured by tracing the paths taken by patients while moving from a clinic to another. Whenever the movement involved a change in floors, the corresponding vertical distance was multiplied by 3 [12].

Table 1: Facilities of Outpatient department.

Facility	Clinic	Facility	Clinic
1	Receiving and Recording	11	X-Ray
2	General Practitioner	12	Orthopedic
3	Pharmacy	13	Psychiatric
4	Gynecological & Obstetric	14	Squint
5	Medicine	15	Minor Operations
6	Pediatric	16	Minor Operations
7	Surgery	17	Dental
8	Ear, Nose & Throat	18	Dental Surgery
9	Urology	19	Dental Prosthetic
10	Laboratory		

Elshafei and Bazaraa eliminated nearly 20% of unnecessary traffic by patients, resulting in an overall more effective treatment center. As a result, their findings were implemented in a new layout of the department [12].

In 1978 Krarup [24] described a related problem for Regensburg Clinic, in Germany. In 2015 Feng and Su [13] applied this model to the Tongji hospital, Shanghai (China), reducing the average walking time for the outpatients by 11.55%. In 2016 Helbert et al [21] followed a similar approach for Hannover Medical School, Germany.

2.2 WEDDING BANQUET

In 1970 Muller [29] described the following situations: we want to arrange n wedding guests around a table minimizing the *annoyance* between them or, if we prefer, maximizing the total pleasantness.

We know the distances d_{ij} between seat i and j. We can consider any form of tables (rectangular, round, ...) or sets of tables. The only thing that matters is knowing the relative distance between seats.
As regards guests, we consider the intensity of relationship f_{rs} between every guest r and s. High values of f_{rs} correspond to a good relationship.

The value of f_{rs} can be related to several characteristics: age, relationship status, interest groups, degrees of acquaintance, sympathy. For example, the author (which has not yet organized any wedding) thinks that kids should be placed together, and near their parents. Note that the spouses should know how good the relationships between guests are. Finally, note that in general $f_{rs} \neq f_{sr}$.

The wedding banquet problem can be described as follows:

$$\min_{\pi \in S_n} \left[\sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\pi(i)\pi(j)} \right]. \tag{15}$$

As we can see, this is the combinatorial form of the QAP.

2.3 BACKBOARD WIRING

This problem was studied by Leon Steinberg in 1961 [32].

The goal of this problem is to minimize the length of connections between units that have to be placed on a rectangular grid, as shown in figure 1. The dimension of the problem is $n = 36$.

1 •	2 •	3 •	4 •	5 •	6 •	7 •	8 •	9 •
10 •	11 •	12 •	13 •	14 •	15 •	16 •	17 •	18 •
19 •	20 •	21 •	22 •	23 •	24 •	25 •	26 •	27 •
28 •	29 •	30 •	31 •	32 •	33 •	34 •	35 •	36 •

Figure 1: Backboard of Steinberg's problem.

As an example, consider two points $\mathsf{P}_i = (x_i, y_i)$ and $\mathsf{P}_j = (x_j, y_j)$. We can refer to their distance $d_{ij} = \mathrm{d}\left(\mathsf{P}_i, \mathsf{P}_j\right)$ in (at least) three different ways:

a) Manhattan distance (or 1-norm): in this case

$$d_{ij}^{a} = |x_i - x_j| + |y_i - y_j|.$$

b) Squared Euclidean distance:

$$d_{ij}^{b} = (x_i - x_j)^2 + (y_i - y_j)^2.$$

c) Euclidean distance multiplied by 1000:

$$d_{ij}^{c} = 1000 \cdot \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

The distance matrix $\boldsymbol{D} = (d_{ij})$ contains the distance between each pair of positions, depending on the distance considered.

The flow matrix $\boldsymbol{F} = (f_{rs})$ (the same for all the three distances) provides the number of connections to make between the units r and s.

The goal is to minimize the total length of wire used to interconnect the components.

2.4 KEYBOARD DESIGN

This problem was firstly studied by Burkard and Offerman in 1977 [4].

The goal is to find out what would be, in theory, the best typewriter keyboards for various languages and for mechanical or electrical machines.

Since the number of letters on ISO basic Latin alphabet is 26, the size of this problems is $n = 26$.

The distance matrix $\boldsymbol{D}$ corresponds to the time between the typing of two keys (the time depends on the fact that the machine is an electrical or a mechanical one).

The flow matrix $\boldsymbol{F}$ contains the frequencies of appearance of two letters in a given language [34].

As four different languages and two typewriters are considered, there are eight problems of this type.

English, French, German and Dutch languages are considered in literature [34]. The proposed methods yield improvements of 7-10% compared with the international standard keyboard.

In 2009 Dell'Amico et al [10] studied the problem of designing a single-finger keyboard for smartphones.

2.5 DARTBOARD DESIGN

The game of darts consists in hitting sectors of a circular target (called *dartboard*) with darts in order to obtain the greatest possible score.

Let us focus on the numbers around the dartboard, shown in figure 2. As we can see, there are 20 numbers (from 1 to 20), each one corresponding to a sector of the dartboard. Why these specific numbers were chosen? If we look at the number 20 (the biggest one) we can see that its closest numbers are 1 and 5. Hence, aiming the sector with 20 has a great risk, since doing a mistake results in loosing many points. This implies a maximization problem: choosing the numbers around a dartboard to maximize the risk.

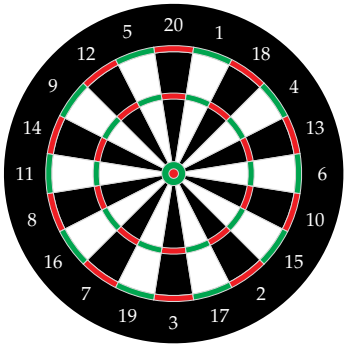


Figure 2: A dartboard.

The current scoring system was devised in 1896 by Brian Gamlin values [11]. In 1991 Eiselt and Laporte [11] described this problem as an instance of QAP.

Let $\pi(k)$ denote the number placed in position k on the dartboard (starting from an arbitrary position) and let $\pi = [\pi(1), \ldots, \pi(20)]$ be any permutation of the numbers $1, \ldots, 20$. In the following lines, $\pi(k)$ must be interpreted as $\pi(k \mod 20)$ whenever $k < 1$ or $k > 20$.

Now, consider a player aiming at $\pi(k)$. Let us suppose that he hits $\pi(k)$ with probability p_0, $\pi(k \pm 1)$ with probability p_1 and, in general, $\pi(k \pm t)$ with probability p_t ($t = 0, \ldots, 10$). Since p_t are probabilities, it must be

$$p_0 + 2\sum_{t=1}^{9} p_t + p_{10} = 1. \tag{16}$$

As suggested by Eiselt and Laporte [11], it seems realistic enough to restrict ourselves to the case where players never hit more than two sectors away from their target. Therefore, we assume that $p_t = 0$ for $t > 2$.

Thus, equation (16) implies

$$p_0 = 1 - 2p_1 - 2p_2. \tag{17}$$

Moreover, we may assume $p_2 = \theta p_1$ with $\theta \in (0, 1)$.

Hence, for every $k \in \{1, \ldots, 20\}$, the expected deviation from the aimed score is

$$\begin{aligned} &p_1\Big(|\pi(k+1) - \pi(k)| + |\pi(k-1) - \pi(k)|\Big) + \theta p_1\Big(|\pi(k+2) - \pi(k)| + |\pi(k-2) - \pi(k)|\Big) \\ &= p_1\Big[\Big(|\pi(k+1) - \pi(k)| + |\pi(k-1) - \pi(k)|\Big) + \theta\Big(|\pi(k+2) - \pi(k)| + |\pi(k-2) - \pi(k)|\Big)\Big] \end{aligned} \tag{18}$$

Notice that in (18) the probability p_1 is constant and, if we sum for k, every term is counted twice.

Therefore, the objective function z to be maximized is

$$z = \sum_{k=1}^{20} |\pi(k+1) - \pi(k)| + \theta \sum_{k=1}^{20} |\pi(k+2) - \pi(k)|. \tag{19}$$

Now, fix a permutation $\pi \in S_{20}$. This problem has an equivalent binary programming form. For every $i, j \in \{1, \ldots, 20\}$, define the binary variables x_{ij} as equal to 1 if (in the permutation π) i is followed immediately by j (i.e. if $\pi(k) = i$ and $\pi(k+1) = j$ for some k), and equal to 0 otherwise. In practice,

$$x_{ij} = \begin{cases} 1 & \text{if exists } k \text{ such that } \pi(k) = i \text{ and } \pi(k+1) = j; \\ 0 & \text{otherwise.} \end{cases} \tag{20}$$

First, we note that $\boldsymbol{X} = (x_{ij})$ is a permutation matrix. Secondly, for $i, j \in \{1, \ldots, n\}$ it follows that

$$\begin{aligned} |i - j|\, x_{ij} \neq 0 &\iff \exists k \text{ such that } i = \pi(k) \text{ and } j = \pi(k+1) \\ &\iff |i - j|\, x_{ij} = |\pi(k+1) - \pi(k)|. \end{aligned} \tag{21}$$

Now, similar to (21), for $i, j, l \in \{1, \ldots, n\}$, we get

$$\begin{aligned} |i-l|\, x_{ij}\, x_{jl} \neq 0 &\iff \exists k \text{ such that } i = \pi(k),\ j = \pi(k+1) \text{ and } l = \pi(k+2) \\ &\iff |i-l|\, x_{ij}\, x_{jl} = |\pi(k+2) - \pi(k)|. \end{aligned} \tag{22}$$

Therefore, summing each terms on (21) and (22), the objective function in (19) can be rewritten:

$$z = \sum_{i=1}^{20}\sum_{j=1}^{20} |i-j|\, x_{ij} + \theta \sum_{i=1}^{20}\sum_{j=1}^{20}\sum_{l=1}^{20} |i-l|\, x_{ij}\, x_{jl}. \tag{23}$$

Finally, the problem can be written in the following form:

$$\begin{aligned} \max \quad & \left[\sum_{i=1}^{20}\sum_{j=1}^{20} |i-j|\, x_{ij} + \sum_{i=1}^{20}\sum_{j=1}^{20}\sum_{l=1}^{20} |i-l| \theta\, x_{ij}\, x_{jl} \right] \\ \text{s.t} \quad & \sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j \in \{1, \ldots, n\} \\ & \sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i \in \{1, \ldots, n\} \\ & x_{ij} \in \{0, 1\} \qquad \forall i, j \in \{1, \ldots, n\} \end{aligned} \tag{24}$$

Which can be expressed as a QAP instance (see [11, p. 116]).

Part II

PRACTICE

3 HEURISTIC ALGORITHMS

COMPUTING optimal solutions is intractable for many optimization problems of industrial and scientific importance. Unlike exact optimization algorithms, i.e., methods theoretically supported, heuristics do not guarantee the optimality of the obtained permutation, but they provide sub-optimal solutions. Moreover, they do not asses how close the obtained permutations are from the optimal ones. In this chapter we will describe two type of methods: *local search* and *greedy* algorithms. Local search algorithms are used by metaheuristic algorithms to improve the objective function, while greedy methods provide an initial permutation used by metaheuristic methods to start with. For each one we will report preliminary results.

NOTATION
Since *one-line* notation is used, often we will consider the permutation π as a vector $\tilde{\boldsymbol{\pi}}$, where $\pi(i) = j$ becomes $\tilde{\pi}_i = j$.

3.1 LOCAL SEARCH ALGORITHMS

As the name suggests, local search algorithms seek a local optimum, i.e., an optimum in a neighborhood.

3.1.1 *Preliminary definitions and results*

We present some theoretical definitions and result.

Definition 3.1 (Distance between permutation)**.** Let $\pi, \sigma \in S_n$ be two permutations. Define the *distance* between π and σ as the numbers of indices at which the corresponding images are different, i.e.,

$$\text{dist}(\pi, \sigma) := |\{i : \pi(i) \neq \sigma(i)\}|.$$

Sometimes this distance is called Hamming distance, from the American mathematician Richard Hamming.

Remark 3.1. This distance is also a topological distance. Moreover, we can observe that there are no permutations with distance 1 between them.

Definition 3.2 (Neighborhood of radius r)**.** Let $\pi \in S_n$ be a permutation and $r \geq 1$ be an integer number. Define the *neighborhood* of center π and radius r as follows:

$$N_r(\pi) := \{\sigma \in S_n \mid \text{dist}(\pi, \sigma) = r\}.$$

Remark 3.2. From remark 3.1 it follows that r cannot be 1.

Definition 3.3 (Composition of permutations)**.** Let $\pi, \sigma \in S_n$. The permutation $\pi \cdot \sigma$ such that

$$(\pi \cdot \sigma)(i) = \pi(\sigma(i)) \qquad \forall i = 1, \ldots, n$$

is called a *composition* of π and σ.

Definition 3.4. Let $\pi \in S_n$ be a permutation. The *support* of π is the set

$$\operatorname{supp}(\pi) := \{i \mid i \in \{1, 2, \dots, n\},\ \pi(i) \neq i\}$$

of the elements "moved" by π.

Definition 3.5 (r-exchange)**.** Let $\pi, \sigma \in S_n$ be two permutations and $r \in \mathbb{N}$ such that $2 \leq r \leq n$. If $\operatorname{supp}(\sigma) = r$, then $\pi \cdot \sigma$ is called a *r-exchange* of π.

For example, let $\pi = [3, 1, 4, 2]$, $\sigma = [2, 1, 3, 4]$ and $\tau = [1, 4, 2, 3]$.
First, note that $|\operatorname{supp}(\sigma)| = |\{1, 2\}| = 2$ and $|\operatorname{supp}(\tau)| = |\{2, 3, 4\}| = 3$.
Then, via composition we obtain that

$$\pi \cdot \sigma = [3, 1, 4, 2] \cdot [2, 1, 3, 4] = [1, 3, 4, 2]$$

is a 2-exchange and

$$\pi \cdot \tau = [3, 1, 4, 2] \cdot [1, 4, 2, 3] = [3, 2, 1, 4]$$

is a 3-exchange.

Proposition 3.1 conveys the idea of the magnitude of a neighborhood N_r is. We did not find a proof of this in the literature, therefore we include it for sake of completeness

Proposition 3.1. *Let $\pi \in S_n$ a permutation. It follows that*

1. *The family of sets $\Big\{\{\pi\},\ N_2(\pi),\ N_3(\pi),\ \dots,\ N_n(\pi)\Big\}$ is a partition of S_n;*
2. $|N_r(\pi)| = \binom{n}{r} r! \sum_{i=2}^{r} \frac{(-1)^i}{i!}$ *for every $\pi \in S_n$.*

Proof. The first statement follows from the definition of the neighborhood N_r.

As regards the second point, let $\pi \in S_n$ be a permutation.

First, note that there are $\binom{n}{r}$ possible way to choose a subset of r elements from a set of n numbers.

Now, without loss of generality, we can consider $\pi = \mathrm{id}$, where id is the identity permutation, i.e., $\mathrm{id}(i) = i$ for every $i = 1, \dots, n$.

The number of possible permutation after a r-exchange with π is equivalent to the numbers of permutation of r elements with no fixed points (called derangements).

Therefore, for every $\pi \in S_n$, $|N_r(\pi)| = \binom{n}{r} \cdot !r$, where $!r$ is the number of derangement of a set of size r.

It is well-known [36, (1)] that the number of derangement of r elements is

$$!r = r! \cdot \sum_{i=0}^{r} \frac{(-1)^i}{i!}.$$

Finally, the second statement follows, since $\sum_{i=0}^{r} \frac{(-1)^i}{i!} = \sum_{i=2}^{r} \frac{(-1)^i}{i!}$. □

Corollary 3.1. *From point 2, it follows that*

- $|N_2(\pi)| = \binom{n}{2} = \frac{n^2 - n}{2}$.
- $|N_3(\pi)| = 2\binom{n}{3} = \frac{n^3 - 3n^2 + 2n}{3}$.

Corollary 3.1 implies that doing a complete visit of N_2 requires $O(n^2)$ operations, while visiting N_3 requires $O(n^3)$ operations.

Definition 3.6. A permutation $\pi^* \in S_n$ is called a *r-optimum* (or *r- optimal*) if $z(\pi^*) \leq z(\sigma)$ for every $\sigma \in N_r(\pi^*)$. An algorithm is called *r-optimum* if it provides a r-optimal permutation.

Table 2: Name of local search algorithms

Name of the algorithm	r	Abbreviation
2-optimum first improvement	2	`2optFirst`
2-optimum best improvement	2	`2optBest`
3-optimum first improvement	3	`3optFirst`
3-optimum best improvement	3	`3optBest`

Basics on r-optimum algorithms

We are going to study two kinds of r-optimum algorithms:

1. **First improvement:** this algorithm explores the r-neighborhood centered in the initial permutation and stops as soon as a reduction of the value of the objective function is obtained. Then, the exploration starts again from the r-neighborhood centered in the best permutation found. In the worst case (i.e., when no improvement is found), a complete evaluation of the neighborhood is performed.

2. **Best improvement:** this algorithm tries every r-exchange and chooses the best one. Hence, the exploration of the r-neighborhood is exhaustive, and all possible moves are examined to select the best neighboring permutation. This form of exploration may result in a longer running time for large neighborhoods.

For $r = 4$ proposition (3.1) tell us that visiting the whole neighbor has a computational cost of $O(n^4)$ operations. Thus, it is not worth to investigate cases for $r \geq 4$. Therefore, this thesis focuses on cases $r = 2$ and $r = 3$.

Every r-optimum algorithm has the same input and output:

INPUT The starting permutation p.

OUTPUT The final permutation m and its objective function value $z_m = z(m)$.

Finally, we will discuss briefly the notation used in the next pages.

Notation

For sake of clarity, in the following pages $\pi_{i_1 i_2}$ will denote the composition of $\pi \cdot \sigma$.

For example, if $\pi = [3, 1, 4, 2]$, then $\pi_{12} = [1, 3, 4, 2]$.

Sometimes we will refer to this 2-exchange as $\{1, 2\} \to \{2, 1\}$.

In table 2 we report the algorithms we considered and the abbreviation we will use to refer to them.

3.1.2 *Preliminary on 2-optimum algorithms*

The first 2-optimum algorithm for QAP was studied by Charles H. Heider in 1973 [20].

In order to reduce the overall computational cost associated to evaluation of the objective function z, it is possible to proceed as follows.

