CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Cybernetics

Master's Thesis

Metaheuristic Algorithms for Optimization Problems Sharing Representation

Jan Hrazdíra

March 2022
Supervisor: Ing. David Woller

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hrazdíra Jan** Personal ID number: **466181**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Computer Vision and Image Processing**

II. Master’s thesis details

Master’s thesis title in English:

Metaheuristic Algorithms for Optimization Problems Sharing Representation

Master’s thesis title in Czech:

Metaheuristické algoritmy pro optimalizační problémy sdílející reprezentaci

Guidelines:

1. Based on the recommended literature, generalize and implement in C++ a set of suitable local search operators (e.g. 2-opt, swap) and some neighborhood-oriented metaheuristics (e.g. Variable Neighborhood Search, Iterated Local Search), so that they can address optimization problems representable by permutation or variation with replacement.
2. Implement efficient fitness functions and potential constraint violation penalization functions for a set of at least four distinct problems sharing the selected representation (e.g., variants of Vehicle Routing Problem, Quadratic Assignment Problem, Permutation Flowshop Problem, Jobshop Problem).
3. Model the same set of problems for the Gurobi solver as a Mixed Integer Linear Programming problem.
4. Compare the achievable solution quality, speed, and scalability of both approaches (heuristic and exact).

Bibliography / sources:

[1] Rafael Martí, Panos M. Pardalos, Mauricio G. C. Resende - Handbook of Heuristics – Springer, 2018
[2] Michel Gendreau, Jean-Yves Potvin – Handbook of Metaheuristics – Springer, 2019
[3] Laurence Wolsey – Integer Programming – John Wiley & Sons, 2020

Name and workplace of master’s thesis supervisor:

Ing. David Woller Intelligent and Mobile Robotics CIIRC

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **10.01.2022** Deadline for master's thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. David Woller	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

. Date of assignment receipt

Student’s signature

Acknowledgement / Declaration

I would like to thank my supervisor Ing. David Woller, for his invaluable assistance and guidance throughout this project.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 20.5.2022

.....
signature

Abstrakt / Abstract

Univerzální optimalizátory jsou praktické nástroje, které mohou být aplikovány na celou řadu problémů za předpokladu, že se tyto problémy dají modelovat příslušným formalismem. V této práci je navržen heuristický optimalizátor založený na metaheuristikách a lokálním vyhledávání. Lze ho využít na jakýkoliv problém kombinatorické optimalizace, jehož řešení lze zapsat jako seřazenou sekvenci potenciálně opakujících přirozených čísel libovolné délky (např. permutaci nebo variaci s opakováním). Navržený optimalizátor se ukázal být efektivnější než Gurobi Optimizer na problémech *Capacitated Vehicle Routing*, *Quadratic Assignment* a *Non-permutation Flowshop* ve smyslu škálovatelnosti a kvality řešení za stejných podmínek. Naopak v *Sudoku* si lépe počínal Gurobi Optimizer.

Klíčová slova: kombinatorická optimalizace; metaheuristiky; lokální vyhledávání; univerzální optimalizátory;

General-purpose optimizers are convenient tools that can be applied to wide classes of problems, given that these problems can be modeled using a given formalism. This thesis proposes a general-purpose heuristic solver based on neighborhood-oriented metaheuristics. It can be used on any combinatorial problem whose solution can be represented as an ordered sequence of potentially recurring nodes with arbitrary length (e.g. permutation/variation with repetition). The proposed solver proved to outperform the commercial Gurobi Optimizer on *Capacitated Variable Neighborhood*, *Quadratic Assignment*, and *Non-permutation Flowshop* problems in terms of scalability and solution quality given the same computational budget. However, it could not outperform Gurobi in *Sudoku*.

Keywords: combinatorial optimization; metaheuristics; local search; general-purpose optimizer;

Contents /

Algorithms & Tables / Figures

Chapter 1
Introduction

Combinatorial optimization problems (COPs) represent a vast group of problems that play a vital role in many different scientific fields. From obvious applications in scheduling [1] and transportation [2] to less apparent applications in computer vision, including object tracking [3], matching [4], medical imaging [5], or 3D-scene reconstruction [6].

While some of these problems can be solved in polynomial time, many practical COPs belong to the infamous NP-Hard complexity class. This thesis focuses on general-purpose optimization methods that can be applied to large groups of NP-Hard COPs. Many versatile solvers exist, but they are usually much slower than other algorithms explicitly designed for a given problem. For example, considering the famous Traveling Salesman Problem (TSP), recent experimental results show that using the Gurobi IP solver, one can find the optimum for an instance of 229 nodes in approximately three hours [7]. In opposition, there is a software called Concorde explicitly designed for solving TSP, and the authors claim on their website that it can solve the same instance in 38.61 seconds [8]. Of course, these experiments were conducted on non-identical setups, but that does not undermine the fact that the computation times were different by three orders.

Sometimes it is needed to solve much larger instances than exact algorithms can in a reasonable amount of time, so a different approach is required. One option would be to abandon any guarantees on the quality of the solution and use metaheuristics. Metaheuristics are problem-independent strategies that orchestrate cooperation between various optimization procedures that are usually problem-specific [9]. A metaheuristic algorithm or solver is a particular implementation exploiting the metaheuristic paradigm. Creating a metaheuristic algorithm tailored to one particular problem can be very time-consuming. For this reason, it is challenging research goal to develop a general-purpose metaheuristic solver that would play a similar role to heuristics as IP solvers for exact algorithms. Such solvers already exist, but they are limited to a common solution representation, e.g., permutations, binary, integer, or real vectors, or to modeling paradigms, such as Integer Programming (IP) or Constraint Programming (CP).

This thesis aims to propose a neighborhood-oriented metaheuristic solver and compare its performance to the IP Gurobi solver on four different NP-hard problems. Namely, Capacitated Vehicle Routing Problem (CVRP), Quadratic Assignment Problem (QAP), Non-Permutation Flowshop (NPFS), and Sudoku. The proposed solver is capable of dealing with problems with dynamic permutative solution representation. Dynamic permutative representation means that the solution is represented by an ordered sequence of potentially recurring nodes and that the length of the permutation may change during the optimization. The neighborhood-oriented approach is expected to exploit problem structure while preserving versatility and scalability. Gurobi was chosen because it is also a multipurpose tool, and this thesis intends to show that the proposed metaheuristic solver can outperform the exact solver in terms of solution quality given a limited amount of time.

Chapter 2
Literature review

In the first section of this chapter, some basic terminology is presented regarding the types of algorithms used for combinatorial optimization. The second part provides an overview of general-purpose tools used for solving broad groups of COPs, usually based on a common problem or solution representation.

All optimization algorithms can be classified into one of three main groups of algorithms: Exact, approximation, and heuristic; see Figure 2.1 for reference. The figure is not by any means an exhaustive list of algorithms. Instead, it presents some of the most famous algorithms as a typical representatives of each algorithm type.

Exact algorithms always guarantee to find the optimal solution. Branch-and-bound and cutting planes are some of the exact algorithms one can use to solve IP. Another example is breadth-first-search or its improved version Backtracking used in Constraint Programming [10]. The downside of using exact algorithms is very poor scalability for solving NP-Hard problems, and so they are used only for small instances.

Approximation algorithms try to deal with NP-hard problems in a more efficient way in the terms of computation budget. Assuming that P≠NP, a broad class of optimization problems, cannot be solved optimally in polynomial time. Approximation algorithms are polynomial algorithms that can find an approximate solution while also providing a guarantee regarding the quality of that solution. Usually, the guarantee of such algorithms is expressed as a multiplicative factor, meaning the optimal solution is always guaranteed to be within a multiplicative factor of the produced solution. Sadly development of an approximation algorithm is problem-specific and cannot be generalized. A famous example is Christofides' algorithm for TSP, which guarantees that its solutions will be within a factor of 3/2 of the optimal solution [11].

Heuristic algorithms seek for high quality solutions at a reasonable computational time, but can not guarantee that a problem will be solved in terms of obtaining the exact solution. They can be used as a stand-alone method, but they are also sometimes used in exact algorithms to provide an upper bound on solution quality. Some implementations of already mentioned Branch-and-bound use a heuristic to choose which branch to explore the first [12].

There are several ways for heuristics classification. Regarding the number of candidate solutions handled in each step, heuristics are classified into single-solution and population-based. Single-solution keep only the best candidate solution. In each iteration, the algorithms search through the candidate's neighborhood. Then it replaces the candidate with another, often the best, solution found. Famous single-solution heuristics include Variable Neighborhood Search [13] and Iterated Local Search [14]. Population-based, on the other hand, maintain a set of candidate solutions and perform search in multiple parts of solution space. Evolutionary Algorithms (EA) [15] are notable representatives of population-based heuristics.

Regarding whether the algorithm maintains some kind of memory of previous steps, heuristics can be divided into memory-based and memory-less. Memory can serve as a

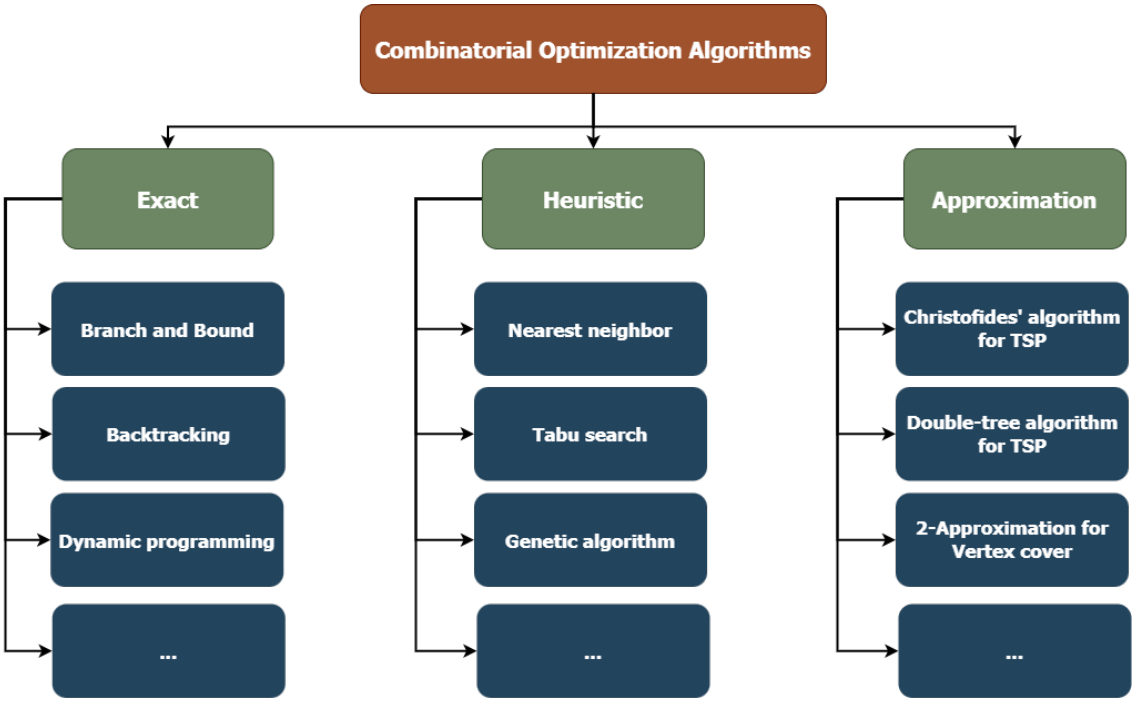

Figure 2.1. Classification of the combinatorial optimization algorithms.

prevention against stepping into already explored solutions (Tabu Search [16]), or as a learning mechanism (Ant Colony Optimization [17]).

Another property of heuristic is whether it is a constructive or local search. Constructive start from an empty solution and append it in every iteration until the complete solution is generated. On the other hand, local search heuristics begins with some random initial solution, and then it searches the solution space making local changes to the previous solution.

Essential property of any algorithm is whether it is deterministic or stochastic. Deterministic algorithms follow a rigorous procedure. Their progress and final results are repeatable, meaning they perform precisely the same search in every algorithm run for a given instance and parameters. A classical example of a deterministic heuristic is Nearest Neighbor algorithm for TSP.

On the other hand, stochastic algorithms always have some randomness in them. Evolutionary Algorithms are a good example. EA represent a broad set of population-based optimization approaches. As the name suggests, they take inspiration from biological processes, like mutation, recombination, and selection. The solution candidates in the population will be different each time the algorithm is executed since EAs use random number generators to simulate recombination and mutation. Though the final result may be similar or even same, the paths to each result are probably different. [18–19]. Another popular stochastic heuristic is Simulated Annealing [20], which randomly accepts suboptimal candidates to escape local optimum.

2.1 General-purpose optimization paradigms

In this thesis, the word paradigm means a problem formulation and solution search pattern. A solver is a particular software that employs such a paradigm. This section discusses the most renowned general-purpose paradigms for COPs and some algorithms that are used in solvers that exploit these paradigms.