Let $\pi \in S_n$ and fix two indices $i_1 \neq i_2$ of $[n]$. In the following results let $\pi_{i_1 i_2}$ denote the permutation obtained by π, exchanging the indices i_1 and i_2, i.e.,

$$\pi_{i_1 i_2}(i) := \begin{cases} \pi(i_2) & \text{if } i = i_1; \\ \pi(i_1) & \text{if } i = i_2; \\ \pi(i) & \text{otherwise.} \end{cases}$$

Definition 3.7. Let $\pi \in S_n$ and fix two indices $i_1 \neq i_2 \in [n]$. Let $\Delta(\pi; i_1, i_2)$ be defined as

$$\begin{aligned} \Delta(\pi; i_1, i_2) : = & \left(f_{i_1 i_1} - f_{i_2 i_2}\right)\left(d_{\pi(i_2)\pi(i_2)} - d_{\pi(i_1)\pi(i_1)}\right) \\ & + \left(f_{i_1 i_2} - f_{i_2 i_1}\right)\left(d_{\pi(i_2)\pi(i_1)} - d_{\pi(i_1)\pi(i_2)}\right) \\ & + \sum_{j \notin \{i_1, i_2\}} \Big\{ \left(f_{i_1 j} - f_{i_2 j}\right)\left(d_{\pi(i_2)\pi(j)} - d_{\pi(i_1)\pi(j)}\right) \\ & \qquad + \left(f_{j i_1} - f_{j i_2}\right)\left(d_{\pi(j)\pi(i_2)} - d_{\pi(j)\pi(i_1)}\right) \Big\}. \end{aligned} \tag{25}$$

Remark 3.3. If the matrices $\boldsymbol{F}$ and $\boldsymbol{D}$ are symmetric, then the formula (25) becomes

$$\Delta(\pi; i_1, i_2) = 2 \sum_{j \notin \{i_1, i_2\}} \left(f_{j i_1} - f_{j i_2}\right)\left(d_{\pi(j)\pi(i_2)} - d_{\pi(j)\pi(i_1)}\right). \tag{26}$$

Remark 3.4. The evaluation of $\Delta(\pi; i_1, i_2)$ requires $2(n-2)+2 = 2n-2$ products and $6n-7$ sums. In the symmetric case, the evaluation requires $n-1$ products and $3n-6$ sums. In every case, Δ can be evaluated in $O(n)$ operations.

Theorem 3.1. *Let $\pi \in S_n$ be a permutation and $i_1 \neq i_2$ be two indices. Then,*

$$z(\pi_{i_1 i_2}) = z(\pi) + \Delta(\pi; i_1, i_2). \tag{27}$$

We did not find a proof of this in the literature, therefore we include it in Appendix A for the sake of completeness.

Observe that theorem 3.1 allows us to evaluate the difference between a permutation π and its 2-exchange $\pi_{i_1 i_2}$ with a cost of $O(n)$ operations.

Taking account that the straightforward evaluation of a quadratic objective function has a cost of $O(n^2)$, we got the conclusion that the approach described below, exploiting the structure of the problem, allows us to save some computational time.

3.1.3 2-*optimum: First improvement*

Description of the algorithm

A detailed description of this algorithm can be found in pseudo code 1.

We can see that there are two `DO` cycles (lines 4-5), in order to try every 2-exchange of the current permutation m.

In every step, the algorithm evaluates the term Δ until it finds a 2-exchange which improves the current permutation (lines 6 - 11). This is achieved by the `IF` cycle in line 7: if $\Delta(m, i_1, i_2) < 0$, then the 2 exchange $m_{i_1 i_2}$ has a lower objective function value than m.

Then, the algorithm performs the 2-exchange and starts again from $m_{i_1 i_2}$ (lines 9-10).

The algorithm stops whenever there are no 2-exchange that improve the current permutation, i.e., when $\Delta(m, i_1, i_2) \geq 0$ for every $i_1, i_2 \in \{1, \ldots, n\}$.

Algorithm 1: 2-optimum, first improvement

```
   Input: n,F,D, p
   Output: m, z_m
   /* initialization                                   */
1  Evaluate z_p = z(p);
2  m ← p;
3  z_m ← z_p;
   /* main loop                                        */
4  for i_1 = 1,...,n-1 do
5      for i_2 = i_1+1,...,n do
6          Evaluate Δ(m;i_1,i_2);
7          if Δ(m;i_1,i_2) < 0 then
8              z_m ← z_m + Δ(m;i_1,i_2);
9              Do the 2-exchange m ← m_{i_1 i_2};
10             Go to step 4;
11         end
12     end
13 end
14 Stop: m is 2-optimal with value z_m.
```

Hence, the final permutation is 2-optimum because there are no more 2-exchange available and every 2-exchange increase the objective function value.

Note that this is a finite termination algorithm because there are a finite number of permutations. Thus, the algorithm performs a finite numbers of iterations.

Complexity of the algorithm

In the best case, an incrementing 2-exchange is always found in the first attempt, so in lines 5 - 12 the algorithm requires only one Δ-evaluation.

The cost of evaluating Δ is $O(n)$ operations.

Hence, the final cost of lines 4 - 13 in the best case is $O(n)$ operations.

In the worst case, the incrementing 2-exchange is found at the end of the loops in steps 4-5. Since the algorithm tests every possible 2-exchange, the number of evaluations done in lines 4 - 13 is

$$\begin{aligned}\sum_{i_1=1}^{n-1}\sum_{i_2=i_1+1}^{n} 1 &= \sum_{i_1=1}^{n-1}(n-i_1)\\ &= \sum_{i_1=1}^{n-1} n - \sum_{i_1=1}^{n-1} i_1\\ &= n\cdot(n-1) - \frac{n}{2}\cdot(n-1)\\ &= \frac{n(n-1)}{2} = \binom{n}{2},\end{aligned} \tag{28}$$

as a further evidence of the proposition 3.1.

Since the evaluation cost of Δ is $O(n)$ operations, the total cost of lines 4-13 in worst case is $O(n^3)$.

These costs are repeated for every 2-exchange made. There are no way to evaluate *a priori* how many 2-exchanges the algorithm is gonna do, but we are going to try some tests in Section 3.1.8.

Table 3: Example of `2optfirst` algorithm

Indices		Permutation				Cost	Remarks
i_1	i_2	1	2	3	4		
		1	2	3	4	908	Initial permutation
1	2					926	No improvement
1	3					1008	No improvement
1	4					1052	No improvement
2	3					1136	No improvement
2	4					850	Exchange locations 2 and 4
		1	4	3	2	850	Start again
1	2					930	No improvement
1	3					790	Exchange locations 1 and 3
		3	4	1	2	790	Start again
1	2					834	No improvement
1	3					850	No improvement
1	4					1066	No improvement
2	3					1018	No improvement
2	4					1008	No improvement
3	4					824	No improvement
		3	4	1	2	790	Best permutation found

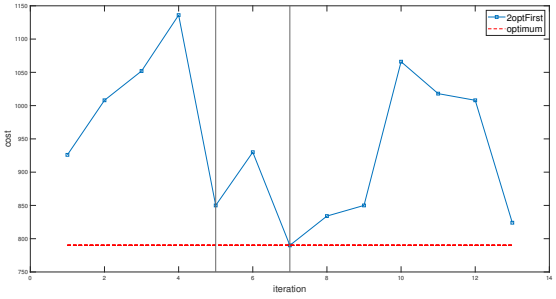


Figure 3: `2optFirst` algorithm, Objective function values versus the iterations.

An example

A basic example is described in table 3. Instance `Neos4` was used. See Section 5.2 for more details.

The quality of a permutation is defined by the Percent Deviation (PD) from the best known solution, calculated according to

$$\mathrm{PD}(z, z_{\mathrm{BKS}}) := \frac{z - z_{\mathrm{BKS}}}{z_{\mathrm{BKS}}} \cdot 100, \tag{29}$$

where z is the obtained permutation and z_{BKS} is the best known solution (or the optimum) of the corresponding problem.

In this case, the 2-optimal permutation achieved is an optimum. This is, in general, not true. In figure 3 we plotted the objective function values versus the iterations. As we can see, as soon as the algorithm finds a permutation reducing the cost value, it starts again from this permutation, exploring its r-neighborhood.

3.1.4 2-*optimum: Best improvement*

Description of the algorithm

A detailed description of this algorithm can be found in pseudo code 2.

Algorithm 2: 2-optimum: best improvement

```
   Input: n,F,D,p
   Output: m, z_m
   /* initialization                                          */
1  Evaluate z_p = z(p);
2  m ← p;
3  z_m ← z_p;
4  d_min ← 0;
   /* main loop                                               */
5  for i_1 = 1,...,n−1 do
6  |  for i_2 = i_1+1,...,n do
7  |  |  Evaluate Δ(m;i_1,i_2);
8  |  |  if Δ(m;i_1,i_2) < d_min then
9  |  |  |  j_1 ← i_1;
10 |  |  |  j_2 ← i_2;
11 |  |  |  d_min ← Δ(m;j_1,j_2);
12 |  |  end
13 |  end
14 end
15 if d_min < 0 then  /* If m is the new best permutation found */
16 |  z_m ← z_m + d_min;
17 |  Do the 2-exchange m ← m_{i_1 i_2};
18 |  Go to step 4;
19 end
20 if d_min = 0 then stop: m is 2-optimal with value z_m;
```

The *best improvement* algorithm starts from a permutation p with objective function value $z(p) = z_p$; it sets $m = p$, $z_m = z_p$ and initializes $d_{\min}$ at 0 (lines 1-4).

There are two `DO` cycles, in order to try every possible 2-exchange of p (lines 5 - 6).

For every i_1 and i_2, algorithm evaluates the term $\Delta(m; i_1, i_2)$ (line 7) and it computes the following term (lines 8-12):

$$d_{\min} := \min\Big\{\Delta(m; i_1, i_2) \,\Big|\, i_1 \in \{1, \ldots, n-1\}, i_2 \in \{i_1 + 1, \ldots, n\}\Big\}.$$

Now, if $d_{\min} < 0$, then the 2-exchange is applied and m and z_m are updated (lines 15 - 17).

The algorithm starts again from the new m (line 18).

This procedure is repeated until $d_{\min} = 0$ (note that $d_{\min}$ cannot be positive, since it is initialized as 0).

Observe that since there are a finite numbers of permutations, the algorithm ends in a finite number of iterations.

In case $d_{\min} = 0$, every possible 2-exchange does not improve the objective function. Therefore, m is 2-optimal with objective function z_m (line 20).

Complexity of the algorithm

Note that `2optBest` algorithm corresponds to the worst case of `2optFirst`. Hence its computational cost is $O(n^3)$, as shown in equation (28).

Table 4: Example of best improvement algorithm

Indices		Permutation				Cost	Remarks
i_1	i_2	1	2	3	4		
		1	2	3	4	908	Initial permutation
1	2					926	No improvement
1	3					1008	No improvement
1	4					1052	No improvement
2	3					1136	No improvement
2	4					850	Improvement: store indices
3	4					864	Improvement, but worst than 850
		1	2	3	4	864	Exchange locations 2 and 4
		1	4	3	2	850	Start again
1	2					930	No improvement
1	3					790	Improvement: store indices
1	4					990	No improvement
2	3					982	No improvement
2	4					908	No improvement
3	4					960	No improvement
		1	4	3	2	960	Exchange locations 1 and 3
		3	4	1	2	790	Start again
1	2					834	No improvement
1	3					850	No improvement
1	4					1066	No improvement
2	3					1018	No improvement
2	4					1008	No improvement
3	4					824	No improvement
		3	4	1	2	790	Best permutation found

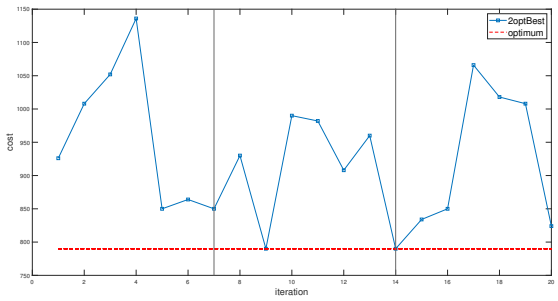


Figure 4: `2optBest` algorithm, Objective function values versus iterations.

An example

A basic example is described in table 4. Instance `Neos4` was used. See section 5.2 for more details.

Note the main differences between Tables 4 and 3: when `2optBest` find an improvement of the objective function, it stores the indices of the improving 2-exchange.

As we can see, the algorithm achieves the optimal solution. Figure 4 shows the variation of objective function value. We can clearly see that every cycle is repeated three times.

3.1.5 *Preliminary on 3-optimum algorithms*

The 3-optimum algorithm was first proposed by G. A. Croes [9] in 1958 for Travelling Salesman Problem.

These 3-optimum algorithm are similar to the 2-optimum algorithms except that they consider swapping three facilities at a time.

With three distinct indices i_1,i_2,i_3, one can obtain two kind of 3-exchanges:

1. $\{i_1, i_2, i_3\} \to \{i_2, i_3, i_1\}$.
2. $\{i_1, i_2, i_3\} \to \{i_3, i_1, i_2\}$.

If $\pi \in S_n$ is a permutation and i_1,i_2,$i_3 \in [n]$ are three distinct indices, define $\pi^1_{i_1 i_2 i_3} \in N_3(\pi)$ as the permutation such that

$$\pi^1_{i_1 i_2 i_3}(j) := \begin{cases} \pi(i_2) & \text{if } j = i_1; \\ \pi(i_3) & \text{if } j = i_2; \\ \pi(i_1) & \text{if } j = i_3; \\ \pi(j) & \text{otherwise.} \end{cases} \tag{30}$$

and $\pi^2_{i_1 i_2 i_3} \in N_3(\pi)$ as

$$\pi^2_{i_1 i_2 i_3}(j) := \begin{cases} \pi(i_3) & \text{if } j = i_1; \\ \pi(i_1) & \text{if } j = i_2; \\ \pi(i_2) & \text{if } j = i_3; \\ \pi(j) & \text{otherwise.} \end{cases} \tag{31}$$

For example, if $\pi = [3, 1, 4, 2]$, then

$$\pi^2_{123} = [1, 3, 4.2] \qquad \text{and } \pi^1_{123} = [4, 3, 1, 2] \qquad .$$

Definition 3.8. Let $\pi \in S_n$ be a permutation and i_1,i_2,$i_3 \in [n]$ three distinct indices. Define $\Delta^1(\pi; i_1, i_2, i_3)$ and $\Delta^2(\pi; i_1, i_2, i_3)$ as follows.

$$\begin{aligned} \Delta^1(\pi; i_1, i_2, i_3) := {} & f_{i_1 i_1}\left(d_{\pi(i_2)\pi(i_2)} - d_{\pi(i_1)\pi(i_1)}\right) + f_{i_1 i_2}\left(d_{\pi(i_2)\pi(i_3)} - d_{\pi(i_1)\pi(i_2)}\right) \\ & + f_{i_1 i_3}\left(d_{\pi(i_2)\pi(i_1)} - d_{\pi(i_1)\pi(i_3)}\right) + f_{i_2 i_1}\left(d_{\pi(i_3)\pi(i_2)} - d_{\pi(i_2)\pi(i_1)}\right) \\ & + f_{i_2 i_2}\left(d_{\pi(i_3)\pi(i_3)} - d_{\pi(i_2)\pi(i_2)}\right) + f_{i_2 i_3}\left(d_{\pi(i_3)\pi(i_1)} - d_{\pi(i_2)\pi(i_3)}\right) \\ & + f_{i_3 i_1}\left(d_{\pi(i_1)\pi(i_2)} - d_{\pi(i_3)\pi(i_1)}\right) + f_{i_3 i_2}\left(d_{\pi(i_1)\pi(i_3)} - d_{\pi(i_3)\pi(i_2)}\right) \\ & + f_{i_3 i_3}\left(d_{\pi(i_1)\pi(i_1)} - d_{\pi(i_3)\pi(i_3)}\right) \\ & + \sum_{j \notin \{i_1, i_2, i_3\}} \Big[f_{i_1 j}\left(d_{\pi(i_2)\pi(j)} - d_{\pi(i_1)\pi(j)}\right) + f_{i_2 j}\left(d_{\pi(i_3)\pi(j)} - d_{\pi(i_2)\pi(j)}\right) \\ & \qquad + f_{i_3 j}\left(d_{\pi(i_1)\pi(j)} - d_{\pi(i_3)\pi(j)}\right) + f_{j i_1}\left(d_{\pi(j)\pi(i_2)} - d_{\pi(j)\pi(i_1)}\right) \\ & \qquad + f_{j i_2}\left(d_{\pi(j)\pi(i_3)} - d_{\pi(j)\pi(i_2)}\right) + f_{j i_3}\left(d_{\pi(j)\pi(i_1)} - d_{\pi(j)\pi(i_3)}\right) \Big]. \end{aligned} \tag{32}$$

and

$$\begin{aligned} \Delta^2(\pi; i_1, i_2, i_3) := {} & f_{i_1 i_1}\left(d_{\pi(i_3)\pi(i_3)} - d_{\pi(i_1)\pi(i_1)}\right) + f_{i_1 i_2}\left(d_{\pi(i_3)\pi(i_1)} - d_{\pi(i_1)\pi(i_2)}\right) \\ & + f_{i_1 i_3}\left(d_{\pi(i_3)\pi(i_2)} - d_{\pi(i_1)\pi(i_3)}\right) + f_{i_2 i_1}\left(d_{\pi(i_1)\pi(i_3)} - d_{\pi(i_2)\pi(i_1)}\right) \\ & + f_{i_2 i_2}\left(d_{\pi(i_1)\pi(i_1)} - d_{\pi(i_2)\pi(i_2)}\right) + f_{i_2 i_3}\left(d_{\pi(i_1)\pi(i_2)} - d_{\pi(i_2)\pi(i_3)}\right) \\ & + f_{i_3 i_1}\left(d_{\pi(i_2)\pi(i_3)} - d_{\pi(i_3)\pi(i_1)}\right) + f_{i_3 i_2}\left(d_{\pi(i_2)\pi(i_1)} - d_{\pi(i_3)\pi(i_2)}\right) \\ & + f_{i_3 i_3}\left(d_{\pi(i_2)\pi(i_2)} - d_{\pi(i_3)\pi(i_3)}\right) \\ & + \sum_{j \notin \{i_1, i_2, i_3\}} \Big[f_{i_1 j}\left(d_{\pi(i_3)\pi(j)} - d_{\pi(i_1)\pi(j)}\right) + f_{i_2 j}\left(d_{\pi(i_1)\pi(j)} - d_{\pi(i_2)\pi(j)}\right) \\ & \qquad + f_{i_3 j}\left(d_{\pi(i_2)\pi(j)} - d_{\pi(i_3)\pi(j)}\right) + f_{j i_1}\left(d_{\pi(j)\pi(i_3)} - d_{\pi(j)\pi(i_1)}\right) \\ & \qquad + f_{j i_2}\left(d_{\pi(j)\pi(i_1)} - d_{\pi(j)\pi(i_2)}\right) + f_{j i_3}\left(d_{\pi(j)\pi(i_2)} - d_{\pi(j)\pi(i_3)}\right) \Big]. \end{aligned} \tag{33}$$

Remark 3.5. In symmetric QAP($\boldsymbol{F}$,$\boldsymbol{D}$), equations (32) and (33) reduce to

$$\begin{aligned}\Delta^1(\pi; i_1, i_2, i_3) := 2\Big\{ & f_{i_1 i_2}\left(d_{\pi(i_2)\pi(i_3)} - d_{\pi(i_1)\pi(i_2)}\right) + f_{i_1 i_3}\left(d_{\pi(i_2)\pi(i_1)} - d_{\pi(i_1)\pi(i_3)}\right) \\ & + f_{i_2 i_3}\left(d_{\pi(i_1)\pi(i_3)} - d_{\pi(i_2)\pi(i_3)}\right) + f_{i_3 i_3}\left(d_{\pi(i_1)\pi(i_1)} - d_{\pi(i_3)\pi(i_3)}\right) \\ & + \sum_{j \notin \{i_1, i_2, i_3\}} \Big[f_{j i_1}\left(d_{\pi(j)\pi(i_2)} - d_{\pi(j)\pi(i_1)}\right) + f_{j i_2}\left(d_{\pi(j)\pi(i_3)} - d_{\pi(j)\pi(i_2)}\right) \\ & + f_{j i_3}\left(d_{\pi(j)\pi(i_1)} - d_{\pi(j)\pi(i_3)}\right) \Big] \Big\}\end{aligned}$$

and

$$\begin{aligned}\Delta^2(\pi; i_1, i_2, i_3) := 2\Big\{ & f_{i_1 i_2}\left(d_{\pi(i_1)\pi(i_3)} - d_{\pi(i_1)\pi(i_2)}\right) + f_{i_1 i_3}\left(d_{\pi(i_2)\pi(i_3)} - d_{\pi(i_1)\pi(i_3)}\right) \\ & + f_{i_2 i_3}\left(d_{\pi(i_1)\pi(i_2)} - d_{\pi(i_2)\pi(i_3)}\right) + f_{i_3 i_3}\left(d_{\pi(i_1)\pi(i_1)} - d_{\pi(i_3)\pi(i_3)}\right) \\ & + \sum_{j \notin \{i_1, i_2, i_3\}} \Big[f_{j i_1}\left(d_{\pi(j)\pi(i_3)} - d_{\pi(j)\pi(i_1)}\right) + f_{j i_2}\left(d_{\pi(j)\pi(i_1)} - d_{\pi(j)\pi(i_2)}\right) \\ & + f_{j i_3}\left(d_{\pi(j)\pi(i_2)} - d_{\pi(j)\pi(i_3)}\right) \Big] . \Big\}\end{aligned}$$

Remark 3.6. The evaluation of Δ^1 and Δ^2 requires $6n - 9$ products and $12n - 8$ sums. Respectively, in symmetric case, the evaluation of them requires $3n - 5$ products and $6n - 12$ sums.

In every case, they can be evaluated in $O(n)$ operations.

Theorem 3.2. *Let $\pi \in S_n$ be a permutation and $i_1, i_2, i_3 \in [n]$ three distinct indices. Then one has*

- $z\left(\pi^1_{i_1 i_2 i_3}\right) = z(\pi) + \Delta^1(\pi; i_1, i_2, i_3)$
- $z\left(\pi^2_{i_1 i_2 i_3}\right) = z(\pi) + \Delta^2(\pi; i_1, i_2, i_3)$

We did not find a proof of this in the literature, therefore we include it here for the sake of completeness. The proof can be found on Appendix B.

3.1.6 3-*optimum: first improvement*

Description of the algorithm

A detailed description of this algorithm can be found in pseudo code 3.

Algorithm 3: 3-optimum: first improvement

Input: n,$\boldsymbol{F}$,$\boldsymbol{D}$,p
Output: m, z_m
/* initialisation */
Read p;
Evaluate $z_p \leftarrow z(p)$;
$z_m \leftarrow z_p$;
/* main loops */
for $i_1 = 1, \ldots, n-2$ **do**
  for $i_2 = i_1 + 1, \ldots, n-1$ **do**
    for $i_3 = i_2 + 1, \ldots, n$ **do**
      Evaluate $\Delta^1(m; i_1, i_2, i_3)$;
      if $\Delta^1(m; i_1, i_2, i_3) < 0$ **then**
        Do the 3-exchange $m \leftarrow m_{i_1 i_2 i_3}$;
        $z_m \leftarrow z_m + \Delta^1(m; i_1, i_2, i_3)$;
        Goto step 4;
      end
      Evaluate $\Delta^2(m; i_1, i_2, i_3)$;
      if $\Delta^2(m; i_1, i_2, i_3) < 0$ **then**
        Do the 3-exchange $m \leftarrow m_{i_1 i_2 i_3}$;
        $z_m \leftarrow z_m + \Delta^2(m; i_1, i_2, i_3)$;
        Goto step 4;
      end
    end
  end
end
Stop: m is 3-optimal with objective function value z_m.

This algorithm is fairly similar to `2optFist`. In fact, the algorithm updates that permutation as soon as it founds an improvement obtained by a 3-exchange.

The main difference is that there are two possible 3-exchanges that can be done. Thus, the algorithm evaluates both Δ^1 and Δ^2 terms.

Complexity of the algorithm

In the best case, the algorithm immediately finds an improving 3-exchange with $\Delta^1 < 0$. Hence in this case, for every 3-exchange, the computational cost of `3optFirst` is $O(n)$ operations.

In the worst case, the algorithm will find the improving 3-exchange only at the end of each iterations. Equation (34) shows us that `3optFirst`, in the

worst case, requires $\binom{n}{3} = O(n^3)$ loops, therefore the total cost in the worst case is $O(n^4)$ operations.

$$\begin{aligned}\sum_{i_1=1}^{n-2}\left(\sum_{i_2=i_1+1}^{n-1}\left(\sum_{i_3=i_2+1}^{n}1\right)\right) &= \sum_{i_1=1}^{n-2}\left(\sum_{i_2=i_1+1}^{n-1}(n-i_2)\right)\\ &= \sum_{i_1=1}^{n-2}\frac{(n-i_1)(n-i_1-1)}{2}\\ &= \sum_{i_1=1}^{n-2}\binom{n-i_1}{2}\\ &= \sum_{j=2}^{n-1}\binom{j}{2} = \sum_{j=0}^{n-1}\binom{j}{2} = \binom{n}{3},\end{aligned} \tag{34}$$

where we used the known identity

$$\sum_{j=0}^{n}\binom{j}{m} = \binom{n+1}{m+1}$$

(see (5.10) in Concrete Mathematicsby D. E. Knuth, O. Patashnik, and R. Graham).

An example

A basic example is described in table 5, where `3optFirst` starts with initial permutation $[1,2,3,4,5]$. Instance `Neos5` was used; see section 5.2 for more details.

The optimal solution of `Neos5` instance is $\pi = [5,4,1,2,3]$ with $z(\pi) = 628$. As we can see, the algorithm did not find the optimum, but 656, with 4% Percent Deviation (PD).

3.1.7 3-*optimum: best improvement*

A detailed description of this algorithm can be found in pseudo code 4.

The algorithm is very similar to `2optBest`. In fact, it evaluates all two types of all possible 3-exchanges from the current permutation m.

In $d_{\min}$ is stored the best value found by doing 3-exchanges, i.e., the greatest difference from z_m.

The indices j_1,j_2 and j_3 that realized this exchange, are stored too.

Then the algorithm updates the permutation with the best one found and starts again (line 27 - 31).

It stops when a permutation that is not improving by any 3-exchange is obtained (line 33).

Complexity of algorithm

The computational cost of `3optBest` is the same as the worst case of `3optFirst`.

An example

A basic example is described in table 5. Instance `Neos5` was used; see section 5.2 for more details.

Like `3optFirst` algorithm, `3optBest` did find 656 with 4% Percent Deviation (PD) from the optimum.

Algorithm 4: 3-optimum: best improvement

Input: n,$\boldsymbol{F}$,$\boldsymbol{D}$,p
Output: m, z_m
/* initialisation */
Evaluate $z_p = z(p)$;
$m \leftarrow p$;
$z_m \leftarrow z_p$;
$d_{\min} \leftarrow 0$;
/* main loops */
for $i_1 = 1, \ldots, n-2$ **do**
  for $i_2 = i_1 + 1, \ldots, n-1$ **do**
    for $i_3 = i_2 + 1, \ldots, n$ **do**
      Evaluate $\Delta^1(m; i_1, i_2, i_3)$;
      if $\Delta^1(m; i_1, i_2, i_3) < d_{\min}$ **then**
        $j_1 \leftarrow i_1$;
        $j_2 \leftarrow i_2$;
        $j_3 \leftarrow i_3$;
        $l \leftarrow 1$;
        $d_{\min} \leftarrow \Delta^1(m; j_1, j_2, j_3)$;
      end
      Evaluate $\Delta^2(m; i_1, i_2, i_3)$;
      if $\Delta^2(m; i_1, i_2, i_3) < d_{\min}$ **then**
        $j_1 \leftarrow i_1$;
        $j_2 \leftarrow i_2$;
        $j_3 \leftarrow i_3$;
        $l \leftarrow 2$;
        $d_{\min} \leftarrow \Delta^2(m; j_1, j_2, j_3)$;
      end
    end
  end
end
if $d_{\min} < 0$ **then**
  if $l = 1$ **then** Do the 3-exchange $m \leftarrow m^1_{j_1 j_2 j_3}$;
  else if $l = 2$ **then** Do the 3-exchange $m \leftarrow m^2_{j_1 j_2 j_3}$;
  $z_m \leftarrow z_m + d_{\min}$;
  Go to step 4;
end
if $d_{\min} = 0$ **then** stop: m is 3-optimal with value z_m;

Table 5: Example of `3optFirst`.

Indices			Permutation					Cost	Remarks
i_1	i_2	i_4	1	2	3	4	5		
			1	2	3	4	5	900	Initial permutation
1	2	3						836	Exchange locations 1, 2 and 3 with π^1
			2	3	1	4	5	836	Start again
1	2	3						780	Exchange locations 1, 2 and 3 with π^1
			3	1	2	4	5	780	Start again
1	2	3						780	No improvement
1	2	4						780	No improvement
1	2	5						780	No improvement
1	3	4						744	Exchange locations 1, 3 and 4 with π^1
			2	1	4	3	5	744	Start again
1	2	3						660	Exchange locations 1, 2 and 3 with π^2
			4	2	1	3	5	660	Start again
1	2	3						660	No improvement
1	2	4						656	Exchange locations 1, 2 and 4 with π^2
			3	4	1	2	5	656	Start again
1	2	3						656	No improvement
1	2	4						656	No improvement
1	2	5						656	No improvement
1	3	4						656	No improvement
1	3	5						656	No improvement
1	4	5						656	No improvement
2	3	4						656	No improvement
2	3	5						656	No improvement
2	4	5						656	No improvement
3	4	5						656	No improvement
			3	4	1	2	5	656	Best permutation found

Table 6: Example of `3optBest` algorithm starting from initial permutation $[1, 2, 3, 4, 5]$.

Indices			Permutation					Cost	Remarks
i_1	i_2	i_4	1	2	3	4	5		
			1	2	3	4	5	900	Initial permutation
1	2	3						836	$\Delta^1 < 0$: store indices
1	2	3						780	$\Delta^2 < 0$: store indices
1	2	4						968	No improvement
1	2	5						876	No improvement
1	3	4						660	$\Delta^2 < 0$: store indices
1	3	5						844	No improvement
1	4	5						1056	No improvement
2	3	4						880	No improvement
2	3	5						864	No improvement
2	4	5						872	No improvement
3	4	5						920	No improvement
									Exchange locations 1, 3 and 4 with π^2
			4	2	1	3	5	660	Start again
1	2	3						880	No improvement
1	2	4						656	$\Delta^2 < 0$: store indices
1	2	5						792	No improvement
1	3	4						836	No improvement
1	3	5						732	No improvement
1	4	5						844	No improvement
2	3	4						900	No improvement
2	3	5						816	No improvement
2	4	5						852	No improvement
3	4	5						864	No improvement
									Exchange locations 1, 2 and 4 with π^2
			3	4	1	2	5	656	Start again
1	2	3						792	No improvement
1	2	4						836	No improvement
1	2	5						788	No improvement
1	3	4						848	No improvement
1	3	5						756	No improvement
1	4	5						792	No improvement
2	3	4						836	No improvement
2	3	5						724	No improvement
2	4	5						732	No improvement
3	4	5						900	No improvement
			3	4	1	2	5	656	Best permutation found

Table 7: Comparison of local search algorithms on instance `tai12a`. The column headers 2F, 2B, 3F, 3B stands for `2optFirst`, `2optBest`, `3optFirst` and `3optBest` respectively. Finally, IP stands for the initial permutation. We also report the mean, the minimum and the maximum value obtained for PD, n_e, n_r.

IP	IPD	PD				n_e				n_r			
		2F	2B	3F	3B	2F	2B	3F	3B	2F	2B	3F	3B
π_1	51.4	11.8	15.2	10.4	7.3	237	462	1292	2640	11	6	20	5
π_2	45.0	11.0	7.7	10.6	8.1	241	462	723	2200	15	6	12	4
π_3	36.3	10.1	12.3	12.2	12.2	182	462	620	2200	7	6	8	4
π_4	39.5	10.1	9.4	12.5	4.7	373	396	841	2640	17	5	20	5
π_5	50.6	10.0	6.5	11.1	11.1	250	594	1135	2640	18	8	17	5
π_6	34.3	5.7	6.4	9.1	8.5	534	858	2408	2200	14	12	18	4
π_7	37.5	15.9	15.9	9.7	15.5	107	198	1189	1320	8	2	12	2
π_8	33.6	10.3	7.3	8.6	9.1	193	726	2274	3080	13	10	19	6
π_9	39.6	14.1	8.1	5.8	6.9	186	528	3104	2640	10	7	18	5
π_{10}	53.8	11.8	4.6	4.4	8.9	367	594	1917	2200	19	8	24	4
π_{11}	32.0	10.5	11.0	8.2	7.2	242	396	1127	1760	8	5	10	3
π_{12}	37.6	12.7	12.0	9.2	3.3	202	396	1213	3520	7	5	10	7
mean	40.9	11.2	9.7	9.3	8.6	260	506	1487	2420	12	7	16	5
min	32.0	5.7	3.3	4.4	3.3	107	198	620	1320	7	2	8	2
max	53.8	15.9	15.9	12.5	15.5	534	858	3104	3520	19	12	24	7

3.1.8 *Implementation and comparison*

In order to compare local search algorithms, we used instances from QAPLIB [6] library's web page. More details can be found in section 5.1.

We implemented local search algorithms in Fortran and make it available on GitHub [26].

In table 7 we tested 2-optimum algorithms on instance `Had12`. The algorithms take in input the permutation π_i defined as

$$\pi_i = [i, i+1, \dots, n, 1, 2, \dots, i-1],$$

for $i \in \{1, 2, \dots, n\}$.

In the following tables we report:

- n_e: the numbers of Δ evaluations.
- n_r: the numbers of r-exchanges realized; this number represents how many r-exchanges the algorithm performed.
- PD: the percent deviation from optimal point, as defined in (29).
- IPD: the initial PD, the PD of the initial permutation.
- mean, min and max : mean, maximum and minimum over the 12 runs.

Note that final results strongly depends on the initial permutation.

For example, the initial permutation π_7 is bad (15.5% PD). The `2optfirst` algorithm provides a permutation with 3.5% PD, while, the starting permutation π_3 is worse (18.5% PD), but the algorithm's result has only a 0.2% PD.

In general, looking at the means and the minimums, first-improvement algorithms outputs are better than best-improvement ones. This could support the strategy "Do less, get better" used sometimes in real life.

In table 8 the four algorithms are compared on several instances. We divided the instances in two categories: real world and random. More details can be found in section 5.1.

The goal here is not analyzing the objective function value, remember that these are heuristic algorithm, that are generally used in more complex strategies.

Our goal is to compare local search algorithms in order to assess if it is worth doing more evaluations (hence, using 3-optimum or best-improvement algorithms) or not.

The first thing we can observe, is that there is not a very sensible difference. Actually, 3-optimum algorithms frequently reached slightly better PD, but often the difference is under the 1%.

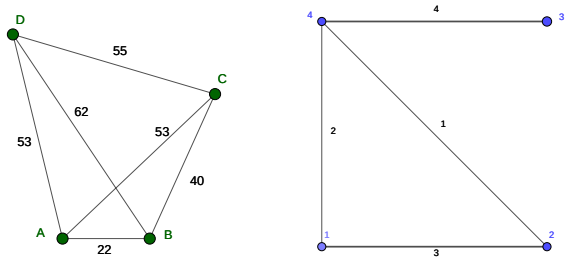
The main difference consists in n_e: the number of the evaluations of Δ. In general 3-optimum algorithms perform much more evaluations than its corresponding 2-optimum, growing with at least one order of magnitude. Sometimes, the grow was two order of magnitude (2F and 3F in `kra32`, 2B and 3B in `nug28`). Remember that each evaluation of Δ has a cost of $O(n)$ operations. Thus, for `kra32`, the `3optBest` algorithm does 5.73×10^6 operations.

On the other hand, the number of r-exchange performed by first-improvement algorithms is higher than that those required by best-improvement method.

3-optimum algorithms on average provide better permutations at expenses of a higher computational cost.

Table 8: Comparison of local search algorithms for several instances. All results are averaged over n runs.

Instance	IPD	PD				n_e				n_r			
		2F	2B	3F	3B	2F	2B	3F	3B	2F	2B	3F	3B
Real-world instances													
bur26a	9.1	0.4	0.3	0.3	0.2	5604	7238	50 005	85 000	97	21	104	15
bur26b	9.8	0.5	0.4	0.3	0.3	6397	6825	39 572	81 800	108	20	123	15
bur26c	9.0	0.4	0.4	0.3	0.2	5815	7550	47 262	80 400	97	22	103	15
bur26d	9.9	0.4	0.5	0.3	0.2	5639	6762	43 511	81 200	105	20	117	15
bur26e	10.1	0.4	0.5	0.2	0.2	6169	7200	53 125	81 800	102	21	109	15
bur26f	11.2	0.5	0.6	0.3	0.3	6963	6837	51 339	80 000	117	20	137	14
bur26g	9.6	0.5	0.3	0.3	0.3	5426	7800	51 251	85 600	98	23	110	15
bur26h	10.7	0.6	0.4	0.4	0.4	7261	7150	45 866	77 600	118	21	125	14
kra30a	48.3	6.7	7.1	5.7	5.9	7168	9381	93 188	125 048	62	21	73	14
kra30b	46.8	5.2	5.5	3.7	3.7	8335	8758	92 554	129 920	65	19	73	15
kra32	51.0	6.9	6.0	4.5	3.9	7728	10 788	119 572	179 180	69	21	85	17
Random instances													
nug12	33.0	6.4	5.6	5.6	5.2	255	418	1170	2090	9	5	10	4
nug14	32.0	6.1	5.2	3.9	4.1	501	754	3098	4940	16	7	19	6
nug15	33.1	5.4	3.1	3.6	4.9	700	1015	3931	6006	19	9	21	6
nug16a	34.6	5.6	4.5	4.2	3.9	786	1290	4305	9030	22	10	25	7
nug16b	36.4	6.2	5.2	4.5	4.1	869	1193	5579	9170	22	9	21	7
nug17	32.5	3.5	4.0	3.7	2.9	965	1552	6111	13 200	26	10	29	9
nug18	32.3	3.8	4.5	3.5	4.0	1333	1658	7730	13 691	29	10	30	7
nug20	33.2	3.9	4.5	3.8	3.4	1962	2669	11 891	24 624	36	13	37	10
nug21	39.8	4.9	4.3	4.0	3.8	2509	3340	20 198	30 907	42	15	42	11
nug22	37.6	4.4	4.5	2.7	3.3	3634	3937	32 819	37 520	50	16	52	11
nug24	29.9	3.9	5.0	3.7	3.7	4285	4772	34 072	52 961	47	16	46	12
nug25	31.9	3.2	4.1	2.9	3.9	5356	5544	51 062	60 904	56	18	57	12
nug27	34.1	4.7	4.9	3.6	4.1	4584	7904	35 982	86 017	61	22	56	14
nug28	34.8	4.2	4.3	3.4	4.3	4502	8937	46 306	103 428	59	23	61	15
nug30	30.5	3.3	4.2	2.7	3.7	10 158	10 353	112 246	132 356	77	23	80	15
rou12	30.8	5.6	6.5	5.4	6.4	268	467	1225	2017	14	6	15	4
rou15	31.5	6.5	7.5	5.3	5.7	556	903	3559	6370	19	8	22	6
rou20	24.5	5.0	4.3	4.4	4.0	1107	2090	7679	18 696	28	10	30	7



(a) *Locations.* (b) *Facilities.*

Figure 5: Graphical description of `Neos4` instance.