In the first two subsections, Integer Programming and Constraint Satisfaction Problem paradigms are introduced as well as the most generic exact algorithms that solve them. The third section is focused on multipurpose metaheuristic solvers, some of which are used as an alternative to the exact algorithms for integer programming and constraint satisfaction. The last section discusses transformation of COPs into other well-studied COPs with efficient problem-specific solvers. Rather than optimization paradigm, it is an important concept allowing reusability of solvers and should also be considered as a solution approach.

2.1.1 Integer Programming

Probably the most renowned paradigm in combinatorial optimization is **Integer Programming** (IP). It has gained popularity because of the versatility of the integer program formulation and because there are efficient algorithms that can solve it. If the objective function and constraints of the program are linear, it is called **Integer Linear Program** (ILP), and it can be formulated as the following minimization problem:

$$\begin{aligned} \text{min:} \quad & f(x) = c^T x \\ \text{subject to:} \quad & Ax \leq b \\ \text{where:} \quad & x \in \mathbb{Z}_+^n \\ & A \in \mathbb{R}^{n \times m} \\ & b \in \mathbb{R}^m \\ & c \in \mathbb{R}^n \end{aligned} \tag{2.1}$$

A special case, when all the optimization variables x are restricted to a value of either zero or one, is called 0-1 Linear Programming. Another commonly used term is **Mixed Integer Linear Program** (MILP), which stands for linear programs that have some optimization variables x real and some are integers.

Another practical type of IP is **Integer Quadratic Program** (IQP), that is, integer program that has quadratic objective function. Nonlinear constraints can make the problem much harder. In fact integer programming with quadratic constraints is undecidable, i.e., there cannot exist an algorithm to solve this problem [12].

All these types of ILP are mentioned here to demonstrate how versatile it really is. Because ILP and 0-1 linear programs are special cases of MILP, they can all be solved using one solver, e.g., Gurobi, CPLEX, or many others. Most generic implementations of solvers are based on branch-and-bound (B&B) and simplex algorithms. In fact, some variant of B&B is also used in Gurobi that was used as a comparison to the proposed solver.

The simplex method is the most commonly used algorithm for solving linear programs, i.e., programs that have linear constraints and objective functions, but the optimized variables are real, not integers. It has exponential worst-case complexity, but amazingly, it was shown to converge in expected polynomial time on various distributions of random inputs. This fact makes it very useful in practice [21].

At first sight, it might seem that dealing with ILPs might be easier than with LPs since the set of possible solutions is smaller. Unfortunately, the opposite is true. ILPs are NP-hard. But fortunately, we can exploit the simplex method to solve ILP faster.

The relaxation of ILP_R is a linear program that has the same objective function and constraints as the original ILP, but the domain of optimized variables is changed to real numbers x. Since possible solutions of the original ILP are a subset of its relaxation, it

means that $OPT(ILP_R) \leq OPT(ILP)$. Therefore we can use simplex to find a lower bound (LB) of the original ILP [22].

Branch-and-bound is an exact deterministic algorithm that utilizes a tree data structure for candidate solution set exploration. The root node represents the whole set of solutions. At every iteration, B&B splits into branches of solution subsets. The algorithm solves relaxation in every node to find the lower bound for the current branch [12]. Each branch's lower bound is checked against the global upper bound (GUB), and if the $LB > GUB$, the whole branch is discarded (pruned) [22]. because it cannot lead to the optimal solution. If the optimum of the current node is an integer for all $x_i \in x$, feasible solution is found and the node is not further branched. It is also used to update GUB if applicable.

This is the most generic version of B&B, but definitely not the only one. There are several issues which need attention and which can improve performance of the algorithm significantly [22], like search strategy [23], i.e., the order in which nodes are explored, the branching strategy [24], i.e., how the solution space is partitioned to create new nodes in the tree, and the pruning rules [25], i.e., rules that prevent visiting branches that can't contain optimal solution. Currently, the most successful method for solving integer programs is branch-and-cut. It is obtained by adding a cutting-plane step to tighten the lower bound before every branching step. For more details consult [12].

2.1.2 Constraint Satisfaction Problem

The constraint satisfaction problem (CSP) is a powerful paradigm for solving combinatorial problems based on constraints. A constraint is a logical relation between one or more variables, each taking a value in a given finite domain. The idea of CSP is that the user defines a set the constraints and a general-purpose constraint satisfaction solver, like CPLEX[26] or Gecode[27], is used to find a solution that satisfies all of them. Formally CSP is a triple $\langle X, D, C \rangle$, where

$$\begin{aligned} X &= \langle X_1, \ldots, X_n \rangle \text{ is a set of n discrete variables} \\ D &= \langle D_1, \ldots, D_n \rangle \text{ is a set of n finite domains such that } X_i \in D_i \\ C &= \langle C_1, \ldots, C_t \rangle \text{ is a set of t constraints restricting the valid values of } X \end{aligned} \quad (2.2)$$

and the solution of CSP is n-tuple $A = \langle a_1, \ldots, a_n \rangle$ where $a_i \in D_i$ for which each C_j is satisfied [28]. There are several types of objectives one may pursue using CSP, e.g., finding any solution, finding all the solutions, and finding an optimal solution regarding some given objective function. The latter of the three is usually referred to as the constraint satisfaction optimization problem (CSOP) [29].

The main algorithm for solving CSPs is backtracking search [28]. It is quite similar to the branch-and-bound algorithm. It is also an exact algorithm that performs a depth-first search of a gradually generated search tree, where each node represents an assignment of a value to a variable, and each branch represents a partial solution. In other words, It builds up partial solutions by choosing values for variables until it reaches a dead-end, where the partial solution cannot be extended without violating some constraint. When this happens, it backtracks to the last choice it made and tries another branch. This is done until the objective is fulfilled, i.e., a feasible or optimal solution is reached depending on the type of assignment.

Many techniques for improving the efficiency of a backtracking search algorithm have been suggested and evaluated, including constraint propagation, no-good recording, back-jumping, heuristics for variable and value ordering, and randomization and restart strategies [30].

CSP is in many ways similar to ILP. If a problem formulation has a linear objective function and all the constraints in C are linear, it can be solved by both CSP and ILP solvers. The fundamental difference between ILP and CSP is that CSP is not limited to arithmetic constraints (equalities and inequalities), but can use any relation $R \subseteq D_1 \times \ldots \times D_n$.

An important type of constraint is a so-called global constraint. A global constraint is a single constraint that represents a highly structured set of constraints. An example would be an *alldifferent* constraint that requires a set of variables to take distinct values. It simplifies the problem formulation as it represents a large set of equations while at the same time solvers can take advantage of the special structure of the constraint and therefore can be implemented much more efficiently [31].

2.1.3 Metaheuristic solvers

Metaheuristics are a group of very flexible paradigms that can be used to design heuristics for virtually any combinatorial optimization problem. They are higher-level strategies that combine one or more optimization procedures, typically heuristic but might be exact as well, and other methods capable of escaping from the local optima trap and performing a robust search of a solution space [9, 32].

This flexibility also comes at a cost: Researchers usually spend a large amount of time properly designing and tuning their metaheuristic. As observed in many published papers, designing an efficient metaheuristic requires a lot of intuition on the part of the metaheuristic designer [33].

Fortunately, there are some solvers that exploit metaheuristic paradigms while preserving various levels of generality. One of the most notable metaheuristic solvers is LocalSolver. It is commercial software that does not rely on a single optimization approach. It combines several exact and heuristic methods dynamically during the computation. Similar to an ILP solver, "LocalSolver searches for feasible or optimal solutions, computes lower bounds and optimality proofs, and can prove the inconsistency of an optimization model. But LocalSolver not only achieves this on linear models but also on highly nonlinear and combinatorial models, involving high-level expressions like collection variables, arrays, or even external functions" [34].

Another interesting solver is OptaPlanner. It is an open source competitor to LocalSolver developed by Red Hat. "OptaPlanner combines sophisticated Artificial Intelligence optimization algorithms (such as Tabu Search, Simulated Annealing, Late Acceptance and other metaheuristics) with very efficient score calculation and other state-of-the-art constraint solving techniques for NP-complete or NP-hard problems" [35].

Evolutionary algorithms are a set of metaheuristic population-based optimization paradigms. There are frameworks that have implemented most popular EAs and let user define new ones, such as Evolutionary Computation Framework (ECF) [36], Distributed Evolutionary Algorithms in Python (DEAP) [37], Evolving Objects [38], and many more. These frameworks let's user define a fitness function, i.e., equivalent of objective function in IP, and choose a genotype, i.e., candidate solution representation. This representation is usually either binary vector, real vector, permutation or a tree-like data structure.

A less versatile yet powerful solver is LKH-3 [39]. It is an extension to the efficient Lin-Kernighan-Helsgaun TSP solver (LKH) [40]. "LKH is powerful local search heuristic for the TSP based on the variable depth local search. LKH has produced optimal solutions for all solved problems we have been able to obtain; including an 85,900-city instance."

[41] LKH-3 extends the original LKH with the possibility to handle constraints and multiple traveling salesmen which increases the number of problems it can solve to 39. However, they are all variations of TSP and Vehicle Routing Problem (VRP) [39].

2.1.4 Transformations to Fundamental Combinatorial Optimization Problems

Another important solution approach is to make use of an efficient problem-specific solver by transforming the given optimization problem into the problem that was the solver designed for. This option should be considered before using a general-purpose solver as specialized solver will most probably offer better results in terms of computation time and solution quality. For example, Flow Shop Scheduling Problem (FSSP) is a famous scheduling problem that can be transformed and solved by TSP solver.

The goal of Traveling Salesman Problem is to find Hamiltonian cycle in an undirected weighted graph that minimizes the sum of distances (weights) of the edges in the cycle. Instance of TSP can be represented as distance matrix.

FSSP can be briefly described as problem of assigning processing order of N jobs to M machines where each job consists of M operations. All jobs have the same processing operation order when passing through the machines but can have different processing time. The order of jobs must be the same on each machine. The goal of FSSP is to find such ordering that minimizes the makespan, i.e., total processing time [42].

To formulate FSSP as TSP, it is necessary to formulate FSSP using distance matrix. To do this, distance can be defined as processing times of both jobs in a following manner: "The distance $D_{a,b}$ is a measure of increase of makespan if job b is scheduled after job a, taking into account only jobs a and b."[43]

Another well known example of problem transformation is the Noon-Bean transformation [44] which can be used to model the General Traveling Salesman Problem as a standard Traveling Salesman Problem.

Chapter 3
Methodology

This chapter describes the proposed metaheuristic solver. It also discusses four problems that were used for its benchmarking, specifically Capacitated Vehicle Routing Problem (CVRP), Quadratic Assignment Problem (QAP), Non-Permutation Flowshop (NPFS), and Sudoku. These four problems were selected because their formulations in the proposed solver are markedly different. But more to that later in this chapter.

The first section of this chapter provides the formal definition of the group of problems that the proposed solver can address. Additionally, it introduces the four benchmarking problems. The second section describes the generic part of the solver, i.e., the segment of the solver that facilitates the optimization based only on fitness function and bounds vectors. It is a neighborhood-oriented framework that offers users to choose from multiple popular local search algorithms. The last section demonstrates how can the four problems be formulated and implemented in the proposed solver framework. Furthermore, it presents MILP formulations used for the comparison with the Gurobi optimizer.

3.1 Problem definitions

3.1.1 Generic problem definition

The proposed solver addresses problems that can be defined as follows:

$$\text{Given set of nodes:} \quad A = \{1, \ldots, n\}, \qquad A \subset \mathbb{Z}_+ \tag{3.1}$$

$$\text{and bounds:} \quad LB = [l_1, l_2, \ldots, l_n], \quad LB \in \mathbb{Z}_+^n \tag{3.2}$$

$$UB = [u_1, u_2, \ldots, u_n], \; UB \in \mathbb{Z}_+^n \tag{3.3}$$

$$\text{Minimize fitness function:} \quad g(X): \quad A^m \to \mathbb{R} \tag{3.4}$$

$$\text{where:} \quad X = [x_1, x_2, \ldots, x_m], \quad X \in A^m \tag{3.5}$$

$$F = [f_1, f_2, \ldots, f_n], \quad F \in \mathbb{Z}_+^n \tag{3.6}$$

$$f_i = \sum_{j=1}^{n} [\![x_j = a_i]\!], \quad \forall i \in [1, n] \tag{3.7}$$

$$l_i \leq f_i \leq u_i, \qquad \forall i \in [1, n] \tag{3.8}$$

In other words, given a set of nodes A with two vectors of upper bounds UB and lower bound LB, find a vector X that minimizes fitness function g while also enforcing node frequency f_i to be between bound l_i and u_i for every node. Frequency f_i represents a number of the nodes a_i in the solution X. The fitness function g does not need to be linear or quadratic, as is the case with ILP, but instead can be an arbitrary function $A^m \to \mathbb{R}$. Vector X can be a permutation, permutation with repetition, variation, variation with repetition, or virtually any vector of whole numbers. It can also have variable length when $LB \neq UB$.