3.2 CONSTRUCTIVE METHODS

This section describes constructive algorithms: these algorithms start with an empty permutation and iteratively select a facility and assign it to a free location. In this thesis three constructive algorithms are proposed, the differences among them are in the rules used to select the current facility and its location.

These algorithms we are going to present are *greedy*. Thus, at every iteration they choose the best and immediate benefit.

Greedy algorithms can achieve a permutation which can be very far from the optimal one, therefore they are usually used only to create a *good* starting permutation for more advanced meta-heuristic methods.

3.2.1 *An introductory example*

As an example, we consider `Neos4` instance, as described in 5.2.

Description of the example

Matrices $\boldsymbol{F}$ and $\boldsymbol{D}$ are symmetric and defined as shown below:

$$\boldsymbol{F} = \begin{bmatrix} 0 & 3 & 0 & 2 \\ 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 4 \\ 2 & 1 & 4 & 0 \end{bmatrix} \quad \text{and} \quad \boldsymbol{D} = \begin{bmatrix} 0 & 22 & 53 & 53 \\ 22 & 0 & 40 & 62 \\ 53 & 40 & 0 & 55 \\ 53 & 62 & 55 & 0 \end{bmatrix}$$

In this example, we are going to use different notations. Locations will be denoted with capital *sans-serif* fonts (like X). This is done to convey the geometric idea that locations are points on the 2D-plane. Facilities will be denoted as usual by numbers $1, \ldots, 4$.

A permutation π is the 4-tuple of letters permuted so that π_i is the location of facility i. For instance, $\pi_1 = \mathsf{B}$ means that we place facility 1 into location B.

We can imagine the situation graphically (Fig. 5).

In figure 5a are shown locations of the `Neos4` instance.

Locations (hence, the four point of the plane) are in green, while their distances are in black. Note that locations A and B are fairly close to each other. Instead, the other locations are far from each other, with similar distances. Our naive, basic intuition (but very useful for this section) suggest that facilities with the highest flow should be placed on locations A and B.

On the other hand, figure 5b shows facilities of the same instance. Facilities are in blue, while flows are in black. Thicker lines correspond to higher flows. If the flow is 0, then no line is displayed. As we can see, facility 3 is connected only with facility 4, with a flow of 4 (the highest one). If Thus, our intuition proposes to place facility 3 and 4 close, while it is not necessary to place facilities 2 and 4 close to each other, since they only have a flow of 1.

In fact, the optimal solution of this instance is $\pi = [\mathsf{C}, \mathsf{D}, \mathsf{A}, \mathsf{B}]$ with an objective function value of $z(\pi) = 790$. As we can see, facility 3 is placed on location A and facility 4 on location B. This follows our previously intuition.

Three basic algorithms

So, how can one devise a greedy algorithm? Remember that we want to minimize the quantity

$$\sum_{i=1}^{4}\sum_{j=1}^{4} f_{ij} d_{\pi(i)\pi(j)}$$

Since there must be a starting assignment, let us suppose that we are assigning location A to facility 1. Therefore, we set $\pi_1 = \mathsf{A}$.

Now, how can we choose a second facility and a second location?

Our first idea was to, at each step, minimize both flow and distance at the same time. This was a bad idea, since this choice is very good at the beginning, but later it has a severe impact on magnitude of last assignments.

Therefore, thinking about it, we have identified three different approaches:

FIRST APPROACH We start choosing facility with *maximum* flow from 1, this is 2. Now, the problem is to find the location to put facility 2. We choose location that *minimizes* the sum of distance from location already chosen. In this case, since we only chose A, the location which minimizes the distance from A, this is B. Hence, we set $\pi_2 = \mathsf{B}$.

For the third assignment, we start choosing facility (not already taken) with highest flow from 2, is 4. For locations, we just evaluates the two sums: $d_{\mathsf{AC}} + d_{\mathsf{BC}} = 53 + 40 = 93$ and $d_{\mathsf{AD}} + d_{\mathsf{BD}} = 53 + 62 = 115$. Since we want the minimum distance, we choose C. Therefore, we just set $\pi_4 = \mathsf{C}$.

The last possible assignment is $\pi_3 = \mathsf{D}$. The final permutation built by this approach is $\pi = [\mathsf{A}, \mathsf{B}, \mathsf{D}, \mathsf{C}]$ with objective function value $z(\pi) = 864$.

SECOND APPROACH This approach is fairly similar to the first one with the roles of facility and locations exchanged. So, we start choosing location with the *minimal* distance A, is B. Now, we want to find facility which *maximizes* the flow from already chosen facilities. This is simply facility 2. Thus, we set $\pi_2 = \mathsf{B}$.

For the third location we choose the closest one (not already taken) to B, which is C. Which facility comes now? Let us just do the math: $f_{13} + f_{23} = 0 + 0 = 0$ and $f_{14} + f_{24} = 2 + 1 = 3$. Since we want to maximize the flux, we set $\pi_4 = \mathsf{C}$.

The last one is forced, so $\pi_3 = \mathsf{D}$. The final permutation is again $\pi = [\mathsf{A}, \mathsf{B}, \mathsf{D}, \mathsf{C}]$ with objective function value $z(\pi) = 864$.

THIRD APPROACH This approach is the simplest one. Choose the facilities with *maximum* flow from facility 1 and assign it to the *closest* location

from A. In this case, location B is assigned to facility 2, therefore we set $\pi_2 = \mathsf{B}$.

Now, 2 has a positive flow only with facility 4, so we will chose that. As regards locations, C is the closest one to B. Therefore, we set $\pi_4 = \mathsf{C}$.

Last pair of facility/location is forced, hence $\pi_3 = \mathsf{D}$. So, this approach builds again permutation $\pi = [\mathsf{A}, \mathsf{B}, \mathsf{D}, \mathsf{C}]$ with objective function value $z(\pi) = 864$.

We will transform these three approaches into three different algorithms, which will be named as `Greedy1`, `Greedy2`, `Greedy3`.

Some observations follows:

- All these three methods are quite greedy. They tend to do the best at every step. The difference is that the third one is "greedier" than the other two, since it considers only the current step, and not the past.
- We obtained three times the same final permutation. This is, in general, not true.
- `Greedy3` seems to be the fastest one, since no sum evaluation is necessary and it only looks for minimum/maximum in a vector.
- `Greedy3` is not an innovative strategy, this procedure is similar to the greedy algorithm for TSP proposed in [22].

From now on, we return to the standard notations. Hence, facilities and locations will be denoted by numbers.

We are now going to describe the three algorithms in details.

Every greedy algorithm presented has three parameters on input: q, n_1, n_2 and two on output: $\boldsymbol{p}$ and z_p.

INPUT The three parameters $q \in \{0, 1\}$, $n_1, n_2 \in \{1, \ldots, n\}$ are used for the first assignment. They act as follows:

- If $q = 1$, then algorithm assigns facility n_1 to location n_2. Hence, $p(n_1) = n_2$.
- If $q = 0$, then algorithm tries every possible initial assignment and chooses the best one. In this case, the algorithm does not use n_1 and n_2 and can result in longer running time.

OUTPUT The final permutation is stored as $\boldsymbol{p}$, a vector of n components. z_p represents the objective function value of p, i.e., $z_p = z(\boldsymbol{p})$.

Each of the three greedy algorithms creates two vectors $\boldsymbol{v}$ and $\boldsymbol{l}$, one for facilities and one for locations. These vectors represent the choice of the algorithm at each step.

Therefore, v_i means that in step i the algorithm choose the facility v_i (and same thing for l_i). Thus at each step, after choosing v_i and l_i, the algorithm add a new entry at the final vector $\boldsymbol{p}$, by setting $p_{v_i} = l_i$.

3.2.2 *Greedy1*

This algorithm represents the first approach we discussed in section 3.2.1.

Pseudocode 5 describes the algorithm in the symmetric case with $q = 0$.

As we said lines 1-3 represent the first assignment.

Algorithm 5: `Greedy1` algorithm

Input: n,$\boldsymbol{F}$, $\boldsymbol{D}$,n_1, n_2
Output: p, z_p
/* initialisation */
1 $v_1 \leftarrow n_1$;
2 $l_1 \leftarrow n_2$;
3 $p_{v_1} \leftarrow l_1$;
/* main loop */
4 **for** $i = 1, \ldots, n-1$ **do**
5 Choose v_{i+1}, the facility with maximum flow from f_i;
6 Choose l_{i+1}, the location minimizing $\sum_{j=1}^{i} f_{l_{i+1},j} \cdot d_{s,j}$;
7 $p_{v_{i+1}} \leftarrow l_{i+1}$;
8 **end**
9 Evaluate $z_p = z(p)$;

For the second assignment (and so on), `Greedy1` chooses facility v_2 which maximizes the flow from and to the previous ones (line 5); therefore

$$v_{i+1} = \arg\max \left\{ \max \left\{ f_{v_i s} \right\}, \max \left\{ f_{s v_i} \right\} \right\}.$$

After that, the algorithm chooses location l_{i+1}, where facility v_{i+1} will be placed. Location l_{i+1} is chosen such that it minimizes the total flow sent to facilities already in place (line 6). In other words,

$$l_{i+1} = \arg\min \left\{ \sum_{(k,j)\in\Omega} \left(f_{v_i k} d_{sj} + f_{k v_i} d_{js} \right) \,\middle|\, s \in [n], s \neq l_1 \right\} \tag{35}$$

where $[n] = \{1, 2, \ldots, n\}$ and Ω is the set of pairs facility-location already assigned.

This procedure repeats for $i = 1, \ldots, n-1$ (line 4).

After choosing v_{i+1} and l_{i+1}, `Greedy1` updates the final permutation setting $p_{v_{i+1}} = l_{i+1}$ (line 7).

In the case $q = 0$, the algorithm starts again with every possible choice of n_1 and n_2. The best objective function value is stored.

We implemented `Greedy1` algorithm in Fortran language and made it available on GitHub [26].

3.2.3 *Greedy2*

The second greedy algorithm follows the second approaches in 3.2.1. Hence, it is similar to the first one, with the role of facilities and locations exchanged.

Therefore, using the same notations as before, location l_{i+1} is the closest one to l_i.

As regards facilities, v_{i+1} is chosen as

$$v_{i+1} = \arg\min \left\{ \sum_{(k,j)\in\Omega} \left(f_{sk}\, d_{l_{i+1} j} + f_{ks}\, d_{j l_{i+1}} \right) \,\middle|\, s \in [n], s \neq v_k \,\forall k \right\}.$$

The Fortran code we implemented for `Greedy2` can be found on GitHub.

3.2.4 *Greedy3*

The third greedy algorithm follows the third approach described in 3.2.1.

For facilities, in each step, the algorithm consider the facility (not already chosen) with highest flow to the previous one.

For locations, `Greedy3` considers the closest location (not already chosen) to the current one.

In pseudo code 6 it is shown the `Greedy3` algorithm for symmetric case. The Fortran code we implemented and used can be found on GitHub [26].

Algorithm 6: `Greedy3` algorithm

Input: n,$\boldsymbol{F}$,$\boldsymbol{D}$,n_1, n_2
Output: p, z_p
`/* initialisation */`
$f_1 \leftarrow n_1$;
$l_1 \leftarrow n_2$;
$p_{f_1} \leftarrow l_1$;
`/* main loop */`
for $i = 1, \dots, n-1$ **do**
    v_{i+1} is the facility with maximum flow from v_i;
    l_{i+1} is the closest location to l_i;
    $p_{v_{i+1}} \leftarrow l_{i+1}$;
end
Evaluates $z_p = z(\boldsymbol{p})$.

3.2.5 *Implementation and Comparison*

We compared these three algorithms in order to choose which algorithm use to provide starting permutation for metaheuristic algorithms, that are described in the next chapter.

Table 9 shows percentage of deviation for various instance of QAP. The case $q = 0$ was chosen for every algorithm.

We can see that all these greedy methods are not very precise, especially on instances `lipaxxb`. Nevertheless, as we stated at the beginning, they are commonly use only to built an initial guess for more advanced metaheuristic algorithms.

Moreover, note that `Greedy3` is much faster than the other two (often with a difference of one order of magnitude), as a confirm of what we supposed. Thus, we will use it for metaheuristic methods presented in the next chapter, despite it shows larger PD values than `Greedy1`.

Table 9: Comparison of Greedy algorithms. PD is the percentage deviation from the best known solution, t is the running time in seconds.

Instance	Greedy1		Greedy2		Greedy3	
	PD	t	PD	t	PD	t
lipa20a	3.18	0.0	5.89	0.0	3.10	0.0
lipa20b	0.00	0.0	29.68	0.0	0.00	0.0
lipa30a	2.78	0.1	4.17	0.1	2.62	0.0
lipa30b	0.00	0.1	29.18	0.1	12.03	0.0
lipa40a	2.11	0.5	3.49	0.5	2.03	0.0
lipa40b	15.08	0.5	30.53	0.5	19.33	0.1
lipa50a	1.74	1.4	3.12	1.3	1.72	0.1
lipa50b	16.58	1.4	30.00	1.4	22.50	0.1
lipa60a	1.58	3.3	2.74	3.2	1.60	0.2
lipa60b	21.19	3.3	32.14	3.2	23.21	0.3
lipa70a	1.45	6.9	2.49	6.8	1.37	0.5
lipa70b	21.11	6.9	32.46	7.0	25.69	0.5
lipa80a	1.28	13.2	2.25	12.9	1.22	0.8
lipa80b	22.23	13.2	33.80	13.0	26.18	0.8
lipa90a	1.17	23.5	1.99	22.3	1.12	1.3
lipa90b	23.31	23.1	33.51	22.6	25.23	1.2
sko42	9.88	0.6	32.95	0.6	18.91	0.1
sko49	8.22	1.3	31.46	1.2	16.40	0.1
sko56	8.44	2.4	31.14	2.4	17.52	0.1
sko64	7.34	4.5	28.48	4.4	15.41	0.2
sko72	7.81	8.0	26.65	7.6	15.08	0.4
sko81	6.51	14.2	26.39	13.5	14.44	0.6
sko90	7.10	23.6	24.95	23.0	13.90	0.9
sko100a	6.31	39.5	22.88	38.0	14.30	1.4
sko100b	6.60	38.8	23.82	37.0	13.52	1.2
sko100c	6.91	38.7	24.26	36.8	13.75	1.3
sko100d	6.04	40.0	23.57	37.9	13.13	1.3
sko100e	6.69	39.8	23.98	37.1	13.42	1.2
sko100f	6.53	40.3	23.87	38.4	14.10	1.4

4 METAHEURISTIC ALGORITHMS

The word heuristic has its origin in the old Greek word ενρισκείν (*heuriskein*), which means the art of discovering new strategies (rules) to solve problems. The term *metaheuristic* was introduced by Fred W. Glover in 1986 [18, p. 541], a professor of University of Colorado. In the referred article *Future paths for integer programming and links to artificial intelligence*, he used this term only once in all the document:

> Tabu search may be viewed as a "meta-heuristic" superimposed on another heuristic.

In fact, the Greek suffix *meta* means "upper level methodology".

Metaheuristic algorithms were designed in order to solve problems too difficult for the heuristic algorithms. Hence, metaheuristics provide "acceptable" solutions in a reasonable time for solving hard and complex problems in science and engineering. This explains the significant growth of interest in metaheuristic domain. Unlike exact optimization algorithms, metaheuristics do not guarantee the optimality of the obtained solutions [35].

Figure 6: Fred Glover

There are two types of metaheuristic algorithms:

POPULATION BASED They can be viewed as an iterative improvement in a population of permutations. At first the population is initialized, then a new one is generated and finally these populations are integrated into a new one using a selection procedure. The search process is stopped when a stopping criterion is satisfied. Some of Population based algorithms are Ant Colony Optimization, Genetic Algorithms, Memetic Algorithms [35].

SINGLE-SOLUTION BASED While solving optimization problems, single-solution based metaheuristics improve a single solution. They can be viewed as "walks" through neighborhoods or search trajectories through the search space of the problem. Tabu Search, Simulated Annealing, GRASP, Variable Neighborhood Search belong to this class of metaheuristics [35].

There are three central concepts in metaheuristics:

NEIGHBORHOOD We define a neighborhood of a feasible solution $\pi \in S_n$ as a generic function $\mathcal{N}\colon \pi \to \mathcal{P}(S_n)$ that maps a solution to a set of solutions.

INTENSIFICATION Exploitation of the best solutions. Often this is achieved by doing a local search starting from the current solution.

DIVERSIFICATION Exploration of the search space, in order to escape from local minimum.

We are going to describe three different metaheuristic algorithms:

- Tabu Search. We implemented the original algorithm in [31].

- Ant Colony Optimization. We elaborated a new strategy for the diversification phase with respect to [15].
- Variable Neighborhood Search. We did not find an implementation of this algorithm for QAP in literature; hence, we devised a new implementation *ex novo*.

4.1 TABU SEARCH

Introduction

The history of Tabu Search (TS) is described in the following table.

Year	Event
1986	Tabu Search algorithm is proposed by Glover [18]
1990	Skorin-Kapov implements tabu search for QAP [31]
1991	Éric Taillard implements *robust tabu search* for QAP [33]
1994	Battiti and Tecchiolli [1] implements *reactive tabu search* for QAP
2011	Misevicius [27] implements *iterated tabu search* (ITS) for QAP

In his famous paper of 1986, Fred Glover introduced Tabu[1] Search algorithm as follows:

> From an AI point of view, *tabu search* deviates to an extent from what might be expected of intelligent human behavior. Humans are often hypothesized to operate according to some sort of random (probabilistic) element which promotes a certain level of "inconsistency". The resulting tendency to deviate from a charted course, sometimes regretted as a source of error, can also prove a source of gain. Such a mechanism has been postulated to be useful, even fundamental, to human ingenuity.

TS is one of the most studied and used algorithm for QAP. In several cases, the methods of this class provide solutions very close to optimality and are among the most effective, if not the best, to tackle the difficult problems at hand. Therefore, these successes have made TS extremely popular [16].

The main idea of TS is to *forbid* certain moves. This is done in order to escape from local optima. To do this, a *tabu list* was adopted. It is an array of size s (called *tabu tenure*), that stores the past evolution of the algorithm.

Figure 7: Jadranka Skorin-Kapov

Since there is a rich variety of strategies and tools, there is a lot of freedom in the implementation of a TS.

The first TS algorithm for QAP was presented by Jadranka Skorin-Kapov, Stony Brook University, New York. Her algorithm starts with an initial permutation π and begin an intensification phase using a local search algorithm; she used the neighborhood N_2, as defined in section 3.2. She forbid the 2-exchange of two units that were swapped during the previous s iterations, inserting them into the tabu list, i.e., every time a 2- exchange is performed, the algorithm inserts the two indices swapped into the tabu list.

After one year, the *robust tabu search* (roTS) was defined by Éric Taillard, École polytechnique fédérale de Lausanne (Lausanne, Switzerland). Based on Skorin-Kapov work, they used the same neighborhood but a different tabu list. For every facility i and location j, the latest iteration at which the facility i occupied location j is recorded. A 2-exchange $\pi_{i_1 i_2}$ is tabu if both i_1 and i_2 are assigned to locations they had occupied in the last s iterations. The tabu tenure varies in a cyclic random manner between a lower value

[1] The name comes from Polynesian word *tàpu* that means prohibited.

$s_{\min} = \lfloor 0.9n \rfloor$ and an high value $s_{\max} = \lceil 1.1n \rceil$. This value is updated every $2 \cdot s_{\max}$ iterations.

In 1994, Roberto Battiti and Giampietro Tecchiolli, Instituto Nazionale di Fisica Nucleare (Trento, Italy), introduced the reactive tabu search (ReTS). They still used neighborhood N_2, but the algorithm evolved in a more complex way. In fact, ReTS *reacts* during the evolution of the search by increasing the tabu tenure when a solution is repeated along the search, and decreasing it if no repetition occurs for a certain number of iterations . They used hashing functions, binary trees, bucket lists as tool to store the solution and to check if a neighbour solution was already visited [5]. The numerical results show that ReTS is competitive with ro-TS in terms of number of iterations performed to reach the best known solution [5].

In 2011, Alfonsas Misevicius , Kaunas University of Technology, (Kaunas, Lithuania) implemented [27] *iterated tabu search* (ITS). He combined the roTS with a special type of mutation. As regards tabu tenure, he differentiated from Taillard's approach. The tabu tenure s various s in a deterministic way: each time it reached a value $s_{\max}$, it drops to a lower value $s_{\min}$. For small size problems ($n \leq 50$) he used $s_{\min} = \lfloor 0.2n \rfloor$ and $s_{\max} = \lceil 0.4n \rceil$. For larger problems ($n > 50$) he used $s_{\min} = \lfloor 0.1n \rfloor$ and $s_{\max} = \lceil 0.2n \rceil$.

Now we are going to describe the algorithm in details.

Description of the algorithm

In our implementation, we followed the main idea of Skorin-Kapov's work.

We consider two types of memories:*short-term memory* and *long-term memory*.

SHORT-TERM MEMORY This memory consists of a tabu list with tabu tenure s. This list is used to prevent revisiting previously visited solutions. This is done in order to achieve diversification.

LONG-TERM MEMORY This memory stores information on the visited solutions during the search. The idea of long-term memory is to record the frequency with which each movement has been realized since the beginning of the algorithm. In our implementation, long-term memory provide an intensification phase.

An other important ingredient is the *aspiration criterion*. It is essentially a condition that, if holds, allows tabu solutions to be accepted. The most common criterion is to accepts the tabu moves that generate solutions better than the best one found so far.

Pseudo code 7 sketches the idea of the algorithm.

The short-term memory is an array of size s where the best local moves performed are stored. Every time that an improvement of the objective function by a 2-exchange occurs, the move performed is stored in the tabu list, hence it is prohibited. Since the dimension of list is finite, when the list is full the next move will replace the oldest one (this is why it forms the short-term memory: it records only the s previous moves).

We implemented long-term memory as an $n \times n$ matrix $\boldsymbol{M} = (m_{ij})$. Each entry of $\boldsymbol{M}$ is related to the quality of the 2-exchange done, e.g., if the 2-exchange $\{1,2\} \to \{2,1\}$ improves the current solution, then we store this information on $\boldsymbol{M}$, updating the matrix as follows:

$$M_{12} \leftarrow M_{12} + 1 \quad \text{and} \quad M_{21} \leftarrow M_{21} + 1.$$

INPUT Tabu tenure s, penalty parameter μ, maximum time $t_{\max}$.

OUTPUT The obtained solution p_{best} and its objective function value z_{best}.

Algorithm 7: Tabu search

```
   Input: n, F, D, μ, s, t_max
   Output: p_best, z_best
   /* initialization                                                    */
1  Build an initial solution p with objective function z_p = z(p) by Greedy3 algorithm;
2  z_best ← z_p;
3  p_best ← p;
4  M ← 0;
5  D̃ ← D;
   /* main loop                                                         */
6  Call CPU_TIME t;
7  while t < t_max do
       /* improvement of the current solution                          */
8      d_min ← 0;
9      Starting from p, search an admissible local 2-optimum p_temp ∈ N_2(p) with
         objective function value z_temp. This is obtained by a 2-exchange of indices (j_1, j_2);
10     d_min ← Δ(p; j_1, j_2);
       /* update short-term memory                                     */
11     Add the pair (j_1, j_2) to tabu list;
12     if d_min < 0 then
13         z_p ← z_temp;
14         p ← p_temp;
           /* update long-term memory                                  */
15         m_{j_1 j_2} = m_{j_1 j_2} + 1;
16         m_{j_2 j_1} = m_{j_2 j_1} + 1;
17         if z_p < z_best then
18             z_best ← z_p;
19             p_best ← p;
20         end
21         Call CPU_TIME t;
22         Go to 7;
23     end
       /* create a new solution using long-term memory                 */
24     D̃ ← D̃ + μM ;
25     Use Greedy3 with matrices F and D̃ to create a new permutation p with objective
         function z_p;
26     Call CPU_TIME t;
27 end
```

Initialization

At first algorithm builds an initial permutation. In our implementation it is provided by the by `Greedy3` method (line 1). The output permutation and its objective function provided by `Greedy3` are denoted by p and z_p.

The long-term matrix is initialized as the null matrix (line 4). A copy of distance matrix $\boldsymbol{D}$ is stored in matrix $\tilde{\boldsymbol{D}}$ (line 5).

Improvement of the current solution

The algorithm looks for an improvement of the current solution (lines 9-10). This is done in our implementation by a modified version of `2optBest` algorithm.

The difference with `2optBest` algorithm is that not every move is allowed. At each step, TS controls if the current indices i_1 and i_2 belong to tabu-list. If this is the case, `2optBest` variant skips to the next pair of indices.

It is possible to overcome the tabu restriction for a pair (j_1, j_2) if the 2-exchange $\{j_1, j_2\} \rightarrow \{j_2, j_1\}$ improves the best solution obtained by far p_{best}.

The term $d_{\min}$ is stored (line 10). It is the best improvement of the objective function value obtained.

If $d_{\min} = 0$, this means that the best 2-exchange found does not improve the current solution. Therefore, p_{temp} is an admissible local optimum. The term *admissible* means that it does not belong to the tabu list, or it belongs but provides an improvement of the best permutation found so far p_{best}.

On the other hand, if $d_{\min} < 0$, then there is indeed an improvement of the objective function value. Suppose that $\{j_1, j_2\} \rightarrow \{j_2, j_1\}$ is the best admissible 2-exchange. Then, the 2-exchange is applied and its objective function value is updated (lines 13-14).

Moreover, if the permutation after the 2-exchange is the best found so far, the algorithms update p_{best} and z_{best} (lines 18-20).

This procedure repeats until the actual permutation p cannot be improved any more ($d_{\min} \geq 0$).

Updating of the memories

The short-term memory is updated: the two indices j_1 and j_2 enter in the tabu list (line 11).

If the tabu list is full, it starts again by substituting the first two index of the list with j_1 and j_2. This idea was taken from Skorin-Kapov [31].

As regards long-term memory, the matrix $\boldsymbol{M}$ is updated by increasing m_{j_1,j_2} and m_{j_2,j_1} by 1 (lines 15-16). This memory, unlike tabu list, is never reset its value. Therefore, the indices j_1 and j_2 remain in $\boldsymbol{M}$ until the end of the algorithm.

Creation of a new permutation

At this point the matrix $\tilde{\boldsymbol{D}}$ is updated as $\tilde{\boldsymbol{D}} \leftarrow \tilde{\boldsymbol{D}} + \mu \boldsymbol{M}$ where μ is an input penalty parameter (line 24).

In our runs, we set

$$\mu = k \cdot \frac{1}{n^2} \sum_{j=1}^{n} \sum_{i=1}^{n} d_{ij}. \tag{36}$$

This allow us to adapt the magnitude of $\mu \boldsymbol{M}$ to the one of matrix $\boldsymbol{D}$;

If $\mu > 0$, the algorithms increases distance between locations frequently swapped, therefore tries to achieve diversification.

On the other hand, if $\mu < 0$, then distance between locations frequently swapped decreases, hence the algorithm stimulates frequent moves to be chosen.

Finally, a constructive algorithm is used to create another solution p^* from the updated matrix $\boldsymbol{D}$. Here the algorithm starts again, setting $p \leftarrow p^*$.

Note that the only `Greedy3` algorithm uses the updated matrix $\tilde{\boldsymbol{D}}$, the rest of the algorithm still uses the original distance matrix $\boldsymbol{D}$.

Implementation and parameters calibration

We implemented algorithm 7 in Fortran language. The code can be found on GitHub [26].

Since the dimension of the instances are very different, we are going to do two separate calibrations: one for small dimensions ($n < 50$) and an other one for large dimensions ($n \geq 50$).

The algorithm stops whenever the execution time exceeds the input value $t_{\max}$. We chose $t_{\max} = 30\,\text{s}$ for $n < 50$ and $t_{\max} = 60\,\text{s}$ for $n \geq 50$.

Finally, we run the code on a i7-9750H processor with 2.60 GHz clock frequency.

The parameters we have to calibrate are

- the parameter k used in (36);
- the size of tabu list s; if its value is too small, cycling may occur in the search process while if its value is too large, appealing moves may be forbidden and leading to the exploration of lower quality solutions, producing a larger number of iterations to find the solution desired.

To calibrate them, we vary the parameters in a reasonable range and then we took the values providing best objective function values at average. Like the previous chapter, we evaluated the Percent Deviation (PD) from the optimal solution.

After some tests we found that the bests values of k were between -1 and -2, hence we explored it more deeply. Finally, the range of tested parameter values is

- $k \in \{1, -1, -1.2, -1.4, -1.6, -1.8, -2, -5, -10, -20\}$
- $s \in \{8, 10, 15, 20, 25, 50, 100, 200\}$

The following pages contain tables where we highlighted in green the optimal solution (or the best known solution) and in blue good solutions, even if they were not the optimal.

Small dimension

Instances used to calibrate are

`tai12a` We achieved the optimum for several (k, s) (table 10) . In most of the cases, the percent error of the solution is less than 4%.

`chr20c` We achieved the optimum for $s = 20, 50$ and $k = -1.8$ (table 11) . Other good result were obtained for $s = 150$ and $k = -1.8, -2, -5, -10, -20$.

`nug30` The best solution we achieve (Table 12) has a percent error of 0.03% with $k = -5, -10, -20$ and $s = 20$. All other solutions are pretty good, since they always are under 2% of percent error, except for $k = 1$.

Large dimension

We used the following instances:

`lipa60b` As we can see in table 13 we reached the optimal solution 7 times: for $\mu = -1$ and $s = 25$, 50, 100, 150, for $\mu = -1.2$ and $s = 8$ and finally for $s = 150$ and $\mu = -1.8$. The instance is asymmetric.

`wil100` The optimal solution is not known. Every solutions we found has less than 1% of error from the best known solution (which is 64).

`esc128` As we can see in table 15, We reached the optimum many times. Notice that TS provides optimal solution for every s if $k = -1.8, -2, -5, -10$.

s	k									
	1	−1	−1.2	−1.4	−1.6	−1.8	−2	−5	−10	−20
8	6.38	0	0	0	0	0	0	3.84	3.84	3.84
10	6.30	0	0	0	0	0	0	2.08	0	0
15	6.30	0	0	0	0	0	0	0	0	0
20	6.30	0	0	0	2.80	2.80	2.08	2.08	2.08	2.08
25	6.30	2.80	2.80	2.08	0	2.08	0	0	0	0
50	6.30	3.84	3.84	2.80	2.80	2.08	3.84	0	0	0
100	6.30	0	2.08	2.08	2.08	2.08	0	2.08	2.08	2.08
150	6.30	6.89	6.89	2.08	2.08	2.08	0	0	2.80	2.80

Table 10: Tabu Search for `Tai12a`

s	k									
	1	−1	−1.2	−1.4	−1.6	−1.8	−2	−5	−10	−20
8	41.61	6.82	20.11	6.04	18.92	13.75	13.75	13.75	13.75	13.75
10	41.61	21.67	4.72	18.22	15.90	19.37	19.37	19.37	19.37	19.37
15	39.57	20.72	18.58	11.51	24.62	18.98	18.98	18.98	18.98	18.98
20	39.57	18.88	20.18	19.30	4.72	0	19.37	19.37	19.37	19.37
25	39.57	18.53	19.81	22.43	19.37	20.80	20.80	20.80	20.80	20.80
50	39.57	13.75	15.90	15.20	16.69	0	28.72	28.72	28.72	28.72
100	36.22	17.83	18.82	18.22	18.53	20.80	20.61	20.61	20.61	20.61
150	36.22	26.79	18.89	18.22	16.39	4.74	4.74	4.72	4.72	4.72

Table 11: Tabu Search for `Chr20c`

Conclusions

The choice of $k = 1$ did not perform well in any instance. This reflects the fact that, for our TS algorithm, intensification is a better strategy than diversification.

For small dimension, we decided to choose $s = 20$ and $k = -1.8$, since it provides the optimal solution for `Chr20c` and for `Tai12a` and `Nug30` the sum of the error is minimal: only 2.9%.

For large dimension, we chose $s = 20$ and $k = -1.4$, since it provides the optimal solution for `Lipa60b` and `Esc128`, while only an error of 0.48% for `Wil100`.

s	k									
	1	−1	−1.2	−1.4	−1.6	−1.8	−2	−5	−10	−20
8	2.45	0.98	1.27	0.78	1.44	0.95	0.95	1.96	1.96	1.96
10	2.45	1.08	0.36	0.78	1.40	1.67	1.67	1.24	1.24	1.24
15	2.45	0.69	1.08	0.98	0.49	0.88	0.88	1.34	1.34	1.34
20	2.45	0.85	0.88	1.47	1.47	0.82	1.86	0.10	0.10	0.10
25	2.45	0.95	0.98	0.59	1.24	0.69	0.91	0.88	0.88	0.88
50	2.45	1.05	0.62	1.14	0.46	0.75	0.69	1.01	1.01	1.01
100	2.02	0.85	0.98	0.52	1.27	1.11	1.18	0.52	0.52	0.52
150	2.45	1.08	1.27	1.24	0.69	1.21	1.08	0.95	0.95	0.95

Table 12: Tabu Search for `Nug30`

s	k									
	1	−1	−1.2	−1.4	−1.6	−1.8	−2	−5	−10	−20
8	19.90	19.13	0	19.03	19.15	19.05	19.05	19.05	19.05	19.05
10	19.90	19.19	18.92	18.82	18.91	19.02	19.02	19.02	19.02	19.02
15	19.54	19.09	18.90	19.44	18.92	19.11	18.92	18.92	18.92	18.92
20	19.90	19.09	19.03	0	18.89	19.14	19.21	18.89	18.89	18.89
25	19.90	0	18.90	19.44	18.92	19.11	18.92	18.92	18.92	18.92
50	19.90	0	19.14	18.86	18.90	19.03	19.03	19.03	19.03	19.03
100	19.83	0	19.00	19.06	19.03	19.08	19.03	19.08	19.08	19.08
150	19.58	0	18.87	19.10	19.04	0	19.042	19.04	19.04	19.04

Table 13: Tabu Search for `Lipa60b`

s	k									
	1	−1	−1.2	−1.4	−1.6	−1.8	−2	−5	−10	−20
8	0.69	0.60	0.69	0.54	0.65	0.75	0.75	0.47	0.47	0.47
10	0.68	0.60	0.69	0.64	0.65	0.77	0.63	0.47	0.47	0.47
15	0.73	0.60	0.72	0.60	0.65	0.82	0.65	0.59	0.59	0.59
20	0.83	0.60	0.72	0.48	0.65	0.67	0.71	0.79	0.79	0.79
30	0.87	0.60	0.74	0.62	0.58	0.69	0.66	0.72	0.72	0.72
50	0.67	0.60	0.74	0.62	0.58	0.69	0.66	0.72	0.72	0.72
100	0.59	0.60	0.62	0.54	0.65	0.60	0.68	0.55	0.55	0.55
150	0.66	0.60	0.70	0.65	0.58	0.64	0.69	0.69	0.69	0.69

Table 14: Tabu Search for `Wil100`

s	k									
	1	−1	−1.2	−1.4	−1.6	−1.8	−2	−5	−10	−20
8	9.38	3.31	3.13	3.13	0	0	0	0	0	0
10	9.38	3.13	0	0	0	0	0	0	0	0
15	9.38	3.13	0	0	0	0	0	0	0	0
20	9.38	3.13	0	0	0	0	0	0	0	0
25	9.38	3.13	0	0	0	0	0	0	0	0
50	9.38	3.13	0	0	0	0	0	0	0	0
100	9.38	3.13	0	0	0	0	0	0	0	0
150	9.38	3.13	3.13	3.13	9.38	0	0	0	0	0

Table 15: Tabu Search for `Esc128`

4.2 ANT COLONY OPTIMIZATION

Ant Colony Optimization algorithm (ACO) was initiated by Dorigo in 1991. The principle of this method is based on the way Argentine ants *Iridomyrmex humilis* search for food and find their way back to the nest [19]. At first, ants explore the area surrounding their nest at random. Then, during the return trip, ants leave a pheromone trail on the ground, in order to guide other ants toward the source of food. The quantity of pheromone left depends on the amount of food found. Pheromone trail evaporates if no one pass there any more.

This algorithm is population based. Hence, at first m initial solutions (the ants) are generated. Then, it comes intensification: each ant is improved in a local search phase. Finally, before the start of the next iteration, the pheromone trail is updated reflecting the "experience" of the ants.

4.2.1 *Hybrid Ant System*

Inspired by Gambardella, Taillard and Dorigo [15] we implemented a variant of ACO known as Hybrid Ant System (HAS). Nevertheless, we will still refer to this algorithm as ACO.

In this implementation the pheromone trail is a $n \times n$ matrix $\boldsymbol{T} = (\tau_{ij})$, where the entry τ_{ij} measures how good is assigning facility i to location j. Pseudo code 8 outlines main steps of ACO.

4.2.2 *Implementation*

Pheromone trail initialization

In the original algorithm by Gambardella et al [15], pheromone trail matrix $\boldsymbol{T}$ was initialized by setting every component to the same value $\tau_0 = \frac{1}{100 z_{\text{best}}}$, where z_{best} is the best know value of the objective function.