3.1.2 Capacitated Vehicle Routing Problem

Vehicle Routing Problem (VRP) is a generalization of TSP and was first proposed by George Dantzig, and John Ramser in 1959 [45]. Formally it can be defined [46] as:

Definition 3.1. Let $G = (V, E)$ be a graph where $V = \{0, 1, 2, \ldots, n-1\}$ is a set of nodes (e.g. customers) and E is a set of edges $e = \{i, j\},\ i \neq j$ with assigned weight c_e (e.g. travel cost). Additionally, assume that the available vehicles are based at the depot $D \in V$. VRP consists of designing a set of routes in such a way that:

- Sum of the routes cost is minimized.
- Each customer in $V \setminus D$ is visited exactly once by exactly one vehicle.
- All vehicle routes start and end in the depot.

Capacitated Vehicle Routing Problem (CVRP) is VRP with capacity constraints. In this specific case, every customer i has some quantity demand $q_i \geq 0$ and every vehicle has capacity $Q > 0$. To satisfy the constraint, the sum of demands in each route must be less or equal to the vehicle's capacity assigned to this route. By associating a binary variable x_e with each edge e in the graph and assuming that depot $D = 0$, we obtain the following formulation [47]:

$$\min \sum_{e \in E} c_e x_e$$

$$\sum_{e=\{0,j\} \in E} x_e = 2k \tag{3.9}$$

$$\sum_{e=\{i,j\} \in E} x_e = 2,\ \forall i \neq j \in V \tag{3.10}$$

$$\sum_{e=\{i,j\} \in E} x_e \geq 2b(S), \quad \forall S \subset V,\ |S| > 1,\ i \in S, j \notin S \tag{3.11}$$

$$b(S) = \left\lceil \frac{\sum_{i \in S} q_i}{Q} \right\rceil, \tag{3.12}$$

where Equations (3.9) and (3.10) are degree constraints and Equations (3.11) and (3.12) enforce the connectivity of the solution, as well as that no route has total demand exceeding the capacity Q. Figure 3.1 shows an example instance of CVRP.

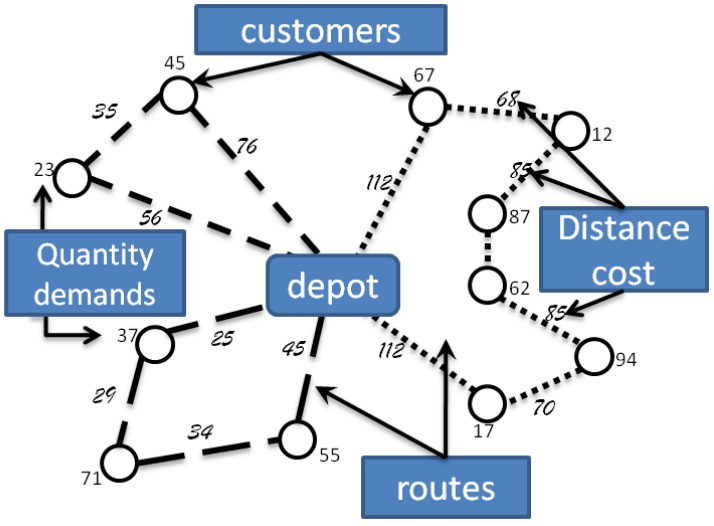

Figure 3.1. Capacitated Vehicle Routing Problem example [48].

3.1.3 Quadratic Assignment Problem

Quadratic Assignment Problem (QAP) is a famous combinatorial problem that was first introduced by Koopmans and Beckmann in 1957 [49]. QAP can be described as a problem of assigning n facilities to n locations, with a cost function given by the product of distance and flow between the facilities. The objective is to minimize the cost function [2].

Definition 3.2. We are given two matrices $F = (f_{ij})$, $D = (d_{kl}) \in \mathbb{R}^{n \times n}$, where f_{ij} is the flow between the facilities i and j and d_{kl} is the distance between the locations k and l. Problem QAP(F, D) can be mathematically formulated as:

$$\min_{\varphi \in S_n} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\varphi(i)\varphi(j)}, \tag{3.13}$$

where S_n is a set of all possible permutations φ of the vector $[1, 2, \ldots, n]$, where $\varphi(i)$ represents assigning the facility i to the location $\varphi(i)$. [50]. Figure 3.2 instance of QAP with four locations and facilities.

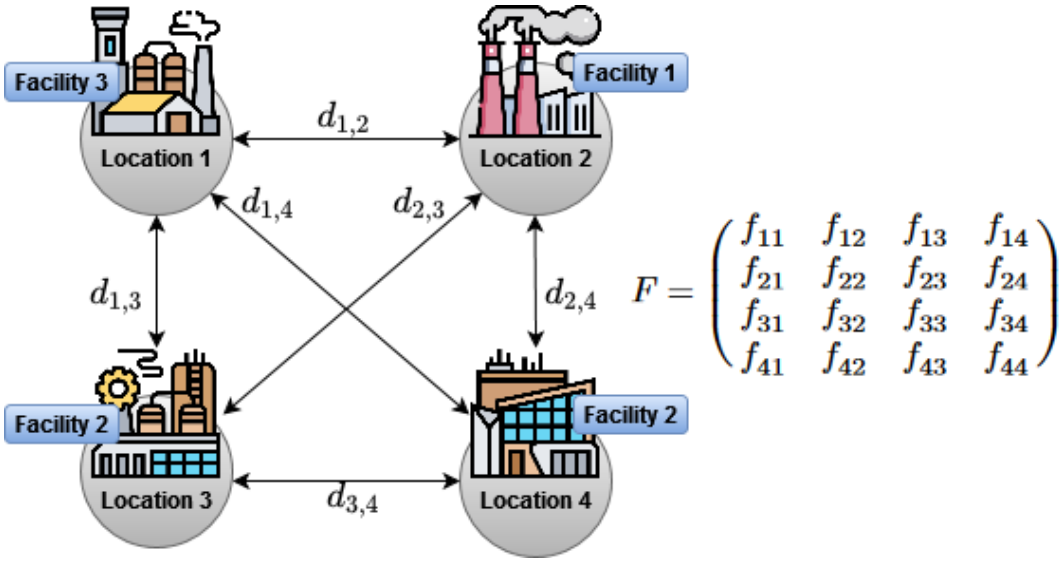

Figure 3.2. Quadratic Assignment Problem.

3.1.4 Non-Permutation Flow Shop

In Non-Permutation Flow Shop Problem (NPFS) we are given a set of m machines $M = \{m_1, m_2, \ldots, m_m\}$ and n jobs $J = \{j_1, j_2, \ldots, j_n\}$. We are also given the processing times p_{ij}, which denote how long the machine m_i need to process the job j_j. The goal of NPFS is to find job ordering for every machine that results in minimal makespan C_{max}. Makespan is the time at which the last machine m_m finishes the last of its jobs. A valid NPFS solution must also satisfy the following conditions [51]:

- Each machine must process each job without interruption.
- At most one machine can process any job at any given instant.
- Jobs must finish on machine m_i before the start of the processing on machine m_{i+1}.

Definition 3.3. We are given a number of machines m and jobs n and a matrix $P = (p_{ij})$, $P \in \mathbb{R}^{m \times n}$, where p_{ij} is the processing time of job j_j on machine m_i. Lets denote x_{ij}

the starting time of job j_j on machine m_i. NPFS can be mathematically formulated as an MILP [52]:

$$
\begin{aligned}
&min \quad C_{max} \\
&s.t. \quad x_{mj} + p_{mj} \leq C_{max} && \forall j \in \{1,\dots,n\} && (3.14)\\
&\quad x_{ij} + p_{ij} \leq x_{i+1,j} && \forall i \in \{1,\dots,m-1\}, \forall j \in \{1,\dots,n\} && (3.15)\\
&\quad x_{ij} + p_{ij} \leq x_{ij'} + M(1 - y_{ijj'}) && \forall i \in \{1,\dots,m\}, \forall j \neq j' \in \{1,\dots,n\} && (3.16)\\
&\quad y_{ijj'} + y_{ij'j} = 1 && \forall i \in \{1,\dots,m\}, \forall j \neq j' \in \{1,\dots,n\} && (3.17)\\
&\quad x_{ij} \geq 0 && \forall i \in \{1,\dots,m\}, \forall j \in \{1,\dots,n\} && (3.18)\\
&\quad y_{ijj'} \in \{0,1\} && \forall i \in \{1,\dots,m\}, \forall j \neq j' \in \{1,\dots,n\} && (3.19)
\end{aligned}
$$

where $y_{ijj'}$ is a binary variable that represents that the job j_j precedes job $j_{j'}$ on machine m_i.

In figure 3.3 you can see an optimal solution to the NPFS instance of 6 jobs on 6 machines given by processing time matrix in (3.20) computed using the Gurobi Optimizer.

$$
P = \begin{pmatrix}
1 & 1 & 2 & 2 & 3 & 1 \\
3 & 2 & 3 & 1 & 2 & 2 \\
2 & 4 & 3 & 4 & 2 & 3 \\
3 & 2 & 3 & 2 & 1 & 1 \\
3 & 2 & 2 & 1 & 4 & 2 \\
2 & 1 & 2 & 1 & 1 & 3
\end{pmatrix} \tag{3.20}
$$

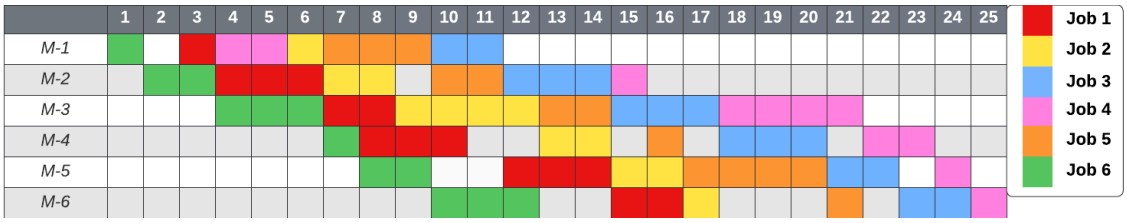

Figure 3.3. Non-Permutation Flowshop Problem example.

3.1.5 Sudoku

Sudoku is a logic-based puzzle designed by Howard Garns. To solve the puzzle, the player must fill in an $n \times n$ matrix so that each row, column, and $m \times m$ submatrix contains each integer 1 through n exactly once. The most common version of Sudoku is $n = 9, m = 3$. However, size m can be any positive integer and n must satisfy equation $n = m^2$. Each instance of Sudoku has some of the matrix fields given, while others are to be filled by the player. The number of given fields determines the instance's difficulty. [53].

Definition 3.4. A general Sudoku is defined [54] as:

- a set $\mathcal{S}$ of n^2 fields (grid squares),
- an index set $\mathcal{I} = \{1, 2, \dots, n\}$,
- a collection $\mathcal{B}$ of conjunctive subsets of $\mathcal{S}$, e.g. rows, columns, and submatrices. Each set $B \in \mathcal{B}$ consists of exactly n fields in $\mathcal{S}$,
- an initial assignment $\mathcal{A} = \{(p_i, k_i),\ i = 1, \dots, r\}$, with field $p_i \in \mathcal{S}$ assigned index $k_i \in \mathcal{I}$.

The goal is to assign indices from $\mathcal{I}$ to the remaining unassigned fields from $\mathcal{S}$ so that all sets B contain each index from $\mathcal{I}$ exactly once. A valid Sudoku instance has only one correct solution. The solution matrix $X = (x_{pk}), X \in \mathbb{R}^{n^2 \times n}$ can be expressed mathematically as:

$$\sum_{k \in \mathcal{I}} x_{pk} = 1, \quad p \in \mathcal{S} \tag{3.21}$$

$$\sum_{p \in B} x_{pk} = 1, \quad B \in \mathcal{B}, k \in \mathcal{I} \tag{3.22}$$

$$x_{p_i,k_i} = 1, \quad i = 1, \ldots, r \tag{3.23}$$

where x_{pk} is assignment of index k to the field p. Equation (3.21) forces each field to hold exactly one index. Equation (3.22) ensures that every block contains all the indices and Equation (3.23) guarantees that the initial assignment will hold in the final solution. [54]

In Figure 3.4, an example of 9×9 Sudoku is shown. This is the most common $n = 9, m = 3$ version of Sudoku. Nevertheless, this thesis addresses any kind of square Sudoku, i.e., the type of sudoku where only subsets represented by rows, columns, and square submatrices are allowed.

	4				1		6	8
3	6		7	4			9	
7		2		8			5	
4	3		8	6			7	5
		7						
		6	4			3		1
6		3				8	1	
	7				6	5		
	5	9		7				

9	4	5	2	3	1	7	6	8
3	6	8	7	4	5	1	9	2
7	1	2	6	8	9	4	5	3
4	3	1	8	6	2	9	7	5
2	9	7	5	1	3	6	8	4
5	8	6	4	9	7	3	2	1
6	2	3	9	5	4	8	1	7
8	7	4	1	2	6	5	3	9
1	5	9	3	7	8	2	4	6

Figure 3.4. Example of unsolved (left) and solved (right) Sudoku [55].