Instead, we tried another approach: two matrices $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ are used and then combined to form matrix $\boldsymbol{T}$.

First, the two matrices $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ are initialized to 0.

Then, n initial permutations are built, by setting

$$\pi_i = [i, i+1, \ldots, n, 1, 2, \ldots, i-1] \qquad \forall i \in 1, \ldots, n$$

After that, for every permutation π_i, `2optBest` algorithm is applied with the following remarks:

- Every 2-exchange made (hence, every improvement of the solution) is stored: if indices p and q of π_i are swapped, therefore we update the matrix $\boldsymbol{M}_2$ as follows:

  $$M_2(p, \pi_i(p)) \leftarrow M_2(p, \pi_i(p)) - 1, \qquad M_2(q, \pi_i(q)) \leftarrow M_2(q, \pi_i(q)) - 1,$$
  $$M_2(p, \pi_i(q)) \leftarrow M_2(p, \pi_i(q)) + 1, \qquad M_2(q, \pi_i(p)) \leftarrow M_2(q, \pi_i(p)) + 1.$$

  Finally, since we want $\boldsymbol{M}_2$ to be positive, we set $\boldsymbol{M}_2 \leftarrow \boldsymbol{M}_2 + \min(\boldsymbol{M}_2)$.

- At the end of local search procedure, a final permutation $\tilde{\pi}_i$ is found. Thus, $\boldsymbol{M}_1$ is updated as follows:

  $$M_1(r, \tilde{\pi}_i(r)) \leftarrow M_1(r, \tilde{\pi}_i(r)) + 1 \quad \forall r \in \{1, \ldots, n\}.$$

Algorithm 8: ACO algorithm

Input: n, $\boldsymbol{F}$, $\boldsymbol{D}$, $t_{\max}$
Output: p_{best}, z_{best}

Initialize pheromone trail matrix $\boldsymbol{T}$;
while $t < t_{\max}$ **do**
  Generate m random initial permutation $\pi^1, \ldots, \pi^m$;
  Improve $\pi^1, \ldots, \pi^m$ with `2optFirst` algorithm;
  Let π_{best} be the best permutation among $\pi^1, \ldots, \pi^m$;
  `/* solution manipulation */`
  for $k = 1, \ldots, m$ **do**
    Apply R 2-exchanges to π^k to obtain $\hat{\pi}^k$;
    Apply `2optFirst` to $\hat{\pi}^k$ to obtain $\tilde{\pi}^k$;
    `/* intensification */`
    if *intensification is active* **then**
      $\pi^k \leftarrow$ best permutation between π^k and $\tilde{\pi}^k$;
    else
      $\pi^k \leftarrow \tilde{\pi}^k$
    end
  end
  Deactivate intensification;
  if *exists k such that* $z(\pi^k) < z_{\text{best}}$ **then**
    Update π_{best};
    Activate intensification;
  end
  `/* pheromone trail updating */`
  Update pheromone trail matrix $\boldsymbol{T}$;
  `/* diversification */`
  if *S loops have been performed without improving* π_{best} **then**
    Perform a diversification: $\boldsymbol{T} \leftarrow \boldsymbol{0}$;
  end
end

Finally, pheromone trail matrix $\boldsymbol{T}$ is built summing $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$:

$$\boldsymbol{T} \leftarrow \boldsymbol{M}_1 + \boldsymbol{M}_2.$$

In a nutshell, τ_{ij} tells us how good is the assignment $\pi(i) = j$.

Initialization of solutions

As in [15], m random permutations are chosen. Each permutation is optimized using a local search procedure. Gambardella [15] used a variant of first improvement algorithm. Instead, we used our `2optFirst` algorithm.

Manipulation of solutions

In [15], a number of R 2-exchanges are applied to each permutation π^k. These operations provides m new permutations $\hat{\pi}^1, \ldots, \hat{\pi}^m$. We followed the same procedure. These R swaps are performed as follows:

First, an index r is chosen, randomly between 1 and n.

Then, a second index $s \neq r$ is chosen and the elements π^k_r and π^k_s are swapped in the current solution π^k.

The second index s is chosen according to following policy:

- With probability given by a parameter q, s is chosen such that $\tau_{r\pi_s} + \tau_{s\pi_r}$ is maximum. This means that s is the best 2-exchange for r we can do, according to pheromone trail matrix $\boldsymbol{T}$.
- With probability $(1 - q)$, s is chosen with a probability proportional to the values contained in $\boldsymbol{T}$. More precisely, s is chosen with probability

$$\frac{\tau_{r\pi_s} + \tau_{s\pi_r}}{\sum_{j \neq r} \left(\tau_{r\pi_j} + \tau_{j\pi_r} \right)}. \tag{37}$$

Note that setting $M_2 \geq 0$ allows τ_{ij} to be positive. Therefore, expression (37) is indeed a density of probability.

After the 2-exchange, `2optfirst` algorithm is applied to every permutation $\hat{\pi}^k$, obtaining m 2-optima: $\tilde{\pi}^1 \ldots, \tilde{\pi}^m$.

Intensification

The intensification mechanism is activated when the best solution produced so far π_{best} has been improved. Intensification remains active while at least one solutions is improved during an iteration. Therefore:

- If intensification is active, then each permutation starts its next iteration as the best permutation between π^k and $\tilde{\pi}^k$.
- If intensification is not active, then the permutation is maintained as $\tilde{\pi}^k$.

Pheromone trail update

Pheromone trail is updated by taking into account only the best solution π_{best}.

Firstly, the pheromone trail $\boldsymbol{T}$ is weakened by setting $\tau_{ij} = (1 - \alpha) \cdot \tau_{ij}$ where $0 < \alpha < 1$ is a parameter that controls *evaporation* of the trail. A value of α close to 0 implies that pheromone is more persistent, while a value close to 1 implies high degree of evaporation (thus, a shorter memory of the system).

Secondly, $\boldsymbol{T}$ is reinforced by considering the best permutation obtained so far π_{best}. In [15], authors update the pheromone trail $\boldsymbol{T}$ as follows:

$$\tau_{i\pi_{\text{best}}(i)} \leftarrow \tau_{i\pi_{\text{best}}(i)} + \frac{\beta}{z_{\text{best}}} \qquad \forall i \in \{1, \dots, n\}. \tag{38}$$

Instead, we followed an other approach: The algorithm builds matrices $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ in the same way of pheromone trail initialization, but only considering π_{best} instead of all the m permutations.

Finally, we update $\boldsymbol{T}$ as follows:

$$\boldsymbol{T} \leftarrow \boldsymbol{T} + (\boldsymbol{M}_1 + \boldsymbol{M}_2)\,. \tag{39}$$

Diversification

Diversification mechanism is activated if, during the last S loops, no improvement to the best generated solution is detected. Diversification consists in erasing all the information contained in the pheromone trail and in randomly generating other m solutions (line 22).

Complexity

The complexity of the algorithm can be evaluated as follows: most time consuming part of the algorithm is the local search procedure, which has a computational cost of $O(n^3)$ operations. This is repeated Im times, where I is the number of loops performed. Hence, the total cost of ACO is $O\left(Imn^3\right)$.

4.2.3 *Parameters calibration*

Table 16 shows us the parameters that must be calibrated, the tested ranges and the chosen values.

Table 16: Parameters of ACO algorithm

Name	Symbol	Value	Range tested
Probability	q	0.85	$\{0.15, 0.50, 0.85\}$
Number of 2-exchanges performed	R	2	$\{5, 10, 15\}$
Evaporation	α	0.25	$\{0.15, 0.25\}$
Maximum number of non improving loops	S	$5n$	$\{n, 2n, 5n\}$
Number of ants	m	10	$\{5, 10, 20\}$

We tested various values of each parameters for 6 instances: `tai12a`, `chr20c`, `nug30`, `lipa60b`, `wil100` and `esc128`.

In this case, we did not make any distinctions between small and large dimensions instances, since no substantially differences were found.

We chose a number of ants m equal to 10, to limit the computation time required for the algorithm.

The parameter S was set equal to $5n$. For $S \leq n$, the algorithm provides poorly solution with high percentage deviation, even for small dimension instances.

As regards the other parameters, they were experimentally found to be good and robust for the instance tested, providing an output permutation with less than 1% of PD.

4.3 VARIABLE NEIGHBORHOOD SEARCH

Introduction

The Variable Neighborhood Search algorithm (VNS) was introduced by Pierre Hansen and Nenad Mladenovic in 1997 [28] for the Traveling Salesman Problem (TSP). The literature of VNS for QAP is not as extended as for Tabu Search or Ant Colony Optimization.

The main idea of the algorithm is to use several neighborhood structures and, when a local optimum is found, to move from one to another.

Now, we present some essential definitions and facts about neighborhood structures. More detail can be found in [16, Ch. 3].

Let us start with two definitions.

Definition 4.1 (Neighborhood structure)**.** A function $\mathcal{N}\colon S_n \to \mathcal{P}(S_n)$ that maps a feasible solution $\pi \in S_n$ to a set of a solutions $\mathcal{N}(\pi) \subseteq S_n$ is called a *neighborhood structure*.

Definition 4.2 (Local optimum)**.** A solution π^* is called a *local optimum* with respect to the neighborhood structure $\mathcal{N}$ if there is no feasible solution $\sigma \in \mathcal{N}(\pi^*)$ such that $z(\sigma) \leq z(\pi^*)$.

Essentially, we can consider $\mathcal{N}(\pi)$ as a set of permutations *close* to π.

Therefore, the operator $\mathcal{N}$ allows us to obtain new feasible solutions realizing a determined operation on an initial solution.

Note that neighborhoods N_r defined by (3.2) are an example of neighborhood structures, since they map a permutation π into the set of permutations that can be reached by π by a r-exchange.

4.3.1 *Local search*

Description of the algorithm

This procedure is a generalization of the local search algorithms discussed in section 3.1.

Pseudo code 9 shows the local search algorithm.

Algorithm 9: Local search procedure.

Input: An initial permutation s_1, a neighborhood structure $\mathcal{N}$
Output: A local optimum s^* w.r.t. $\mathcal{N}$

```
1 repeat
2 |   Examine N(s_1);
3 |   if exists s_2 ∈ N(s_1) such that z(s_2) ≤ z(s_1) then
4 |   |   s_1 ← s_2;
5 |   else
6 |   |   s* ← s_1;
7 |   |   Exit;
8 |   end
9 until a local optimum s* is found;
```

Imagine to fix a neighborhood structure $\mathcal{N}$, with an initial permutation s_1.

Then, the local search starts, and the algorithm exhaustively examines every permutation in $\mathcal{N}(s_1)$ (line 4).

After investigating $\mathcal{N}(s_1)$, there are two possibilities:

1. A permutation s_2 such that $z(s_2) < z(s_1)$ is found.

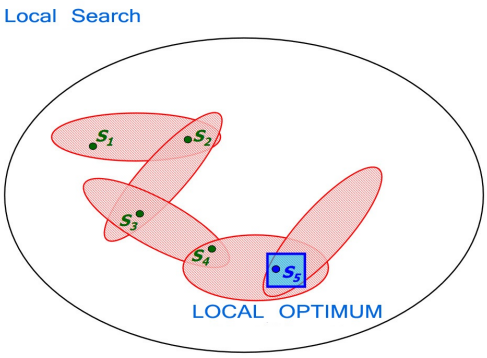


Figure 8: Local search procedure, starting from initial point s_1.

2. No better permutation is found. Therefore, s_1 is a local optimum in $\mathcal{N}_1$, i.e., $z(s_1) = \min\{z(p) \mid p \in \mathcal{N}(s_1)\}$.

In the first case, the algorithm repeats setting $s_1 \leftarrow s_2$. In the second case, the algorithm stops (lines 4-8).

Since the number of permutations is finite, in a finite number of steps the algorithm provides a local optimum with respect to the neighborhood structure $\mathcal{N}$ (line 9).

Note that in Section 3.1, the neighborhood structure $\mathcal{N}$ of local search algorithm was N_r, for $r = 2$ or $r = 3$.

Figure 8 sketches an idea of the local search procedure. The pink oval represents the neighborhoods structure. The local search starts from s_1, looks for a "better" solution on $\mathcal{N}(s_1)$ and finds s_2. Then, it continues until arriving at s_5, which is a local optimum w.r.t. $\mathcal{N}$. Thus, it stops.

Two final remarks:

- The algorithm stops when it finds a local optimum.
- The search is limited to only one neighborhood structure.

The first point suggests us that local search procedure should be used as a part of an intensification method, combined with other technique to escape from local optima.

As regards the second point, the algorithm that solves this problem is the Variable neighborhood Search (VNS).

4.3.2 *Variable neighborhood Search*

The basic idea of Variable Neighborhood Search algorithm (VNS) is a systematic change of neighborhood both within a descent phase to find a local optimum and in a perturbation phase to get out of the corresponding valley [16, Ch. 3].

Suppose we have $\{\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_k\}$, a finite set of pre-selected k neighborhood structures.

Hence, every time a local optimum is reached, the algorithm changes the neighborhood structure and searches a local optimum belonging to the new neighborhood. Once it is reached, VNS starts again from the beginning, with the first neighborhood structure.

Figure 9 shows us a VNS procedure with three different neighborhood structures. The algorithm starts with s_1 and explores the first neighborhood

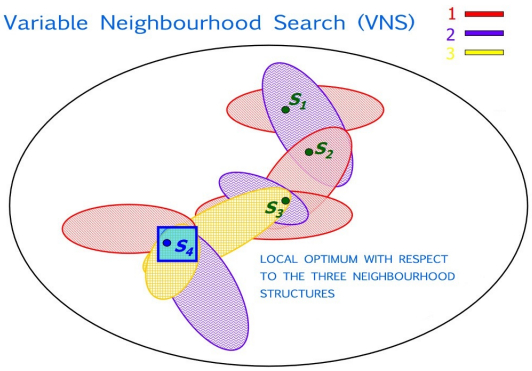


Figure 9: VNS procedure

(in red). If s_1 is a local optimum w.r.t. $\mathcal{N}_1$, then it explores $\mathcal{N}_2(s_1)$ and finds s_2. Now, it starts again looking for a local optimum w.r.t. $\mathcal{N}_1(s_2)$ and so on. The algorithm stops as soon as he finds a local optimum w.r.t. all the three neighborhood structures.

VNS algorithm is based on three simple facts [16]:

FACT 1 A local minimum with respect to one neighborhood structure is not necessarily so for another.

FACT 2 A global minimum is a local minimum with respect to all possible neighborhood structures.

FACT 3 For many problems, local minima with respect to one or several $\mathcal{N}_j$ are relatively close to each other.

There are three different ways to use Facts 1-3:

1. Deterministic;
2. Stochastic.
3. Both deterministic and stochastic.

In Section 4.3.3 we will focus on *Variable neighborhood Descent* (VND) , an algorithm which belongs to the first approach.

In Section 4.3.4 we will study the *General Variable Neighborhood Search* (GVNS), an example of the third approach.

4.3.3 *Variable Neighborhood Descent*

The Variable neighborhood Descent algorithm (VND) performs a change of neighborhoods in a deterministic way. As the name says, it is a descent method, hence it could be implemented as an intensification phase of more sophisticated algorithm.

Pseudo code 10 describes a general VND method.

Description of the algorithm

As usual, let us suppose we have a set of prefixed neighborhood structures $\{\mathcal{N}_k \mid k = 1, \ldots, k_{\max}\}$ and an initial permutation s.

The algorithm starts doing a local search with respect to the first neighborhood structure $\mathcal{N}_1$. A local optimum s' w.r.t. $\mathcal{N}_1$ is found (line 3).

Now there are two cases:

Algorithm 10: VND algorithm

Input: s, $\{\mathcal{N}_1, \ldots, \mathcal{N}_{k_{\max}}\}$;
Output: s^*, a local optimum w.r.t. all neighborhood structures

```
1  k ← 1;
2  while k ≤ k_max do
3  |   Do a local search in N_k(s); a local optimum s̃ w.r.t. N_k is found;
4  |   if z(s̃) < z(s) then
5  |   |   s ← s̃;
6  |   |   k ← 1;
7  |   else
8  |   |   k ← k + 1;
9  |   end
10 end
11 s* ← s;
```

1. If $z(s') < z(s)$ then s' is a *better* permutation than s, therefore the algorithms starts again exploring $\mathcal{N}_1(s')$ (lines 5-6).
2. If $z(s') = z(s)$, then s' is a local optimum with respect to $\mathcal{N}(s)$, the algorithm explores the next neighborhood structure $\mathcal{N}_2(s')$ (line 8).

This procedure is repeated until a local optimum w.r.t. all neighborhood structures is found (lines 2-10). Note that $z(s') > z(s)$ cannot occur, since s' is a local optimum with respect to $\mathcal{N}_k(s)$.

The final solution s is a local optimum with respect to all neighborhood structures (line 11).

Implementation

We implemented VND in the most immediate way. We used the neighborhood structures defined in 3.2, in particular we used N_2 and N_3.

We know that there are two strategies for a local search on these neighborhood: first-improvement and best-improvement. Therefore, we implemented two algorithms:

- VNDfirst, that uses a first-improvement strategy.
- VNDbest, that uses a best-improvement strategy.

Pseudocode 11 shows the VNDfirst algorithm. The best-improvement version is totally similar, except for lines 6 and 9, where algorithms `2optBest` and `3optBest` are called.

Note that VND algorithm only achieves intensification, therefore we still lack some procedure for diversification phase. Here it will come the GVNS algorithm.

4.3.4 *GVNS*

A more general approach is the General Variable neighborhood Search algorithm (GNVS).

Description of the algorithm

GVNS introduces a new set of neighborhood structures $\{\mathcal{P}_1, \ldots, \mathcal{P}_{h_{\max}}\}$. Hence, we have two sets of neighborhood structures:

Algorithm 11: VNDfirst algorithm

Input: n, $\boldsymbol{F}$, $\boldsymbol{D}$, π_{start}, z_{start};
Output: π_{best}, z_{best};
1 $\pi_{\text{best}} \leftarrow \pi_{\text{start}}$;
2 $z_{\text{best}} \leftarrow z_{\text{start}}$;
3 $k \leftarrow 1$;
4 **while** $k \leq 2$ **do**
5 **if** $k = 1$ **then**
6 Call `2optFirst`$(p_{\text{best}}, z_{\text{best}}, \pi, z_\pi)$
7 **end**
8 **else if** $k = 2$ **then**
9 Call `3optFirst`$(p_{\text{best}}, z_{\text{best}}, \pi, z_\pi)$
10 **end**
11 **if** $z_\pi < z_{\text{best}}$ **then**
12 $z_{\text{best}} \leftarrow z_\pi$;
13 $\pi_{\text{best}} \leftarrow \pi$;
14 $k \leftarrow 1$;
15 **else**
16 $k \leftarrow k + 1$;
17 **end**
18 **end**
19 Stop: π_{best} is 2-optimal and 3-optimal;

- The set $\{\mathcal{N}_1, \ldots, \mathcal{N}_{k_{\max}}\}$ which will be used in VND algorithm (intensification phase).
- The set $\{\mathcal{P}_1, \ldots, \mathcal{P}_{h_{\max}}\}$ which will be used to move randomly (diversification phase).

In general, these two sets can be totally different even in the number of elements ($k_{\max} \neq h_{\max}$).

Pseudo code 12 conveys an idea of the algorithm.

First, the algorithm builds an initial point s (line 1).

Then, the main loop starts (line 2). At first, a diversification is applied: the initial permutation s is moved in a random way (we will discuss this step in the next section) into another permutation s' belonging to $\mathcal{P}_1(s)$ (line 4).

After that, in order to achieve intensification, VND algorithm is applied to s' and provides a permutation s'' which is local optimum w.r.t. $\mathcal{N}_1, \ldots, \mathcal{N}_{k_{\max}}$ (line 5).

Now there are two cases:

1. If $z(s'') < z(s)$, then the algorithm starts again with the diversification phase updating permutation s as s'' (line 7).
2. If $z(s'') \geq z(s)$, then the neighborhood $\mathcal{P}_2$ is considered. A perturbation is applied and the permutation s'' is randomly moved into another permutation belonging to $\mathcal{P}_2(s'')$ (lines 10-12).

This procedure is repeated until $h > h_{\max}$ (line 16). In this case, s^* is a local optimum w.r.t. neighborhood structures $\mathcal{N}_1, \ldots, \mathcal{N}_{k_{\max}}$.

In general, s^* is not an optimum for the other neighborhood structures $\mathcal{P}_h$, since their role is just to provide diversification perturbing solution.

The stopping criterion we chose is the execution time $t_{\max}$ and a maximum number of 10 000 loops without any improvement.

Algorithm 12: GVNS pseudo code

Input: a set of neighborhood structures $\{\mathcal{N}_1, \ldots, \mathcal{N}_{k_{\max}}\}$ for VND algorithm, a set of neighborhood structures $\{\mathcal{P}_1, \ldots, \mathcal{P}_{h_{\max}}\}$ for diversification phase

Output: s^* local optimum w.r.t. $\mathcal{N}_1, \ldots \mathcal{N}_{k_{\max}}$

1 Construct an initial solution s;
2 **while** *stopping criterion* **do**
3 $h \leftarrow 1$;
`/* diversification */`
4 Perturbation: choose at random $s' \in \mathcal{P}_h(s)$;
`/* intensification */`
5 Realize a VND starting from s', obtaining s'';
6 **if** $z(s'') < z(s)$ **then**
7 $s \leftarrow s''$;
8 Go to step 3;
9 **end**
10 **if** $h \leq h_{\max}$ **then**
11 $h \leftarrow h + 1$;
12 Go to step 4;
13 **else**
14 Exit
15 **end**
16 **end**
17 $s^* \leftarrow s$;
18 s^* is the minimum w.r.t. neighborhood structures $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{k_{\max}}$.

Figure 10 shows GVNS procedure with three neighborhood structures.

The algorithm starts with an initial permutation s_1 (obtained after a diversification phase, which is omitted in the figure for sake of clarity).

Then, a VND is applied and a new permutation s_2 is obtained. Then, the diversification phase starts again.

This procedure repeats until the stopping criterion.

Implementation

For the intensification phase we used VNDfirst and VNDbest algorithms, with neighborhood structures $\{N_2, N_3\}$. Since there are two VND procedures, we implemented two GVNS algorithms: GVNSfirst and GVNSbest.

As regards the GVNS algorithms, we added one extra neighborhood structure: $\{N_2, N_3, \mathcal{P}\}$, where $\mathcal{P}(\pi)$ is the neighborhood (a singleton) formed by dividing the permutation π in two and swapping the first part with the second one.

For example,

$$\text{If } \pi = [\,1{,}4\,,\,3{,}2\,], \text{ then } \mathcal{P}(\pi) = [\,3{,}2\,,\,1{,}4\,]$$

Thus, in our case, $k_{\max} = 2$ and $h_{\max} = 3$.

Input and output are the same on every algorithm:

INPUT The initial solution p and z_p is its objective function value.

OUTPUT The best found solution p_{best} and z_{best} its objective function value.

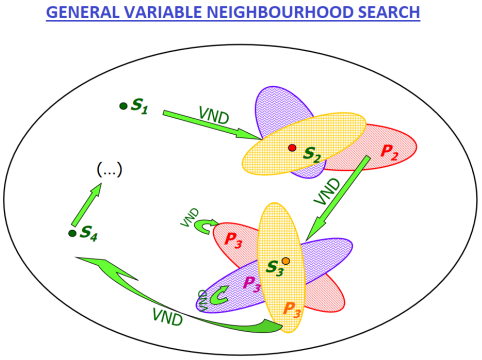


Figure 10: GVNS procedure.

As we can see, there are no parameters to calibrate.

Pseudo code 13 explains GVNDfirst algorithm. The best-improvement version is the same, exchanging only line 17.

The stopping criterion we chose is the execution time $t_{\max}$ and a maximum number of 10 000 loops without any improvement.

Algorithm 13: GVNSfirst

```
   Input: n, F, D, p, z_p
   Output: p_best, z_best
   /* initialization                                          */
1  p_best ← p;
2  z_best ← z;
   /* main loop                                               */
3  repeat
4      j ← 1 ;
5      while j ≤ 3 do
           /* diversification                                 */
6          if j > 3 then exit;
7          else if j = 1 then
8              Do a random 2-exchange {i1, i2} on p;
9              z_p ← z_p + Δ(p; i1, i2)
10         else if j = 2 then
11             Do a random 3-exchange {i1, i2, i3} on p;
12             z_p ← z_p + Δ^1(p; i1, i2, i3)
13         else if j = 3 then
14             Swap the first half of p with the second half, call it again p;
15             Evaluate the objective function z_p of p;
16         end
           /* intensification                                 */
17         Call VNDfirst algorithm;
18         if z(p) < z(p_best) then
19             p_best ← p;
20             z_best ← z;
21             Go to 4
22         end
23         j ← j + 1 ;
24     end
25 until stopping criterion;
```

5 COMPUTATIONAL RESULTS

5.1 QAPLIB LIBRARY

QAPLIB [6] is an online library of instances, solutions and lower bounds. The QAPLIB home page was created by Stefan Karisch in 1991.

According to [34], QAPLIB's instances can be classified into four groups:

UNSTRUCTURED, RANDOMLY GENERATED INSTANCES Instances with the distance and flow matrix entries generated randomly according to an uniform distribution.

These instances are among the hardest to solve exactly. Nevertheless, most iterative search methods find solutions within 1%/2% from the best known solutions relatively fast.

UNSTRUCTURED INSTANCES WITH GRID-DISTANCES Instances with the distance matrix $\boldsymbol{D}$ defined as the Manhattan distance between grid points on a $n_1 \times n_2$ 2 grid and with random flows.

REAL LIFE INSTANCES These instances are discussed later.

REAL-LIFE LIKE INSTANCES These instances are generated in such a way that the matrix entries resemble the distribution found in real-life problems

Real life instances

These instances arise from practical applications. There are five group of real life instances. In chronological order they are:

- *Steinberg's* [32] (1961). This instance is described in section 2.3. There are three instances, the first one has a distance matrix corresponding to Manhattan distances, the second the square of Euclidean distances and the third Euclidean distances. These instances are denoted `Ste36a`, `Ste36b`, `Ste36c`. Their size is $n = 9 \times 4 = 36$ and they are asymmetric. They are solved optimally.
- *Elshafei's* (1977). This problem is described in section 2.1. The size of the problem is $n = 19$. It is denoted `Els19` and it is solved optimally.
- *Burkard and Offermann* (1977). This instance is described in section 2.4. The size is $n = 26$. Since four different languages and two variety of typewriters are considered, there are eight problems of this type. Since there are 26 'Latin' letters, the size of the problems is $n = 26$. They are denoted `Bur26a`,..., `Bur26h` and they are solved optimally. These instances are asymmetric.
- *J. Krarup and P.M. Pruzan*(1981). This instance is similar to Elshafei's one. There are three instances, denoted by `Kra30a`, `Kra30b` and `Kra32`.

The real-life instances have in common that the flow matrices have (in contrast to the previously mentioned randomly generated instances) many zero entries and the remaining entries are clearly not uniformly distributed.

5.2 NEOS

Other instances can be found on NEOS webpage on https://neos-guide.org/content/quadratic-assignment-problem.

They have small size : $n = 4, 5, 6, 7, 8, 9$. Thus, we will denote them as `Neos4`, . . . , `Neos9`. Every instance is symmetric.

5.3 COMPARISON OF ALGORITHMS

In table 17 metaheuristic algorithms are compared. As we can see, Ant Colony Optimization algorithm performed fairly well, especially with instances of low dimension. In particular, he provided the optimal solution for every `bur26x` instances and most of the Nugent.

On the other hand, GVNSfirst sometimes performed better than ACO with instances of higher dimension, e.g., `sko42`,`sko72`, `sko100b-f` and `wil50`). Note that GVNSfirst provided the optimal solution of `sko42` and `sko72`. Nevertheless, he failed in `sko56` and `sko64`, with a 6% PD.

GVNSbest in general performed worse than GVNSfirst with some exceptions (`sko49`, the already cited `sko56`, `sko64`, `nug22`,`nug25`).

As regards Tabu Search, it did not perform well in general. The minimum of PD provided is 0.54% for `bur26h`, while the other algorithms reached the optimal solution. Often it provide a permutation with more than 5% of PD.

Table 17: Comparison of metaheuristic algorithms.The results are the minimum over 5 independent runs. For each instance we set $t_{\max} = 60\,\mathrm{s}$. Instances for which the optimal value is not known are marked with "*".

Instance	TS	ACO	GVNSfirst	GVNSbest
bur26a	0.97	0.00	0.00	0.11
bur26b	1.15	0.00	0.17	0.17
bur26c	1.25	0.00	0.00	0.00
bur26d	1.62	0.00	0.00	0.00
bur26e	1.40	0.00	0.00	0.00
bur26f	1.94	0.00	0.00	0.02
bur26g	1.19	0.00	0.00	0.00
bur26h	0.54	0.00	0.00	0.00
nug12	9.69	0.00	0.00	1.38
nug14	7.50	0.00	0.39	1.97
nug15	9.22	0.00	0.00	0.87
nug16a	9.19	0.00	0.00	1.49
nug16b	13.71	0.00	0.00	0.00
nug17	9.01	0.00	0.00	0.00
nug18	12.12	0.00	0.00	0.73
nug20	11.67	0.00	0.70	1.09
nug21	12.06	0.00	0.16	0.33
nug22	10.85	0.00	0.50	0.00
nug24	13.19	0.00	0.00	0.69
nug25	8.12	0.00	0.16	0.11
nug27	6.38	0.00	0.00	0.84
nug28	10.61	0.12	0.00	0.66
nug30	9.96	0.07	0.00	1.57
sko42*	7.94	0.11	0.00	0.04
sko49*	6.78	0.42	1.00	0.03
sko56*	5.60	0.38	5.00	0.49
sko64*	5.03	0.39	5.00	0.24
sko72*	4.76	0.44	0.00	0.27
sko81*	4.42	0.99	1.33	1.41
sko90*	4.43	1.06	0.98	1.34
sko100a*	3.99	0.67	0.79	1.00
sko100b*	3.89	1.38	0.67	1.24
sko100c*	4.00	1.06	0.58	1.22
sko100d*	4.05	1.31	0.54	1.52
sko100e*	3.85	1.38	1.44	1.48
sko100f*	4.42	1.03	1.06	1.87
tho30	12.86	0.00	0.29	2.47
tho40*	11.89	0.20	0.05	0.93
tho150*	4.58	0.76	1.38	1.45
wil50*	2.65	0.15	0.03	0.43
wil100*	1.93	0.31	0.37	0.93

6 CONCLUSIONS AND FUTURE WORKS.

6.1 FUTURE WORKS

We can list some possible future developments:

EVALUATION OF 2, 3-EXCHANGES As regards local search algorithms, we can observe that the 3-exchange $\{i_1, i_2, i_3\} \to \{i_2, i_3, i_1\}$ can be obtained by performing two 2-exchanges: $\{i_1, i_2, i_3\} \to \{i_2, i_1, i_3\} \to \{i_2, i_3, i_1\}$.

The same thing occurs with the 3-exchange $\{i_1, i_2, i_3\} \to \{i_3, i_1, i_2\}$, in fact $\{i_1, i_2, i_3\} \to \{i_2, i_1, i_3\} \to \{i_3, i_1, i_2\}$.

One can observe that both contains the 2-exchange $\{i_1, i_2, i_3\} \to \{i_2, i_1, i_3\}$. This allows us to reduce the number of operations needed to evaluate $\Delta^1(\pi; i_1, i_2, i_3)$ and $\Delta^2(\pi; i_1, i_2, i_3)$.

A 3-optimum algorithm implemented with this technique will provide 2-optimal and 3-optimal solutions with a computational cost of $O(n)$ operations.

IMPROVEMENT OF TABU SEARCH ALGORITHM As we saw in Chapter 5, Tabu Search algorithms did not perform well. There can be many reasons for that. One possible solution is to modify the algorithm, adding more parameters, which are typically used in modern TS. An other possibility is to use an adaptive algorithm to for the auto-calibration of parameters (like Particle Swarm Optimization, PSO).

NEW ALGORITHMS Of course there are lots of possible new algorithms to implement. One can cite Simulated Annealing, Genetic Algorithm, Memetic algorithm, GRASP, Bee Colony Optimization, and so on and so forth. Another approach could be to make an hybrid method mixing, for example, Tabu Search with Ant Colony Optimization.

6.2 CONCLUDING REMARKS

We presented the QAP, an optimization problem that challenged hundreds of researches since 1957.

The goal of this thesis was to study, implement, compare heuristic and metaheuristic algorithms. Every aspect of the goal has been accomplished and has been an educative experience.

A

PROOF OF THEOREM 3.1

Theorem. *Let $\pi \in S_n$ be a permutation and $i_1 \neq i_2$ be two indices. Then,*

$$z(\pi_{i_1 i_2}) = z(\pi) + \Delta(\pi; i_1, i_2). \tag{40}$$

Proof.
Applying definition (2) it follows that

$$\begin{aligned} z(\pi_{i_1 i_2}) - z(\pi) &= \sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} d_{\pi_{i_1 i_2}(i)\pi_{i_1 i_2}(j)} - \sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} d_{\pi(i)\pi(j)} \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij}\left[d_{\pi_{i_1 i_2}(i)\pi_{i_1 i_2}(j)} - d_{\pi(i)\pi(j)}\right]. \end{aligned} \tag{41}$$

Let $K = \{i_1, i_2\}$. Note that the sums can be split into different ones [1]:

$$\begin{aligned} \sum_{i=1}^{n}\sum_{j=1}^{n} &= \sum_{j\notin K}\sum_{i=1}^{n} + \sum_{j\in K}\sum_{i=1}^{n} \\ &= \underbrace{\sum_{j\notin K}\sum_{i\notin K}}_{\text{♙}} + \underbrace{\sum_{j\notin K}\sum_{i\in K}}_{\text{♘}} + \underbrace{\sum_{i=1}^{n}\sum_{j\in K}}_{\text{♗}}. \end{aligned}$$

The first sum ♙ is null, since $\pi(i) = \pi_{i_1 i_2}(i)$ for every $i \notin K$ and therefore every term is 0.

As regards ♘, we obtain

$$\begin{aligned} \text{♘} &= \sum_{j\notin K}\sum_{i\in K} f_{ij}\left[d_{\pi_{i_1 i_2}(i)\pi_{i_1 i_2}(j)} - d_{\pi(i)\pi(j)}\right] \\ &= \sum_{j\notin K}\left\{f_{i_1 j}\left[d_{\pi(i_2)\pi(j)} - d_{\pi(i_1)\pi(j)}\right] + f_{i_2 j}\left[d_{\pi(i_1)\pi(j)} - d_{\pi(i_2)\pi(j)}\right]\right\} \\ &= \sum_{j\notin K}\left\{\left(f_{i_1 j} - f_{i_2 j}\right) d_{\pi(i_2)\pi(j)} + \left(f_{i_2 j} - f_{i_1 j}\right) d_{\pi(i_1)\pi(j)}\right\} \\ &= \sum_{j\notin K}\left(f_{i_1 j} - f_{i_2 j}\right)\left(d_{\pi(i_2)\pi(j)} - d_{\pi(i_1)\pi(j)}\right). \end{aligned} \tag{42}$$

Doing same calculus on ♗, we get

$$\begin{aligned} \text{♗} &= \sum_{i=1}^{n}\sum_{j\in K} f_{ij}\left[d_{\pi_{i_1 i_2}(i)\pi_{i_1 i_2}(j)} - d_{\pi(i)\pi(j)}\right] \\ &= \sum_{i=1}^{n}\left\{f_{ii_1}\left[d_{\pi_{i_1 i_2}(i)\pi(i_2)} - d_{\pi(i)\pi(i_1)}\right] + f_{ii_2}\left[d_{\pi_{i_1 i_2}(i)\pi(i_1)} - d_{\pi(i)\pi(i_2)}\right]\right\} \\ &= \sum_{j=1}^{n}\left\{f_{ji_1}\left[d_{\pi_{i_1 i_2}(j)\pi(i_2)} - d_{\pi(j)\pi(i_1)}\right] + f_{ji_2}\left[d_{\pi_{i_1 i_2}(j)\pi(i_1)} - d_{\pi(j)\pi(i_2)}\right]\right\}. \end{aligned} \tag{43}$$

[1] Terms of the sums are omitted for sake of clarity.