3.2 Proposed framework design

This chapter gives an overview of the proposed solver building blocks. It consists of two main components: generic solver and problem-specific component. As the name suggests, the generic solver component is the problem-independent part of the solver. It contains three classes: Instance, Solution, and Generic Optimizer. For illustration, see Figure 3.5.

The Instance is an abstract class that defines mandatory properties that every problem instance must have defined. Those are fitness function g, number of nodes A, and the lower LB and upper bounds UB of the node frequency F in the solution. To address a new problem using the proposed solver, a user has to define a new class, which inherits from this one. It has to set all the mandatory properties and implement a corresponding fitness function g.

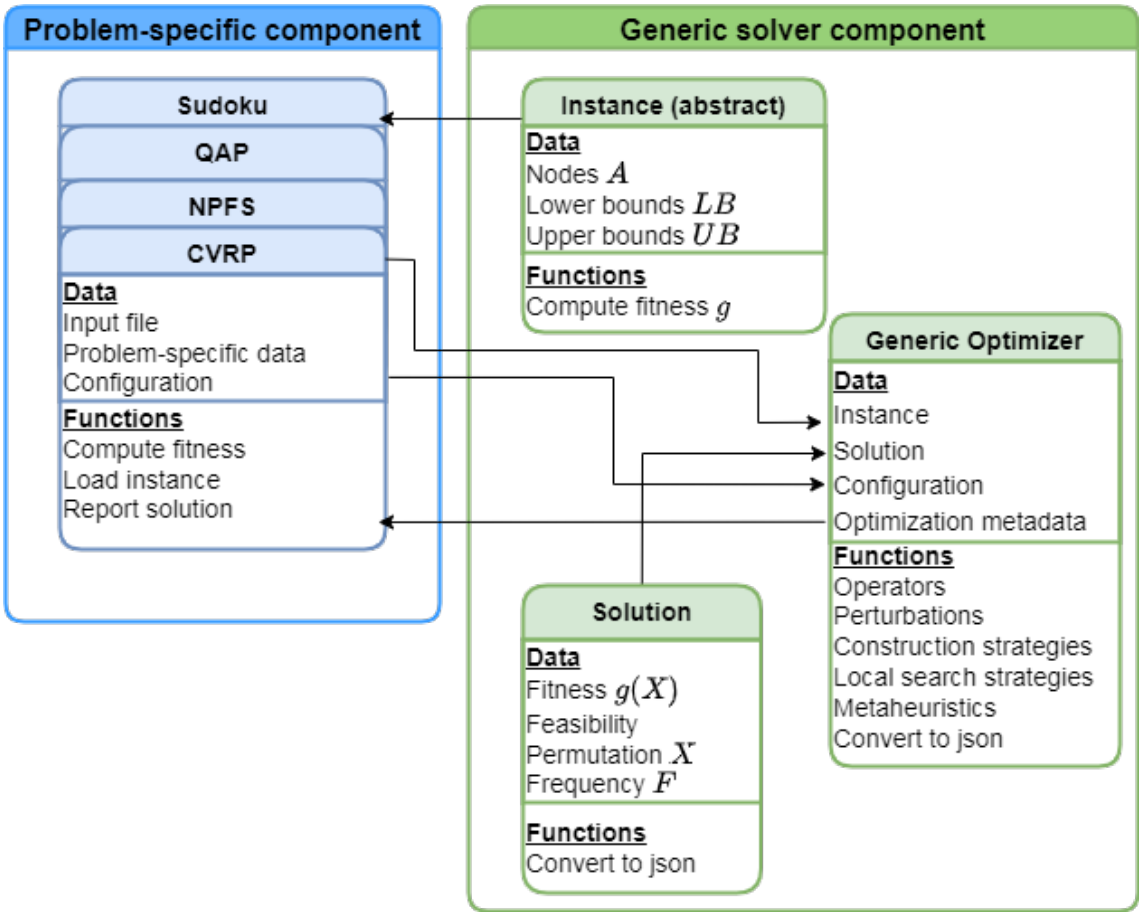

Figure 3.5. Proposed solver class diagram.

The solution class holds the solution vector X and the frequency vector F. It also contains the fitness $g(X)$ of the solution vector X and the information about whether this solution is feasible or not. Additionally, it has a function that exports the solution to JSON. The most important class is the Generic Optimizer. It is the optimization engine that works without the knowledge of the underlying problem with the exception of the information that is provided through the Instance class, i.e., number of nodes $|A|$ and vectors LB, UB. It contains five main groups of functions. These are operators (section 3.3.1), perturbations (section 3.3.2), construction strategies (section 3.3.3), local search strategies (section 3.3.4), and the metaheuristics (section 3.3.5).

The operators form the basis of the optimization engine. They perform the search in a given neighborhood and return the solution X with the best fitness $g(X)$ inside of that neighborhood. In the scope of this thesis, neighborhood $\mathcal{N}_k(X)$ is defined as follows:

Definition 3.5. Given a solution space S and an operator function $\rho(X, i)\colon S \to S$, which represents some transformation of the solution $X \in S$. This transformation is parameterized by a variable i, defined on some finite domain D. Neighborhood $\mathcal{N}_k(X)$ of the solutions X is a set of candidate solutions $\rho(X, i)$, where X is fixed, and i takes any value from its domain D:

$$\mathcal{N}_k(X) = \{\rho(X, i) \in S | \ \forall i \in D\} \tag{3.24}$$

The function ρ can be an arbitrary function that returns a solution $Y \in S$. Note that Y can even be incomplete solution, i.e., Equation (3.8) can be violated for some indices.

The proposed solver offers ten different operators, like insert, two-opt, move, swap, and many more.

The perturbations are randomized operators. They are intended to escape the local optimum by making a random, usually non-improving, selection of a new solution X outside of the previously explored neighborhoods. There are six perturbations to choose from: Double Bridge, Random-Reverse Double Bridge, Reinsert, Random Move, Random Swap, and Random Move-All.

The construction strategies are functions that are used to generate an initial solution. There are three types of construction strategies implemented: greedy, random, and random replicate. Random creates a random solution that satisfies UBs and LBs. Random replicate generates one permutation of all the nodes and then appends the solution with the replicas of this permutation until LBs are satisfied. Greedy constructs the solution iteratively, evaluating fitness after all possible node insertions and inserting the best node at every step. Therefore the user-defined fitness function must return a reasonable value even for incomplete solutions.

Local search strategies are functions that select the order in which the neighborhoods are explored. They perform the exploration until none of the operators are able to find an improving solution. The proposed solver implements five variations of Variable Neighborhood Descent (VND): The Basic Variable Neighborhood Descent (BVND), Pipe Variable Neighborhood Descent (PVND), Cyclic Variable Neighborhood Descent (CVND), Random Variable Neighborhood Descent (RVND), and Random Pipe Variable Neighborhood Descent (RPVND).

Finally, the metaheuristics are the most high-level functions that make use of all the other types of functions in the optimizer. The proposed solver implements some established neighborhood-oriented metaheuristics, namely Iterated Local Search (ILS) and Basic Variable Neighborhood Search (BVNS). Additionally, an extension to BVNS called Calibrated Variable Neighborhood Search is proposed later in this chapter. All these metaheuristics work on a similar basis. First, they generate an initial solution using one of the construction strategies. Then they iteratively call local search strategies, which search for the local optimum in the selected neighborhoods, i.e., minimizing fitness using the operator functions. After reaching a local optimum, the current solution is randomly altered by a perturbation function to escape the local optimum. The process is repeated until a timeout is reached.

During the optimization, the generic optimizer collects metadata. This includes a histogram of improving operators' calls, and a graph of the fitness w.r.t. the elapsed computation time. Steps in the graph are made after every improving local search call because there would be too many records if the graph included all the operators as well.

Users can define which components the optimizer invokes in a JSON configuration file. There are also other parameters that a user should configure, like the number of threads in which the optimizer should run or the maximum computation time, which is currently the only end condition for the optimization loop. Another important setting of the optimizer decides whether the optimizer should accept infeasible solutions after every perturbation. If not, it would reset the solution to the state before perturbation and repeat the perturbation until a feasible solution is acquired. Note that the perturbations are randomized selection functions, and the set of candidate solutions they choose from is large, so the probability of choosing an already explored solution is low.

The problem-specific component is the part that is implemented by the user. Nevertheless, this is not a tedious process. The amount of work is comparable to formulating a problem as an ILP or CSP. The user has to implement a problem-specific Instance

class. It has to initialize the mandatory properties of the abstract Instance class like LB, UB vectors and the fitness function $g(X)$. The user should also choose a suitable configuration for the given problem. The configuration can be tuned either in the trial-error manner or by some specialized tuning tool like irace [56].

The fitness function $g(X)$ should return the fitness of a solution X and its feasibility. The fitness function has to be designed efficiently both in terms of complexity and implementation as it is the most frequently called function in the whole optimization process. The overall performance is heavily affected by the fitness function.

Users should also consider adding some kind of penalty $p(X)$ to the fitness function $g(X)$. The purpose of penalty functions is to add penalties to infeasible solutions so that solutions that are close to feasible solutions are penalized less than solutions that are farther. All the problem-specific components implemented for the purpose of this thesis use the static penalization function. Static penalization computes penalty based only on the solution X, but there are also other types of penalization like dynamic, which usually increases over time, or adaptive, which depends on the current solution quality.

3.3 Generic solver components

This section provides a detailed description of individual generic solver components. Section 3.3.1 describes all the operators. In Section 3.3.2 are presented the perturbations. Section 3.3.3 introduces the construction strategies. Local search strategies are described in Section 3.3.4. Finally, Section 3.3.5 describes the metaheuristics.

For more clarity some of the frequently used terms in this section are explained. The elements of $A = \{1, 2, \ldots, n\}$ are the nodes a of a given problem. Solution X is a vector with variable length m whose elements are the nodes from A. Solution substring is a vector $X_{sub} = (x_i, \ldots, x_j)$ that is equivalent to some part of the original vector $X = (x_1, \ldots, x_m), \quad 1 \leq i < j \leq m$. i-th element of vector X is denoted X_i. The frequency vector is denoted by F and corresponds to the original solution X. F_a is frequency of node a in the solution X. Additionally, the LB_a and UB_a are lower bound and upper bound of the frequency F_a.

3.3.1 Operators

This section describes all the implemented operators and provides their pseudocodes with an example. In the following text, time complexities of find, insert, remove, and reverse modifications of the vector X, are considered linear, i.e., $O(m)$. Please note the difference between operators and modifications. Modification is an atomic change performed on a vector X. Operators try out all possible modifications and select the best one.

3.3.1.1 Insert

Insert is an operator that is used mainly but not exclusively during the initial solution generation. It tries to insert one node $a \in A$ to all possible locations i in the solution X. After iterating through all possibilities, it returns the solution with the best fitness $g(X_{\text{best}})$. If there was no improving modification the algorithm returns the original solution. Pseudocode is shown in the Algorithm 3.1. On lines 6-7, the function also checks that the frequency F_a of the node a is not greater than or equal to UB_a. If that is the case, it does not try to insert node a. An example insert operation, where $i = 2$ and $a = 4$, is illustrated in Figure 3.6. The time complexity of the insert operator

is $O(|A|m^2)$, where $|A|$ is the number of nodes and m is the length of X. Note that the length of the solution increases, so this operator cannot be used for problems with $UB = LB$ with the exception of initial solution construction.

```
1  function insert(X)
2  X_best ← X
3  for  i in range from 0 to m
4      X_new ← X
5      for node a ∈ A
6          if F_a ≥ UB_a
7              continue
8          insert node a into X_new at position i
9          if g(X_new) < g(X_best)
10             X_best ← X_new
11 return X_best
```

Algorithm 3.1. Insert operator pseudocode.

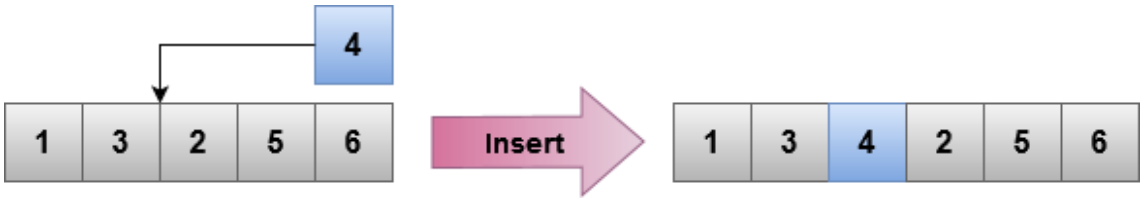

Figure 3.6. Insert operator example where $i = 2$ and $a = 4$.

3.3.1.2 Remove

Remove is a counterpart to the insert operator. It tries to remove one of the nodes X_i from all possible locations i in the solution X. After trying out all possibilities, it returns the solution with the best fitness $g(X_{\text{best}})$. If no improving modification was made, the algorithm returns the original solution. Pseudocode is shown in Algorithm 3.2. On lines 6-7, Remove also skips the node a if the frequency F_a is less than or equal to LB_a. An example remove operation, where $i = 2$ and $a = 4$, is illustrated in Figure 3.7. The time complexity of the remove operator is $O(m^2)$, where m is the length of the X. Note that the length of the solution decreases, so this operator cannot be used for problems with $UB = LB$.

```
1  function remove(X)
2  X_best ← X
3  for i in range from 0 to m
4      X_new ← X
5      a ← X_i
6      if F_a ≤ LB_a
7          continue
8      remove node from X_new at position i
9      if g(X_new) < g(X_best)
10         X_best ← X_new
11 return X_best
```

Algorithm 3.2. Remove operator pseudocode.

Figure 3.7. Remove operator example where $i = 2$.

3.3.1.3 Move

Move is an operator that attempts to move all possible solution substrings X_{sub} of length n to all possible locations j and performs the most improving move. If no move improves the fitness $g(X_{\text{best}})$, the original solution is returned. Pseudocode is shown in the Algorithm 3.3. On lines 4-6, a substring X_{sub} is selected from X and X_{new} is created that is identical to X except for the X_{sub} which is missing. On lines 7-9, X_{sub} is inserted to all possible destination locations j. An example move operation, where $i = 3$ and $j = 1$, is illustrated in Figure 3.8. The time complexity of the move operator is $O((m-n)^3)$, where m is the current solution length and n is the length of the substring. The proposed solver also offers a second version of the Move operator that is called Reverse Move, which not only moves the substring but also reverts it.

```
1   function move(X, n)
2   X_best ← X
3   for i in range from 0 to (m − n)
4       X_reduced ← X
5       X_sub ←substring of X at position i with length n
6       remove X_sub from X_reduced
7       for j in range from 0 to (m − n)
8           X_new ← X_reduced
9           insert X_sub into X_new at position j
10      if g(X_new) < g(X_best)
11          X_best ← X_new
12  return X_best
```

Algorithm 3.3. Move operator pseudocode.

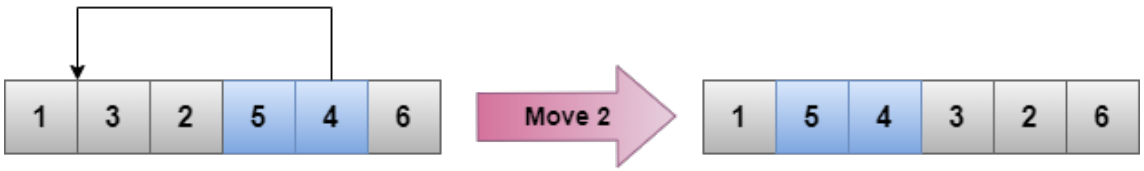

Figure 3.8. Move operator example where $i = 3$ and $j = 1$.

3.3.1.4 Move all

Move all operator moves all occurrences of some node a up to some maximal distance d and chooses the most improving solution X_{best}. Pseudocode is shown in the Algorithm 3.4. Positions p of all the occurrences of a in X are found on line 4. Then, on lines 5-9, for every distance i in the range from $-d$ to d (except 0), the function removes the node a from position p and inserts it back at position $p + i$. An example move all operation, where $a = 3$ and $i = 2$, is illustrated in Figure 3.9. Time complexity of the move all operator is $O(|A|dfm)$, where $|A|$ is the number of nodes, m is the length of the X, and f is frequency of node a.

```
1  function move_all(X, d)
2  X_best ← X
3  for node a in A
4      pos ← find all positions of node a in X
5      for i in [−d, ..., −1, 1, ..., d]
6          X_new = X
7          for p in pos
8              remove node a from X_new at position p
9              insert node a into X_new at position p + i
10         if g(X_new) < g(X_best)
11             X_best ← X_new
12 return X_best
```

Algorithm 3.4. Move All operator pseudocode.

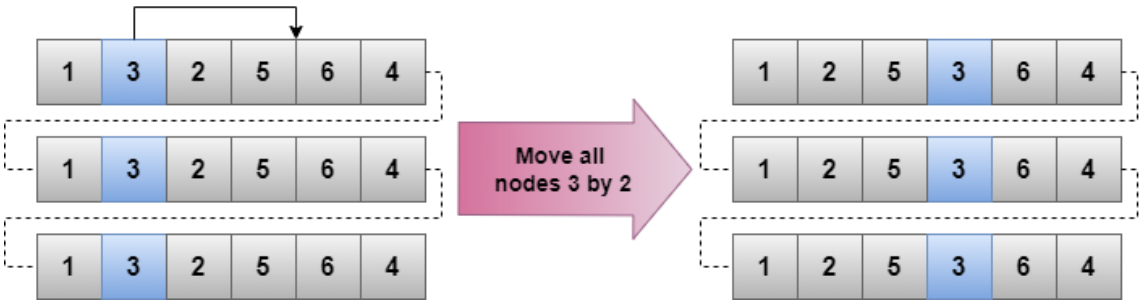

Figure 3.9. Move All operator example where $i = 2$ and $a = 3$.

3.3.1.5 Swap

The Swap operator takes two arguments, n, and p. It tries to swap all possible substrings X_{sub1} and X_{sub2} of length p and q in all possible locations i and j of the X. After iterating through all possibilities it returns the most improving solution X_{best}. Pseudocode is shown in the Algorithm 3.5. On lines 5-6, the algorithm checks that the selected substrings in positions i and j are not overlapping. On lines 8-9, the two substrings are selected, and on lines 10-13, the swap is performed by removing and then reinserting both substrings into the former location of the other substring. An example Swap operation, where $p = 2$, $q = 1$, $i = 3$, and $j = 1$, is illustrated in Figure 3.10. Time complexity of the Swap operator is $O(m(m-p)(m-q))$, where m is the length of the X, and p and q are lengths of the substrings X_{sub1} and X_{sub2} respectively. The proposed solver also offers a second version of the Swap operator that is called Reverse Swap, which not only swaps the two substrings but also reverts them.

```
1  function swap(X, p, q)
2  X_best ← X
3  for i in range from 0 to (m − p)
4      for j in range from 0 to (m − q)
5          if ((i ≥ j and i < j + q) or (j ≥ i and j < i + p))
6              continue
7          X_new ← X
8          X_sub1 ← substring of X at position i with length p
9          X_sub2 ← substring of X at position j with length q
10         remove X_sub1 from X_new at position i
11         insert X_sub2 into X_new at position i
12         remove X_sub2 from X_new at position j
13         insert X_sub1 into X_new at position j
14         if g(X_new) < g(X_best)
15             X_best ← X_new
16 return X_best
```

Algorithm 3.5. Swap operator pseudocode.

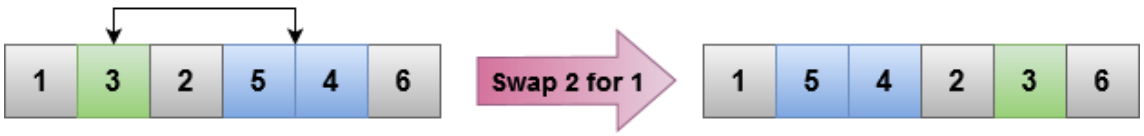

Figure 3.10. Swap operator example where $p = 2$, $q = 1$, $i = 3$, and $j = 1$.

3.3.1.6 Exchange Nodes

Exchange Nodes operator iterates over all possible node pairs (a, b) and exchanges all occurrences of these nodes in X. After iterating through all possibilities it returns the best solution X_{best}. The key part of the Algorithm 3.6 is on lines 6-10 where it iterates over the elements of X and swaps any node a for node b and vice versa. An example Exchange Nodes operation, where $a = 3$ and $b = 5$, is illustrated in Figure 3.11. The time complexity of Exchange Nodes operator is $O(|A|^2 m)$, where m is the length of the X, and $|A|$ is the number of nodes. Note that the values of f_a and f_b also swap.

```
1  function exchange_nodes(X)
2  X_best ← X
3  for node a in A
4      for node b < a in A
5          X_new ← X
6          for i in range from 0 to m
7              if X_i = a
8                  X^i_new = b
9              else if X_i = b
10                 X^i_new = a
11         if g(X_new) < g(X_best)
12             X_best ← X_new
13 return X_best
```

Algorithm 3.6. Exchange Nodes operator pseudocode.

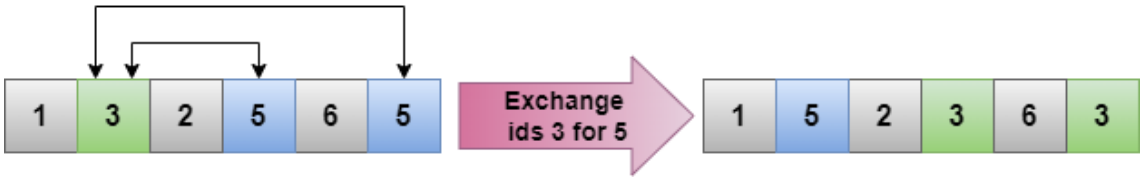

Figure 3.11. Exchange Nodes operator example where $a = 3$ and $b = 5$.

3.3.1.7 Exchange First n Nodes

Exchange First n Nodes operator is a generalization of Exchange Nodes operator. The main difference is that this operator also iterates over the allowed number of exchanges n. Each iteration makes only exchanges of the first n occurrences of nodes a and b, leaving the other occurrences at their original position. Pseudocode is shown in the Algorithm 3.7. On line 7, the counters C_a and C_b are initialized. On lines 11 and 14, the relevant counters are incremented if a is exchanged for b or b is exchanged for a. The conditions on lines 9 and 12 are extended by a condition that the relevant counter does not exceed the value of n. An example Exchange First n Nodes operation, where $a = 5$, $b = 3$, and $n = 2$, is illustrated in Figure 3.12. This generalization comes at the price of increased time complexity. The time complexity of the Exchange First n Nodes operator is $O(|A|^2 mf)$, where m is the length of the X, $|A|$ is the number of nodes, and f is the biggest element of F. Note that F may change if the frequency $f_a > f_b$.

```
1  function exchange_first_n_nodes(X)
2  X_best ← X
3  for node a in A
4      for node b < a in A
5          for n in range from 1 to F_a
6              X_new ← X
7              C_a ← 0, C_b ← 0
8              for i in range from 0 to m
9                  if X_i = a and C_a < n
10                     X^i_new ← b
11                     C_a ← C_a + 1
12                 else if X_i = b and C_b < n
13                     X^i_new ← a
14                     C_b ← C_b + 1
15             if g(X_new) < g(X_best)
16                 X_best ← X_new
17 return X_best
```

Algorithm 3.7. Exchange First n Nodes operator pseudocode.

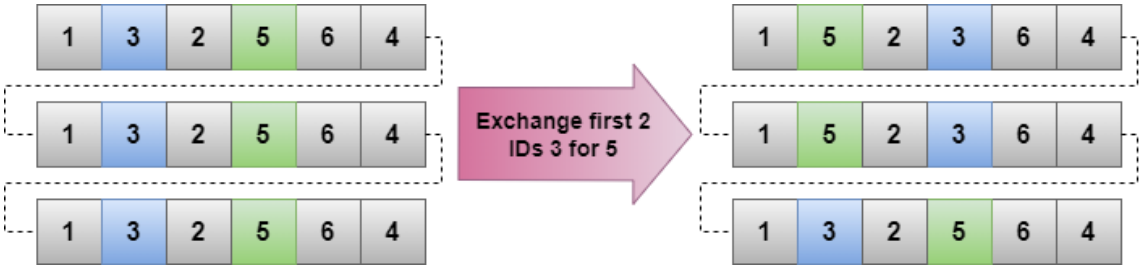

Figure 3.12. Exchange First n Nodes operator example where $a = 5$, $b = 3$, and $n = 2$.

3.3.1.8 Two-opt

The Two-opt operator tries to reverse all possible substrings X_{sub} of the original solution X. If one of the reverses results in improved fitness $g(X_{\text{new}})$, the function returns a new solution X_{new}, otherwise it returns the original solution. Pseudocode is shown in the Algorithm 3.8. On lines 3-4, the boundaries i and j of the substring X_{sub} are selected, and then, on line 6, the substring is reversed. An example Two-opt operation, where $i = 1$ and $j = 4$, is illustrated in Figure 3.13. The time complexity of the Two-opt operator is $O(m^3)$, where m is the length of the X.

```
1  function two_opt (X)
2  X_best ← X
3  for i in range from 0 to (m − 2)
4      for j in range from 2 to m
5          X_new = X
6          reverse X_new between positions i and j
7          if g(X_new) < g(X_best)
8              X_best ← X_new
9  return X_best
```

Algorithm 3.8. Two-opt operator pseudocode.