Where in the last step we changed the name of the variable from i to j. Then, the sum can be split again :

$$\begin{aligned} \text{🯅} &= \sum_{j\in K}\left\{f_{ji_1}\left[d_{\pi_{i_1i_2}(j)\pi(i_2)}-d_{\pi(j)\pi(i_1)}\right]+f_{ji_2}\left[d_{\pi_{i_1i_2}(j)\pi(i_1)}-d_{\pi(j)\pi(i_2)}\right]\right\} \\ &+\sum_{j\notin K}\left\{f_{ji_1}\left[d_{\pi_{i_1i_2}(j)\pi(i_2)}-d_{\pi(j)\pi(i_1)}\right]+f_{ji_2}\left[d_{\pi_{i_1i_2}(j)\pi(i_1)}-d_{\pi(j)\pi(i_2)}\right]\right\}. \end{aligned} \tag{44}$$

The first sum can be written explicitly:

$$\begin{aligned} &\sum_{j\in K}\left\{f_{ji_1}\left[d_{\pi_{i_1i_2}(j)\pi(i_2)}-d_{\pi(j)\pi(i_1)}\right]+f_{ji_2}\left[d_{\pi_{i_1i_2}(j)\pi(i_1)}-d_{\pi(j)\pi(i_2)}\right]\right\} \\ &=f_{i_1i_1}\left[d_{\pi(i_2)\pi(i_2)}-b_{\pi(i_1)\pi(i_1)}\right]+f_{i_1i_2}\left[d_{\pi(i_2)\pi(i_1)}-d_{\pi(i_1)\pi(i_2)}\right] \\ &+f_{i_2i_1}\left[d_{\pi(i_1)\pi(i_2)}-d_{\pi(i_2)\pi(i_1)}\right]+f_{i_2i_2}\left[d_{\pi(i_1)\pi(i_1)}-d_{\pi(i_2)\pi(i_2)}\right] \\ &=\left(f_{i_1i_1}-f_{i_2i_2}\right)\left(d_{\pi(i_2)\pi(i_2)}-d_{\pi(i_2)\pi(i_1)}\right) \\ &+\left(f_{i_1i_2}-f_{i_2i_1}\right)\left(d_{\pi(i_2)\pi(i_1)}-d_{\pi(i_1)\pi(i_2)}\right). \end{aligned} \tag{45}$$

Finally, putting together (42), (43) and (45), we get

$$\begin{aligned} z(\pi_{i_1i_2})-z(\pi) &= \sum_{j\notin K}\left(f_{i_1j}-f_{i_2j}\right)\left(d_{\pi(i_2)\pi(j)}-d_{\pi(i_1)\pi(j)}\right) \\ &+\sum_{j\notin K}\left(f_{ji_1}-f_{ji_2}\right)\left(d_{\pi(j)\pi(i_2)}-d_{\pi(j)\pi(i_1)}\right) \\ &=\left(f_{i_1i_1}-f_{i_2i_2}\right)\left(d_{\pi(i_2)\pi(i_2)}-d_{\pi(i_2)\pi(i_1)}\right) \\ &+\left(f_{i_1i_2}-f_{i_2i_1}\right)\left(d_{\pi(i_2)\pi(i_1)}-d_{\pi(i_1)\pi(i_2)}\right). \end{aligned}$$

□

B

PROOF OF THEOREM 3.2

Theorem. *Let $\pi \in S_n$ be a permutation and $i_1, i_2, i_3 \in [n]$ three distinct indices. Then one has*

- $z\left(\pi^1_{i_1 i_2 i_3}\right) = z(\pi) + \Delta^1(\pi; i_1, i_2, i_3)$.
- $z\left(\pi^2_{i_1 i_2 i_3}\right) = z(\pi) + \Delta^2(\pi; i_1, i_2, i_3)$.

Proof.
It is sufficient to evaluate the differences $z\left(\pi^1_{i_1 i_2 i_3}\right) - z(\pi)$ and $z\left(\pi^2_{i_1 i_2 i_3}\right) - z(\pi)$.

We will do the first evaluation, the second one is, *mutatis mutandis*, totally similar. For sake of clarity we will denote $\pi^1_{i_1 i_2 i_3}$ as $\tilde{\pi}$. Taking inspiration from theorem 3.1,

$$\begin{aligned} z(\tilde{\pi}) - z(\pi) &= \sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} d_{\tilde{\pi}(i)\tilde{\pi}(j)} - \sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} d_{\pi(i)\pi(j)} \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij}\left[d_{\tilde{\pi}(i)\tilde{\pi}(j)} - d_{\pi(i)\pi(j)}\right]. \end{aligned} \tag{46}$$

Let $K = \{i_1, i_2, i_3\}$. The sum can be split into two parts:

$$\underbrace{\sum_{j\notin K}\sum_{i=1}^{n} f_{ij}\left[d_{\tilde{\pi}(i)\tilde{\pi}(j)} - d_{\pi(i)\pi(j)}\right]}_{\text{🛕}} + \underbrace{\sum_{j\in K}\sum_{i=1}^{n} f_{ij}\left[d_{\tilde{\pi}(i)\tilde{\pi}(j)} - d_{\pi(i)\pi(j)}\right]}_{\text{🦢}}.$$

As regards 🛕, we can note that those i such that $i \notin K$ produce a null term, hence the sum $\sum_{i\notin K}\sum_{j\notin K}$ is equal to 0. Thus,

$$\begin{aligned} &\sum_{j\notin K}\sum_{i\in K} f_{ij}\left[d_{\tilde{\pi}(i)\tilde{\pi}(j)} - d_{\pi(i)\pi(j)}\right] \\ =&\sum_{j\notin K}\Big\{f_{i_1 j}\left[d_{\pi(i_2)\pi(j)} - d_{\pi(i_1)\pi(j)}\right] + f_{i_2 j}\left[d_{\pi(i_3)\pi(j)} - d_{\pi(i_2)\pi(j)}\right] \\ &\quad + f_{i_3 j}\left[d_{\pi(i_1)\pi(j)} - d_{\pi(i_3)\pi(j)}\right]\Big\}. \end{aligned} \tag{47}$$

For 🦢, with the same idea we get

$$\begin{aligned} &\sum_{j\in K}\sum_{i=1}^{n} f_{ij}\left[d_{\tilde{\pi}(i)\tilde{\pi}(j)} - d_{\pi(i)\pi(j)}\right] \\ =&\sum_{i=1}^{n}\Big\{f_{i i_1}\left[d_{\tilde{\pi}(i)\pi(i_2)} - d_{\pi(i)\pi(i_1)}\right] + f_{i i_2}\left[d_{\tilde{\pi}(i)\pi(i_3)} - d_{\pi(i)\pi(i_2)}\right] \\ &\quad + f_{i i_3}\left[d_{\tilde{\pi}(i)\pi(i_3)} - d_{\pi(i)\pi(i_3)}\right]\Big\} \\ \overset{i\to j}{=}&\sum_{j=1}^{n}\Big\{f_{j i_1}\left[d_{\tilde{\pi}(j)\pi(i_2)} - d_{\pi(j)\pi(i_1)}\right] + f_{j i_2}\left[d_{\tilde{\pi}(j)\pi(i_3)} - d_{\pi(j)\pi(i_2)}\right] \end{aligned}$$

$$
\begin{aligned}
&\quad + f_{ji_3}\left[d_{\tilde{\pi}(j)\pi(i_3)} - d_{\pi(j)\pi(i_3)}\right]\Big\} \\
&= \sum_{j\notin K}^{n}\Big\{f_{ji_1}\left[d_{\pi(j)\pi(i_2)} - d_{\pi(j)\pi(i_1)}\right] + f_{ji_2}\left[d_{\pi(j)\pi(i_3)} - d_{\pi(j)\pi(i_2)}\right] \\
&\quad + f_{ji_3}\left[d_{\pi(j)\pi(i_3)} - d_{\pi(j)\pi(i_3)}\right]\Big\} \\
&\quad + f_{i_1i_1}\left[d_{\pi(i_2)\pi(i_2)} - d_{\pi(i_1)\pi(i_1)}\right] + f_{i_1i_2}\left[d_{\pi(i_2)\pi(i_3)} - d_{\pi(i_1)\pi(i_2)}\right] \\
&\quad + f_{i_1i_3}\left[d_{\pi(i_2)\pi(i_1)} - d_{\pi(i_1)\pi(i_3)}\right] + f_{i_2i_1}\left[d_{\pi(i_3)\pi(i_2)} - d_{\pi(i_2)\pi(i_1)}\right] \\
&\quad + f_{i_2i_2}\left[d_{\pi(i_3)\pi(i_3)} - d_{\pi(i_2)\pi(i_2)}\right] + f_{i_2i_3}\left[d_{\pi(i_2)\pi(i_1)} - d_{\pi(i_2)\pi(i_3)}\right] \\
&\quad + f_{i_3i_1}\left[d_{\pi(i_1)\pi(i_2)} - d_{\pi(i_3)\pi(i_1)}\right] + f_{i_3i_2}\left[d_{\pi(i_1)\pi(i_3)} - d_{\pi(i_3)\pi(i_2)}\right] \\
&\quad + f_{i_3i_3}\left[d_{\pi(i_1)\pi(i_1)} - d_{\pi(i_3)\pi(i_3)}\right].
\end{aligned}
$$

Putting (46) and (47) together, we obtain

$$
\begin{aligned}
z(\tilde{\pi}) - z(\pi) = \sum_{j\notin K}^{n}\Big\{&f_{ji_1}\left[d_{\pi(j)\pi(i_2)} - d_{\pi(j)\pi(i_1)}\right] + f_{ji_2}\left[d_{\pi(j)\pi(i_3)} - d_{\pi(j)\pi(i_2)}\right] \\
&+ f_{ji_3}\left[d_{\pi(j)\pi(i_3)} - d_{\pi(j)\pi(i_3)}\right] + f_{i_1j}\left[d_{\pi(i_2)\pi(j)} - d_{\pi(i_1)\pi(j)}\right] \\
&+ f_{i_2j}\left[d_{\pi(i_3)\pi(j)} - d_{\pi(i_2)\pi(j)}\right] + f_{i_3j}\left[d_{\pi(i_1)\pi(j)} - d_{\pi(i_3)\pi(j)}\right]\Big\} \\
&+ f_{i_1i_1}\left[d_{\pi(i_2)\pi(i_2)} - d_{\pi(i_1)\pi(i_1)}\right] + f_{i_1i_2}\left[d_{\pi(i_2)\pi(i_3)} - d_{\pi(i_1)\pi(i_2)}\right] \\
&+ f_{i_1i_3}\left[d_{\pi(i_2)\pi(i_1)} - d_{\pi(i_1)\pi(i_3)}\right] + f_{i_2i_1}\left[d_{\pi(i_3)\pi(i_2)} - d_{\pi(i_2)\pi(i_1)}\right] \\
&+ f_{i_2i_2}\left[d_{\pi(i_3)\pi(i_3)} - d_{\pi(i_2)\pi(i_2)}\right] + f_{i_2i_3}\left[d_{\pi(i_3)\pi(i_1)} - d_{\pi(i_2)\pi(i_3)}\right] \\
&+ f_{i_3i_1}\left[d_{\pi(i_1)\pi(i_2)} - d_{\pi(i_3)\pi(i_1)}\right] + f_{i_3i_2}\left[d_{\pi(i_1)\pi(i_3)} - d_{\pi(i_3)\pi(i_2)}\right] \\
&+ f_{i_3i_3}\left[d_{\pi(i_1)\pi(i_1)} - d_{\pi(i_3)\pi(i_3)}\right],
\end{aligned}
$$

which is equal to $\Delta^1(\pi, i_1, i_2, i_3)$ on formula (32). □

BIBLIOGRAPHY

[1] Roberto Battiti and Giampietro Tecchiolli. "The continuous reactive tabu search: Blending combinatorial optimization and stochastic search for global optimization". In: *Annals of Operations Research* 63.2 (Apr. 1996), pp. 151–188. DOI: https://doi.org/10.1007/BF02125453 (cit. on p. 39).

[2] Jan Bos. "Zoning in Forest Management: a Quadratic Assignment Problem Solved by Simulated Annealing". In: *Journal of Environmental Management* 37.2 (Feb. 1993), pp. 127–145. DOI: 10.1006/jema.1993.1010 (cit. on p. 8).

[3] M.J. Brusco and S. Stahl. "Using Quadratic Assignment Methods to Generate Initial Permutations for Least-Squares Unidimensional Scaling of Symmetric Proximity Matrices". In: *Journal of Classification* 17.2 (July 2000), pp. 197–223. DOI: 10.1007/s003570000019 (cit. on p. 8).

[4] R. E. Burkard and J. Offermann. "Entwurf von Schreibmaschinentastaturen mittels quadratischer Zuordnungsprobleme". In: *Zeitschrift für Operations Research* 21.4 (Aug. 1977), B121–B132. DOI: 10.1007/bf01918175 (cit. on pp. 8, 10).

[5] Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems*. Society for Industrial and Applied Mathematics, Jan. 2012. DOI: 10.1137/1.9781611972238 (cit. on pp. viii, 2, 6, 8, 40).

[6] Rainer E. Burkard, Stefan E. Karisch, and Franz Rendl. "QAPLIB – A Quadratic Assignment Problem Library". In: *Journal of Global Optimization* 10.4 (1997), pp. 391–403. DOI: https://doi.org/10.1023/A:1008293323270 (cit. on pp. 29, 57).

[7] Sergio A de Carvalho Jr and Sven Rahmann. "Microarray layout as quadratic assignment problem". In: *German Conference on Bioinformatics*. Gesellschaft für Informatik eV. 2006 (cit. on p. 8).

[8] Eranda Çela. *The Quadratic Assignment Problem*. Springer US, 1998. DOI: 10.1007/978-1-4757-2787-6 (cit. on p. 8).

[9] G. A. Croes. "A Method for Solving Traveling-Salesman Problems". In: *Operations Research* 6.6 (Dec. 1958), pp. 791–812. DOI: 10.1287/opre.6.6.791 (cit. on p. 21).

[10] Mauro Dell'Amico et al. "The single-finger keyboard layout problem". In: *Computers & Operations Research* 36.11 (Nov. 2009), pp. 3002–3012. DOI: 10.1016/j.cor.2009.01.018 (cit. on p. 10).

[11] H. A. Eiselt and Gilbert Laporte. "A Combinatorial Optimization Problem Arising in Dartboard Design". In: *Journal of the Operational Research Society* 42.2 (Feb. 1991), pp. 113–118. DOI: 10.1057/jors.1991.21 (cit. on pp. 11, 12).

[12] Alwalid N. Elshafei. "Hospital Layout as a Quadratic Assignment Problem". In: *Operational Research Quarterly (1970-1977)* 28.1 (1977), p. 167. DOI: 10.2307/3008789 (cit. on pp. 8, 9).

[13] Xiongfeng Feng and Qiang Su. "An applied case of quadratic assignment problem in hospital department layout". In: *2015 12th International Conference on Service Systems and Service Management (ICSSSM)*. IEEE, June 2015. DOI: 10.1109/icsssm.2015.7170278 (cit. on p. 9).

[14] Gerd Finke, Rainer E. Burkard, and Franz Rendl. "Quadratic Assignment Problems". In: *Surveys in Combinatorial Optimization*. Elsevier, 1987, pp. 61–82. DOI: 10.1016/s0304-0208(08)73232-8 (cit. on p. 6).

[15] L M Gambardella, D Taillard, and M Dorigo. "Ant colonies for the quadratic assignment problem". In: *Journal of the Operational Research Society* 50.2 (Jan. 1999), pp. 167–176. DOI: 10.1057/palgrave.jors.2600676 (cit. on pp. 39, 46, 48, 49).

[16] M. & Potvin Gendreau. *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. Springer International Publishing, 2019. DOI: 10.1007/978-3-319-91086-4 (cit. on pp. viii, 39, 50–52).

[17] A. M. Geoffrion and G. W. Graves. "Scheduling Parallel Production Lines with Changeover Costs: Practical Application of a Quadratic Assignment/LPApproach". In: *Operations Research* 24.4 (Aug. 1976), pp. 595–610. DOI: 10.1287/opre.24.4.595 (cit. on p. 8).

[18] Fred Glover. "Future paths for integer programming and links to artificial intelligence". In: *Computers & Operations Research* 13.5 (Jan. 1986), pp. 533–549. DOI: 10.1016/0305-0548(86)90048-1 (cit. on pp. 38, 39).

[19] S. Goss et al. "Self-organized shortcuts in the Argentine ant". In: *Naturwissenschaften* 76.12 (Dec. 1989), pp. 579–581. DOI: 10.1007/bf00462870 (cit. on p. 46).

[20] Charles H. Heider. "Ann-step, 2-variable search algorithm for the component placement problem". In: *Naval Research Logistics Quarterly* 20.4 (Dec. 1973), pp. 699–724. DOI: 10.1002/nav.3800200409 (cit. on p. 16).

[21] Stefan Helber et al. "A hierarchical facility layout planning approach for large and complex hospitals". In: *Flexible Services and Manufacturing Journal* 28.1-2 (Feb. 2015), pp. 5–29. DOI: 10.1007/s10696-015-9214-6 (cit. on p. 9).

[22] B. Kim, Jaeik Shim, and Min Zhang. "Comparison of TSP Algorithms Project for Models in Facilities Planning and Materials Handling December 1998". In: 2001. URL: https://pja.mykhi.org/4sem/NAI/rozne/Comparison%20of%20TSP%20Algorithms/Comparison%20of%20TSP%20Algorithms.PDF (cit. on p. 34).

[23] Tjalling C. Koopmans and Martin Beckmann. "Assignment Problems and the Location of Economic Activities". In: *Econometrica* 25.1 (Jan. 1957), p. 53. DOI: 10.2307/1907742 (cit. on p. viii).

[24] Jakob Krarup and Peter Mark Pruzan. "Computer-aided layout design". In: *Mathematical Programming in Use*. Springer Berlin Heidelberg, 1978, pp. 75–94. DOI: 10.1007/bfb0120827 (cit. on p. 9).

[25] Eugene L. Lawler. "The Quadratic Assignment Problem". In: *Management Science* 9.4 (July 1963), pp. 586–599. DOI: 10.1287/mnsc.9.4.586 (cit. on p. 3).

[26] Tommaso Mannelli Mazzoli. *QAP*. GitHub. 2020. URL: https://github.com/Tommaso-Mannelli-Mazzoli/QAP (cit. on pp. 29, 35, 36, 42).

[27] Alfonsas Misevicius. "An implementation of the iterated tabu search algorithm for the quadratic assignment problem". In: *OR Spectrum* 34.3 (Oct. 2011), pp. 665–690. DOI: 10.1007/s00291-011-0274-z (cit. on pp. 39, 40).

[28] N. Mladenovic and P. Hansen. "Variable neighborhood search". In: *Computers and Operations Research* 24.11 (Nov. 1997), pp. 1097–1100. DOI: 10.1016/s0305-0548(97)00031-2 (cit. on p. 50).

[29] Heiner Müller-Merbach. *Optimale Reihenfolgen*. Springer Berlin Heidelberg, 1970. DOI: 10.1007/978-3-642-87727-8 (cit. on p. 9).

[30] Sartaj Sahni and Teofilo Gonzalez. "P-Complete Approximation Problems". In: *Journal of the ACM (JACM)* 23.3 (July 1976), pp. 555–565. DOI: 10.1145/321958.321975 (cit. on pp. viii, 8).

[31] Jadranka Skorin-Kapov. "Tabu Search Applied to the Quadratic Assignment Problem". In: *ORSA Journal on Computing* 2.1 (Feb. 1990), pp. 33–45. DOI: https://doi.org/10.1287/ijoc.2.1.33 (cit. on pp. 38, 39, 42).

[32] Leon Steinberg. "The Backboard Wiring Problem: A Placement Algorithm". In: *SIAM Review* 3.1 (Jan. 1961), pp. 37–50. DOI: 10.1137/1003003 (cit. on pp. 9, 57).

[33] E. Taillard. "Robust taboo search for the quadratic assignment problem". In: *Parallel Computing* 17.4-5 (July 1991), pp. 443–455. DOI: https://doi.org/10.1016/S0167-8191(05)80147-4 (cit. on p. 39).

[34] Éric D. Taillard. "Comparison of iterative searches for the quadratic assignment problem". In: *Location Science* 3.2 (Aug. 1995), pp. 87–105. DOI: 10.1016/0966-8349(95)00008-6 (cit. on pp. 10, 57).

[35] El-Ghazali Talbi. "Metaheuristics". In: (June 2009). DOI: 10.1002/9780470496916 (cit. on p. 38).

[36] Eric W. Weisstein. "Derangement". In: *MathWorld–A Wolfram Web Resource.* (). URL: https://mathworld.wolfram.com/Derangement.html (cit. on p. 15).