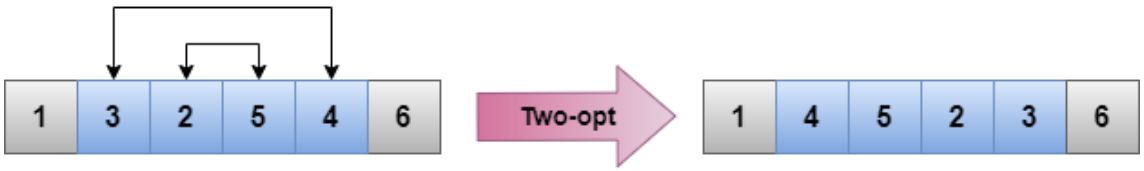

Figure 3.13. Two-opt operator example where $i = 1$ and $j = 4$.

3.3.2 Perturbations

Perturbations are very similar to operators because every perturbation is associated with some neighborhood, but instead of exploring the neighborhood exhaustively, the perturbation returns a random solution from the given neighborhood. All the perturbations also accept a parameter k, which is used to parameterize the perturbation's "strength." A larger parameter k corresponds to a larger neighborhood from which a solution is chosen. This section describes the six perturbations that are implemented in the proposed solver. Namely, Double Bridge, Reinsert, Random Swap, Random Move, and Random Move All.

3.3.2.1 Double Bridge

Double Bridge perturbation splits the original solution X into k substrings and reverts them. Pseudocode is shown in the Algorithm 3.9. On lines 2-4, the algorithm generates k random indices plus the first and the last index of X and stores the indices in a sorted array I. Then on lines 5-6, the function reverses the $k + 1$ substrings bounded by the indices. An example Double Bridge perturbation, where $k = 2$ and $I = \{2, 5\}$, is illustrated in Figure 3.14. The proposed solver also offers a second version of Double Bridge called Random-Reverse Double Bridge, which randomly reverses only some of the substrings with probability $p = 0.5$.

```
1  function double_bridge (X, k)
2  I ← generate a set of k random indices
3  I ← I + {0, m − 1}
4  sort I in ascending order
5  for i in |I|
6      reverse X between positions I_i and I_{i+1}
7  return X
```

Algorithm 3.9. Double Bridge perturbation pseudocode.

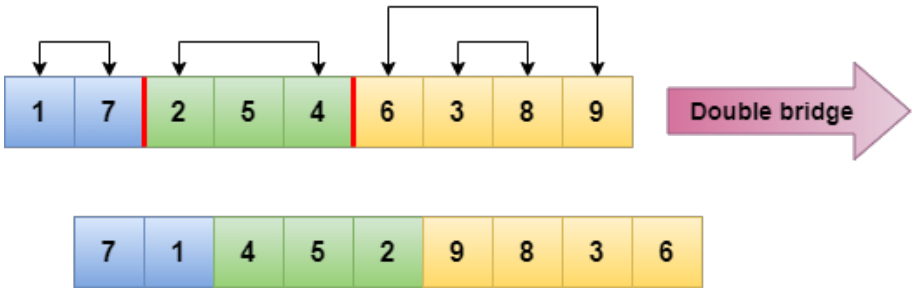

Figure 3.14. Double Bridge perturbation example where $k = 2$ and $I = \{2, 5\}$.

3.3.2.2 Reinsert

Reinsert randomly selects k nodes, deletes all of their occurrences in X, and then reinserts them at random locations. Pseudocode is shown in the Algorithm 3.10. First, k nodes are randomly selected on line 2. Then between lines 4-8, the algorithm iterates over selections of node $a \in I$ and $b \in X$. If $a = b$ the node b is removed from X and the counter C is incremented. Finally, on lines 9-10, the algorithm inserts the node a into random positions of X for every previously removed node. An example Reinsert perturbation, where $k = 1$ and $I = \{3\}$, is illustrated in Figure 3.15.

```
1  function reinsert (X, k)
2  I ← generate k random nodes from A
3  for node a in I
4      C ← 0
5      for node b in X
6          if a = b
7              C ← C + 1
8              remove node b from X
9      for i in range from 0 to C
10         insert node a into X at random position
11 return X
```

Algorithm 3.10. Reinsert perturbation pseudocode.

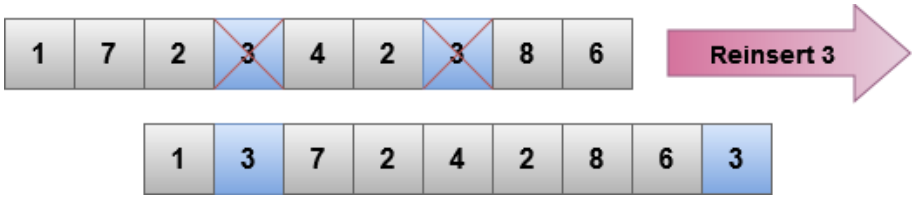

Figure 3.15. Reinsert perturbation example where $k = 1$ and $I = \{3\}$.

3.3.2.3 Random Swap

Random Swap perturbation selects two nodes from X and swaps them. This is repeated k times. Pseudocode is shown in the Algorithm 3.11. An example Random Swap perturbation, where $k = 2$ is illustrated in Figure 3.16. The first iteration swaps nodes at positions $i = 1$ and $j = 5$, and the second iteration swap the nodes at positions $i = 3$ and $j = 6$. Note that if the Swap operator is used during the local search, parameter k should be larger than 1. This should prevent the solver from reverting the perturbation step by a single call of the Swap operator.

```
function random_swap (X, k)
repeat k times
    i ← random position from X
    j ← random position from X
    swap nodes from X at positions i and j
return X
```

Algorithm 3.11. Random Swap perturbation pseudocode.

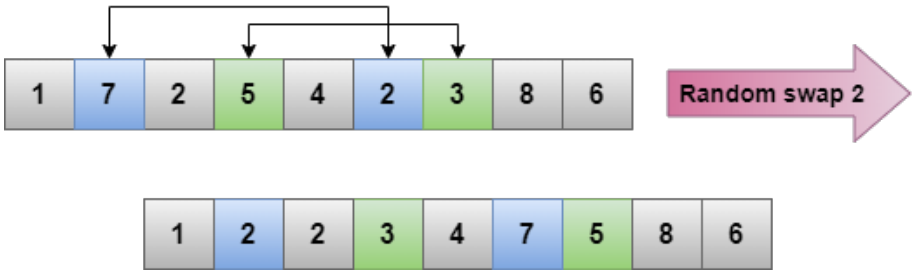

Figure 3.16. Random Swap perturbation example where $k = 2$.

3.3.2.4 Random Move

Random Move perturbation randomly selects k nodes from X and moves them to random locations. Pseudocode is shown in the Algorithm 3.12. An example Random Move perturbation, where $k = 2$ is illustrated in Figure 3.17. In the first iteration, node 7 at positions $i = 1$ is moved to $j = 4$ and in the second iteration, node 8 is moved to $j = 2$. Note that if the Move operator is used during the local search, parameter k should be larger than 1. This should prevent the solver from reverting the perturbation step by a single call of the Move operator.

```
function random_move (X, k)
repeat k times
    i ← random position from X
    remove node a from X at position i
    j ← random position from X
    insert node a into X at position j
return X
```

Algorithm 3.12. Random Move perturbation pseudocode.

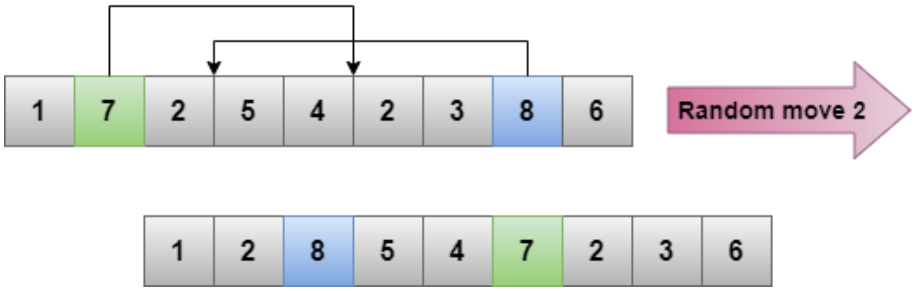

Figure 3.17. Random Move perturbation example where $k = 2$ and $I = \{2, 5\}$.

3.3.2.5 Random Move All

Random Move All is similar to the Move All operator, but it moves every node by a different distance. It moves all occurrences of a node a up to some maximal distance k. Pseudocode is shown in the Algorithm 3.13. On lines 3-4, a random node a is selected, and all occurrences of a in X are found. Then on lines 5-8, every occurrence of a is removed from X and moved by a random distance `dist`. An example Random Move All perturbation, where $k = 1$ is illustrated in Figure 3.18. In this example, the distance `dist` sequentially takes following values: $-1, 0, 1$

```
1  function random_move_all (X, k)
2  repeat k times
3      a ← random node from A
4      P ← find all positions of node a in X
5      for p ∈ P
6          d ← random integer from interval [−k, ..., k]
7          remove node a from X at position p
8          insert node a into X at position p + d
9  return X
```

Algorithm 3.13. Random Move All perturbation pseudocode.

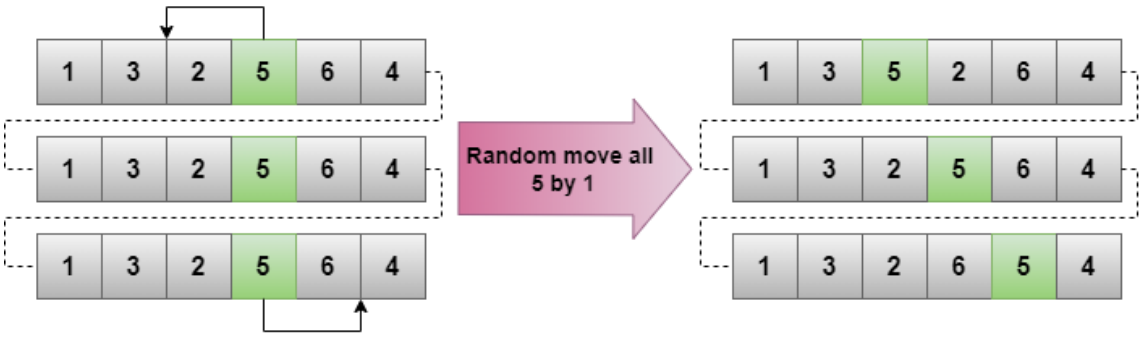

Figure 3.18. Random Move All perturbation example where $k = 2$ and $I = \{2, 5\}$.

3.3.3 Construction strategies

The construction strategies generate an initial solution that is used later as a starting point in all the metaheuristics. This section describes the three construction strategies implemented in the proposed solver. They are greedy, random, and random-replicate. All of them are guaranteed to return a valid solution. A valid solution is a solution

that is not necessarily feasible with respect to the optimized problem definition, but it satisfies the lower and upper bounds (Equations (3.8)).

3.3.3.1 Greedy

The greedy construction strategy repeatedly calls the insert operator until a valid solution is created. If the user wants to use the greedy construction strategy, it is necessary to deal with invalid solutions in the fitness function g. This means that the fitness $g(X)$ should decrease if the frequencies F_{new} of the new solution X_{new} are closer to the LB or UB than frequencies F of the original solution X. This can be expressed as:

$$h(f, lb, ub) = \begin{cases} lb - f & \text{for } f < lb, \\ 0 & \text{for } lb \leq f \leq ub, \\ f - ub & \text{for } f > ub \end{cases}$$

$$g(X_{new}) < g(X) \quad iff \quad \sum_{i=1}^{n} h(F_i^{new}, LB_i, UB_i) < \sum_{i=1}^{n} h(F_i, LB_i, UB_i) \tag{3.25}$$

3.3.3.2 Random

This is the most straightforward construction strategy of the three. It inserts nodes, one by one, to random locations until all lower bounds are satisfied. So the returned solution always has $F = LB$. If the problem has $LB \neq UB$, it is recommended to use insert and remove operators later during the local search. The random construction strategy addresses problems where greedy construction isn't efficient in terms of a trade-off between computation time and fitness improvement per iteration.

3.3.3.3 Random-replicate

This construction strategy generates a random permutation of all nodes, and then it repeatedly appends its copies to the initial solution until all lower bounds are satisfied. In one of the node frequencies reaches the upper bound before all the other nodes reach the lower bound, it is not included in the replica, so the result of this construction strategy is always a valid solution. The random-replicate strategy is proposed in this thesis to address problems represented as several permutations that are similar to each other. Non-Permutation Flowshop is an example of such a problem.

3.3.4 Local search strategies

Local search strategies are functions that choose the order in which the operators are invoked and after reaching a local optimum they return the acquired solution. All the implemented local search strategies are variations of the Variable Neighborhood Descent (VND) [57]. This section describes the five local search strategies implemented in the proposed solver: The Basic VND, Pipe VND, Cyclic VND, Random VND, and Random Pipe VND. All the algorithms expect a list of available operators $\mathcal{O}$.

3.3.4.1 Basic Variable Neighborhood Descent

BVND iterates through the list $\mathcal{O}$ sequentially. If the fitness does not improve after the selected operator O_i, it selects the following operator O_{i+1} and tries again. If the fitness $g(X)$ improves after the call of the selected operator O_i, it returns to the first operator O_1 in the list. Once it reaches the end of the list without any improvement, it returns the solution X. The pseudocode is shown in the Algorithm 3.14.

```
1   function BVND(X, 𝒪)
2   do
3       X_prev ← X
4       for O ∈ 𝒪
5           X_new ← O(X)
6           if g(X_new) < g(X)
7               X ← X_new
8               break
9   while g(X) < g(X_prev)
10  return X
```

Algorithm 3.14. Basic Variable Neighborhood Descent pseudocode.

3.3.4.2 Pipe Variable Neighborhood Descent

PVND iterates through the list $\mathcal{O}$ sequentially as well. Also, same as BVND, if the fitness $g(X)$ does not improve after the selected operator O_i, it selects the following operator O_{i+1} and tries again. The main difference is that if the fitness $g(X)$ improves after the call of the selected operator O_i, the algorithm calls the same operator repeatedly as long as the operator improves the solution X. If PVND reaches the end of the list without any improvement, it returns the solution X. The pseudocode is shown in the Algorithm 3.15. Note that the operation list $\mathcal{O}$ can be large, and therefore on lines 6-7, the algorithm checks whether it reached the last improving operator. If that is the case, it stops, so it does not have to go through the rest of the list.

```
1   function PVND(X, 𝒪)
2   do
3       X_prev ← X
4       O_last ← ∅
5       for O ∈ 𝒪
6           if O = O_last
7               return X
8           X_new ← O(X)
9           while g(X_new) < g(X)
10              X ← X_new
11              X_new ← O(X)
12              O_last ← O
13  while g(X) < g(X_prev)
14  return X
```

Algorithm 3.15. Pipe Variable Neighborhood Descent pseudocode.

3.3.4.3 Cyclic Variable Neighborhood Descent

CVND also iterates through the list $\mathcal{O}$ sequentially. The difference between PVND and CVND is that regardless of whether the fitness $g(X)$ improves or not after the call of the selected operator O_i, CVND moves to another operator O_{i+1}. CVND remembers the last improving operator O_{last} and if $O_{i+1} = O_{\text{last}}$, it stops and returns the solution X. The pseudocode is shown in Algorithm 3.16. Note that on lines 6-7, the algorithm checks whether it reached the last improving operator, and if that is the case, it stops, so it does not have to go through the rest of the list $\mathcal{O}$.

```
function CVND(X, 𝒪)
do
    X_prev ← X
    O_last ← ∅
    for O ∈ 𝒪
        if O = O_last
            return X
        X_new ← O(X)
        if g(X_new) < g(X)
            X ← X_new
            O_last ← O
while g(X) < g(X_prev)
return X
```

Algorithm 3.16. Cyclic Variable Neighborhood Descent pseudocode.

3.3.4.4 Random Variable Neighborhood Descent

RVND iterates through the list $\mathcal{O}$ in random order. Regardless of whether the fitness $g(X)$ improves or not after the call of the selected operator O_i, RVND moves to another operator O_{i+1}. If at least one operator improves fitness $g(X)$, the list of operators $\mathcal{O}$ is again randomly reordered, and all of the operators are called in the new order. The algorithm ends when none of the operators improved fitness. The pseudocode is shown in the Algorithm 3.17. This strategy can be beneficial if the optimization is often getting stuck in the same local optimum as it adds extra stochasticity to the optimization.

```
function RVND(X, 𝒪)
do
    X_prev ← X
    shuffle the list 𝒪
    for O ∈ 𝒪
        X_new ← O(X)
        if g(X_new) < g(X)
            X ← X_new
while g(X) < g(X_prev)
return X
```

Algorithm 3.17. Random Variable Neighborhood Descent pseudocode.

3.3.4.5 Random Pipe Variable Neighborhood Descent

RPVND, as the name suggests, is a combination of RVND and PVND. It goes through the operation list $\mathcal{O}$ in random order, but if an improving operator is found, the algorithm repeats the operator call until a non-improving call occurs. The algorithm ends when none of the operators improved fitness. The pseudocode is shown in the Algorithm 3.18.

```
1  function RPVND(X, 𝒪)
2  do
3      X_prev ← X
4      shuffle the list 𝒪
5      for O ∈ 𝒪
6          do
7              X_new ← O(X)
8          while g(X_new) < g(X)
9          X ← X_new
10 while g(X) < g(X_prev)
11 return X
```

Algorithm 3.18. Random Pipe Variable Neighborhood Descent pseudocode.

3.3.5 Metaheuristics

The metaheuristics are the high-level strategies that use the previously described algorithms. They systematically balance between exploitation and exploration optimization components. Exploitation means to focus on the search in a local region by exploiting the information that a current good solution is found in this region, while exploration means to generate diverse solutions to explore the search space globally ??.

The metaheuristics implemented in the proposed solver work on a similar basis. First step is to generate initial solution X using one of the construction strategies C from Section 3.3.3. Then the metaheuristics repeatedly call one of the local search strategies L from Section 3.3.4 and perturbation P from Section 3.3.2. The difference between particular metaheuristics is in the way the perturbations are handled.

3.3.5.1 Iterated Local Search

ILS is the simplest out of the three metaheuristics. The pseudocode in Algorithm 3.19 shows that the perturbation P is called with a constant parameter k. Also, note that the perturbation P on line 8 is called on the best-known solution X_{best} instead of the last acquired solution X.

```
1  function ILS(C, L, 𝒪, P, k)
2  X ← C()
3  X_best ← L(X, 𝒪)
4  do
5      X ← P(X_best, k)
6      X ← L(X, 𝒪)
7      if g(X) < g(X_best)
8          X_best ← X
9  while timeout not reached
10 return X_best
```

Algorithm 3.19. Iterated Local Search pseudocode.

3.3.5.2 Basic Variable Neighborhood Search

BVNS is an extension to the ILS where the perturbation parameter k is a variable. The pseudocode of BVNS is given in Algorithm 3.20. The function expects two extra parameters k_{min} and k_{max}. The algorithm starts with k set to k_{min}, but it is incremented by one up to the maximal value of k_{max} after every local search that does not improve

the best-known solution X_{best} . On the other hand, if the local search finds a new best-known solution, the parameter k is again assigned the value of $k_{\min}$.

```
1  function BVNS(C, L, 𝒪, P, k_min, k_max)
2  k ← k_min
3  X ← C()
4  X_best ← L(X, 𝒪)
5  do
6      X ← P(X_best, k)
7      X ← L(X, 𝒪)
8      if g(X) < g(X_best)
9          X_best ← X
10         k ← k_min
11     else if k < k_max
12         k ← k + 1
13 while timeout not reached
14 return X_best
```

Algorithm 3.20. Basic Variable Neighborhood Search pseudocode.

3.3.5.3 Calibrated Variable Neighborhood Search

Sampling from a too large neighborhood in perturbation P may result in long runs of local search algorithms. Therefore it might be desirable to increase the neighborhood size only when the local search algorithms converge to the same optimum as the last iteration. CVNS is an extension to the BVNS where the perturbation parameter k is a variable. The pseudocode of CVNS is shown in Algorithm 3.21. The function expects two extra parameters $k_{\min}$ and $k_{\max}$. The only difference to BVNS is that CVNS extends the condition on line 11 with the expression $X = X_{\text{best}}$.

```
1  function CVNS(C, L, 𝒪, P, k_min, k_max)
2  k ← k_min
3  X ← C()
4  X_best ← L(X, 𝒪)
5  do
6      X ← P(X_best, k)
7      X ← L(X, 𝒪)
8      if g(X) < g(X_best)
9          X_best ← X
10         k ← k_min
11     else if X = X_best and k < k_max
12         k ← k + 1
13 while timeout not reached
14 return X_best
```

Algorithm 3.21. Calibrated Variable Neighborhood Search pseudocode.

3.4 Problem-specific components

This section provides implementation details about the problem-specific solver components used in the benchmarking experiments. Namely, the lower bound vector LB,

the upper bound vector UB, the solution representation, and the fitness function $g(X) = p(X) + \hat{g}(X)$. The function $g(X)$ consists of two parts: The penalty function $p(X)$ and the actual fitness function $\hat{g}(X)$. The purpose of the penalty function is to score infeasibility of solutions X so that the optimizer can make steps from infeasible towards feasible solutions.

Additionally, this section provides MILP formulations used in the Gurobi Optimizer.

3.4.1 Capacitated Vehicle Routing Problem

Lets assume that we are given distance matrix $C = (c_{ij}) \in \mathbb{R}^{n \times n}$, the number of available vehicles k, and the depot node D. A solution of the CVRP is a set of k variations $\phi = \{\varphi_1, \ldots, \varphi_k\}$ of the nodes V. In the proposed solver, however, the solution must be defined in the form of one permutation with repetition. This can be done by forming the solution as $X = [D, \varphi_1, D, \varphi_2, D, \ldots, D, \varphi_k, D]$. The CVRP Instance class is initialized based on the following formulation:

$$A = \{1, \ldots, n\} \tag{3.26}$$

$$D = 1 \tag{3.27}$$

$$LB = [k+1, 1, 1, \ldots, 1] \tag{3.28}$$

$$UB = [k+1, 1, 1, \ldots, 1] \tag{3.29}$$

$$\hat{g}(X) = \sum_{i=1}^{m-1} C(X_i, X_{i+1}), \tag{3.30}$$

$$\begin{aligned} p(X) = M_1 \sum_{a \in A \setminus \{D\}} (1 - f_a) + M_2 [\![X_1 \neq D]\!] + \\ M_2 [\![X_m \neq D]\!] + M_3 |k - (f_D - 1)| \end{aligned} \tag{3.31}$$

Where A is the set of available nodes, X_i is the i-th node in the solution X. Note that node 1 is the depot D, and therefore the first elements[1] of upper and lower bound vectors UB and LB are set to $k+1$. The M_1, M_2, and M_3 are some large constants. As described in Section 3.1.2, all the nodes, except the depot, must be visited exactly once. This is expressed by the sum in the first term in the penalty function. The second and the third term ensure that the solution X starts and ends with the depot node. The last term enforces that a feasible solution has exactly k tours.

3.4.1.1 MILP formulation

For the comparison with the Gurobi Optimizer, the Miller-Tucker-Zemlin formulation was used. They proposed the following MILP[58]:

$$\min \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij}$$

$$\text{s.t.} \sum_{i \in V} x_{ij} = 1 \qquad j \in V \setminus \{D\} \tag{3.32}$$

$$\sum_{j \in V} x_{ij} = 1 \qquad i \in V \setminus \{D\} \tag{3.33}$$

$$\sum_{i \in V} x_{iD} = k \tag{3.34}$$

[1] The elements of vectors LB and UB are indexed by the nodes A

$$\sum_{j \in V} x_{Dj} = k \tag{3.35}$$

$$y_j - y_i + Q(1 - x_{ij}) \geq q_j \qquad i, j \in V \setminus \{D\} \tag{3.36}$$

$$x_{ij} \in \{0, 1\} \; i, j \in V \tag{3.37}$$

$$y_i \in \mathbb{R} \qquad i \in V \tag{3.38}$$

Where x_{ij} is a binary decision variable which represents whether some of the routes goes through the edge (i, j). The real variables y_j represent the sum of the quantity demands q_i of the nodes that precede the node j in the current tour plus the quantity demand q_j of the node j.

Equations (3.32) and (3.33) represent the indegree and outdegree constraints for all the nodes except the depot D. Equations (3.34) and (3.35) represent the indegree and outdegree constraints for the depot node and Equation (3.36) enforces the continuity of the tours while also ensuring that the quantity demands in each route does not exceed the capacity Q.

3.4.2 Quadratic Assignment Problem

In the following text, lets assume we are given flow matrix $F = (f_{ij}) \in \mathbb{R}^{n \times n}$ and distance matrix $D = (d_{ij}) \in \mathbb{R}^{n \times n}$. Solution of the QAP is a permutation φ of the nodes A. This makes formulation in the proposed solver quite straightforward as we can directly map X to φ. Following formulation was used for the benchmarking experiments:

$$A = \{1, \ldots, n\} \tag{3.39}$$

$$LB = [1, 1, 1, \ldots, 1] \tag{3.40}$$

$$UB = [1, 1, 1, \ldots, 1] \tag{3.41}$$

$$\hat{g}(X) = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{X_i X_j} \tag{3.42}$$

$$p(X) = M \sum_{a \in A} (1 - f_a) \tag{3.43}$$

where $LB = UB$ are unit vectors. The only penalization of the infeasible solutions is the number of unused nodes in the solution X multiplied by some large constant M. The fitness function $\hat{g}(X)$ is identical with the definition in section 3.1.3.

3.4.2.1 MILP formulation

The Xia Yuan linearization [59] was used for the experiments with the Gurobi Optimizer. Using the binary decision variables x_{ij} and real variables y_{ij} we obtain following MILP:

$$\min \quad \sum_{i=1}^{n} \sum_{j=1}^{n} (y_{ij} + f_{ii} d_{jj} x_{ij})$$

$$\text{s.t.} \quad y_{ij} \geq l_{ij} x_{ij} \qquad i, j = 1, 2, \ldots, n, \tag{3.44}$$

$$y_{ij} \geq u_{ij} x_{ij} - u_{ij} + \sum_{k \neq i} \sum_{l \neq j} f_{ik} d_{jl} x_{kl} \qquad i, j = 1, 2, \ldots, n, \tag{3.45}$$

$$X = (x_{ij}) \in \Pi_n \qquad i, j \in V \tag{3.46}$$

$$y_{ij} \in \mathbb{R} \qquad i \in V \tag{3.47}$$

where X is a permutation matrix and l_{ij} and u_{ij} are lower and upper bounds defined as:

$$l_{ij} = \min \sum_{k \neq i} \sum_{l \neq j} f_{ik} d_{jl} x_{kl} \tag{3.48}$$

$$u_{ij} = \max \sum_{k \neq i} \sum_{l \neq j} f_{ik} d_{jl} x_{kl} \tag{3.49}$$

For a detailed description of the MILP derivation discuss [59–60].

3.4.3 Non-Permutation Flowshop

Lets assume we are given processing time matrix $P = (p_{ij}) \in \mathbb{R}^{m \times n}$ of the Non-Permutation Flowshop instance with m machines and n jobs. Solution of the NPFS is m permutations π_i of length n. The solution X is a vector $[\pi_1, \pi_2, \ldots, \pi_m]$. Following formulation was used for the benchmarking experiments:

$$A = \{1, \ldots, n\} \tag{3.50}$$

$$LB = [m, m, m, \ldots, m] \tag{3.51}$$

$$UB = [m, m, m, \ldots, m] \tag{3.52}$$

$$\hat{g}(X) = C_{\max} \tag{3.53}$$

$$p(X) = M_1 \sum_{a \in A} (m - f_a) + M_2 \sum_{\pi \in X} \sum_{a \in A} |1 - f_a^{\pi}| \tag{3.54}$$

where $LB = UB$ are vectors with all elements equal to m, $C_{\max}$ is the end time of processing on the last machine, f_a^{π} is the frequency of node a in the permutation π and M_1 and M_2 are some large constants. The first term of the penalty function $p(X)$ penalizes the solution if the frequency of any node a is not within the bounds m. The second term enforces that the solution X consists of permutations, not variations with repetitions, i. e., each machine processes each job exactly once.

For the experiments with Gurobi optimizer, the formulation introduced in the section 3.1.4 was used. The MILP is defined by Equations (3.14) to (3.19).

3.4.4 Sudoku

Lets assume that we are given the initial assignment $\boldsymbol{\mathcal{A}} = \{(p_i, k_i),\ i = 1, \ldots, r\}$. A solution to Sudoku is an assignment of the remaining free fields. The solution is a vector $X \in \boldsymbol{\mathcal{I}}^{n^2 - |\boldsymbol{\mathcal{A}}|}$. In other words, the solution X contains the indices from $\boldsymbol{\mathcal{I}}$ of all the free fields. The following formulation in the proposed solver framework was used for the benchmarking experiments:

$$A = \boldsymbol{\mathcal{I}} = \{1, \ldots, n\} \tag{3.55}$$

$$LB = [n - f_1^{\boldsymbol{\mathcal{A}}}, n - f_2^{\boldsymbol{\mathcal{A}}}, \ldots, n - f_n^{\boldsymbol{\mathcal{A}}}] \tag{3.56}$$

$$UB = [n - f_1^{\boldsymbol{\mathcal{A}}}, n - f_2^{\boldsymbol{\mathcal{A}}}, \ldots, n - f_n^{\boldsymbol{\mathcal{A}}}] \tag{3.57}$$

$$\hat{g}(X) = 0 \tag{3.58}$$

$$p(X) = \sum_{B \in \boldsymbol{\mathcal{B}}} \sum_{i \in \boldsymbol{\mathcal{I}}} |1 - f_i^B| \tag{3.59}$$

$f_i^{\boldsymbol{\mathcal{A}}}$ is the frequency of index i in the initial assignment $\boldsymbol{\mathcal{A}}$. $\boldsymbol{\mathcal{B}}$ is the set of n rows, n columns, and n square submatrices. The frequency f_i^B represents the number of

occurrences of the index i in the block B. Note that Sudoku is not a typical optimization problem but rather a constraint satisfaction problem. Therefore, the fitness function $\hat{g}(X)$ equals zero and the overall fitness $g(X)$ consists solely of the penalty function $p(X)$. This is understandable because we only care about the feasibility of solutions rather than their quality.

3.4.4.1 MILP formulation

For the experiments with Gurobi, the three-index binary variable ILP[53] was used. If we denote x_{ijk} assignment of the index k to the field in row i and column j, we obtain following ILP:

$$
\begin{aligned}
\min \quad & 0 \\
\text{s.t.} \quad & \sum_{k=1}^{n} x_{ijk} = 1 && i,j = 1,2,\ldots,n && (3.60)\\
& \sum_{i=1}^{n} x_{ijk} = 1 && j,k = 1,2,\ldots,n && (3.61)\\
& \sum_{j=1}^{n} x_{ijk} = 1 && i,k = 1,2,\ldots,n && (3.62)\\
& \sum_{j=m(q-1)+1}^{mq} \sum_{i=m(p-1)+1}^{mp} x_{ijk} = 1 && k = 1,2,\ldots,n, \quad p,q = 1,2,\ldots,m && (3.63)\\
& x_{ijk} = 1 && \forall (i,j,k) \in \boldsymbol{\mathcal{A}} && (3.64)\\
& x_{ijk} \in \{0,1\} && \forall (i,j,k) \notin \boldsymbol{\mathcal{A}} && (3.65)
\end{aligned}
$$

Equation (3.60) enforces that every field has to have assigned some value k. Constraints (3.61), (3.62), and (3.63) ensures that each value k is in every row, column, and square submatrix exactly once.

Chapter 4
Results

The proposed solver described in Section 3.2 was implemented in C++ [61] for the purpose of benchmarking experiments. Gurobi is a commercial exact solver which can also be used to find approximate solution as it provides best-so-far solution obtained during the optimization. Therefore, it is reasonable to compare Gurobi to the proposed solver on a limited time span. To compete fairly with the Gurobi optimizer, the proposed solver was parallelized using the OpenMP library. In fact the only components that were parallelized are the operators, so the logic of all algorithms stayed the same but some serial computations were parallelized in order to increase efficiency. Both Gurobi and the proposed solver were configured to run on eight threads.

Ten instances of gradually increasing difficulty were used for each problem benchmarking. There are three small, three medium, and four large instances for every problem in the dataset. The optimization timeout was empirically set to 10 minutes for small instances, 30 minutes for medium instances, and one hour for large instances. More detailed description of the instances can be found in the following sections.

Since the proposed solver is based on stochastic algorithms, the experiments were repeated $R = 50$ times with a random seed for each instance and the final fitness results were then averaged. Another measured metrics are the average time of the last improving step and the histogram of the improving operator calls. The averages were calculated simply as

$$\overline{g(X)} = \frac{1}{R}\sum_{i=1}^{R} g(X_i), \qquad \bar{T} = \frac{1}{R}\sum_{i=1}^{R} T_i, \qquad \bar{O} = \frac{1}{R}\sum_{i=1}^{R} O_i,$$

where $g(X_i)$, T_i, and O_i are the final solution's fitness, the time of last improvement, and the number of improving calls of the operator O produced by the i-th solver run. The $\overline{g(X)}$, $\bar{T}$, and $\bar{O}$ are their averages over all trials. Gurobi Optimizer, on the other hand, is deterministic, and therefore only one optimization run was needed for comparison.

The result tables in the following sections contain columns with percentage difference $\overline{\mathrm{GAP}}_{\mathrm{BKS}}$ of the proposed solver's average fitness w.r.t. best known solution and percentage difference $\mathrm{GAP}_{\mathrm{BKS}}$ of the Gurobi Optimizer solution X_{GRB} w.r.t. best known solution. The values of BKS were taken from the relevant literature. The relative percentage differences are computed as follows:

$$\overline{\mathrm{GAP}}_{\mathrm{BKS}} = 100\left(\frac{\overline{g(X)} - \mathrm{BKS}}{\mathrm{BKS}}\right), \qquad \mathrm{GAP}_{\mathrm{BKS}} = 100\left(\frac{g(X_{\mathrm{GRB}}) - \mathrm{BKS}}{\mathrm{BKS}}\right),$$

Another tracked result reported by Gurobi Optimizer is its internal Gap. It is the percentage difference between Gurobi's lower bound (LB) and the best solution found so far computed as:

$$\mathrm{GAP}_{\mathrm{LB}} = 100\left(\frac{g(X_{\mathrm{GRB}}) - \mathrm{LB}}{g(X_{\mathrm{GRB}})}\right)$$

This value is important because it gives an estimate of how far the Gurobi is from reaching optimum.

Ideally, the configurations used for each problem would be selected by some automated parameter tuning tool like the already mentioned iRace. But that is a rather time consuming task and would notably increase the computation time required for the experiments. Therefore, we decided to choose the configuration in a trial-and-error manner. Each of the following sections contain a list of configurations used for that given problem. The experiments were conducted on machines completely dedicated to these experiments with following equal setups:

- OS: Ubuntu 18.04
- CPU: Intel Core i7-7700
  - Architecture: x86_64
  - Frequency: 3.6GHz (Turboboost 4.2GHz)
  - Total Cores: 4
  - Total Threads: 8
  - L3 Cache: 8MB
- Memory: 32GB
- Swap memory: 34GB

4.1 Capacitated Vehicle Routing Problem

The CVRP experiments were conducted on instances drawn from two different datasets. The small instances come from the dataset proposed by Philippe Augerat in 1995 [62]. These instances consist of up to one hundred nodes. The medium and large instances were taken from the dataset proposed by Eduardo Uchoa et al. [62–63] which provides problems with up to 1001 nodes. The following solver configuration was used on all CVRP instances:

- Metaheuristic: BVNS ($k_{\min} = 4, k_{\max} = 8$)
- Local search: PVND
- Construction: Greedy
- Perturbation: Double bridge
- Operators: Two-opt, Move(1), Move(2), Move(3), Move(4), Move(5), Swap(1, 1), Swap(1, 2), Swap(2, 2), Swap(2, 3), Swap(2, 4), Swap(3, 3), Swap(3, 4), Swap(4, 4)

The optimization results from the proposed solver and the Gurobi Optimizer are listed in the Table 4.1. Note that the instance names consist of one letter followed by the number of nodes n and tours k. The best known solutions for computation of the percentage difference were taken from CVRPLIB [62]. The results show that the proposed solver is superior to the Gurobi Optimizer in solving the capacitated vehicle routing problems in multiple aspects. The Gurobi Optimizer was able to find feasible solutions for only the three smallest problems and failed to obtain it on the medium and large instances in the given time span. On two out of three small instances the final fitness was more than 10% larger than the optimum. On the other hand, the proposed solver was able to find feasible solutions for all the problems. Moreover, the average fitness $\overline{g(X)}$ of the solutions to the small instances was within 0.2% of the optimal solution and 1.5% of the medium instances optima.

Figure 4.1 displays $\overline{\text{GAP}}_{\text{BKS}}$ over time obtained by the proposed solver on four different problems. It also shows the average time of a last improving local search. It is apparent that on the large instances (X-n561-k42 and X-n1001-k43) the solver was making improving local search steps until the very end of the allocated time and would have probably continued improving the fitness if a bigger timeout was set. On the other

Instance	T[s]	Proposed solver			Gurobi Optimizer		
		$\overline{g(X)}$	$\sigma(g(X))$	$\overline{\text{GAP}}_{\text{BKS}}$	$g(X)$	GAP_{LB}	GAP_{BKS}
*P-n050-k10	600	697.22	1.09	0.18%	715	20.70%	2.73%
*P-n076-k05	600	628.04	0.81	0.17%	714	20.59%	13.88%
*A-n65-k09	600	1176.24	1.60	0.19%	1515	45.74%	29.05%
*X-n148-k46	1800	43662.76	101.23	0.49%	-	-	-
*X-n204-k19	1800	19786.20	58.93	1.13%	-	-	-
*X-n251-k28	1800	39227.28	102.21	1.40%	-	-	-
X-n351-k40	3600	27039.46	142.99	4.42%	-	-	-
X-n561-k42	3600	44117.96	156.54	3.28%	-	-	-
X-n749-k98	3600	81682.94	319.72	5.71%	-	-	-
X-n1001-k43	3600	78878.78	400.11	9.02%	-	-	-

Table 4.1. CVRP optimization results. The rows starting with the symbol * are instances with known optimal solution.

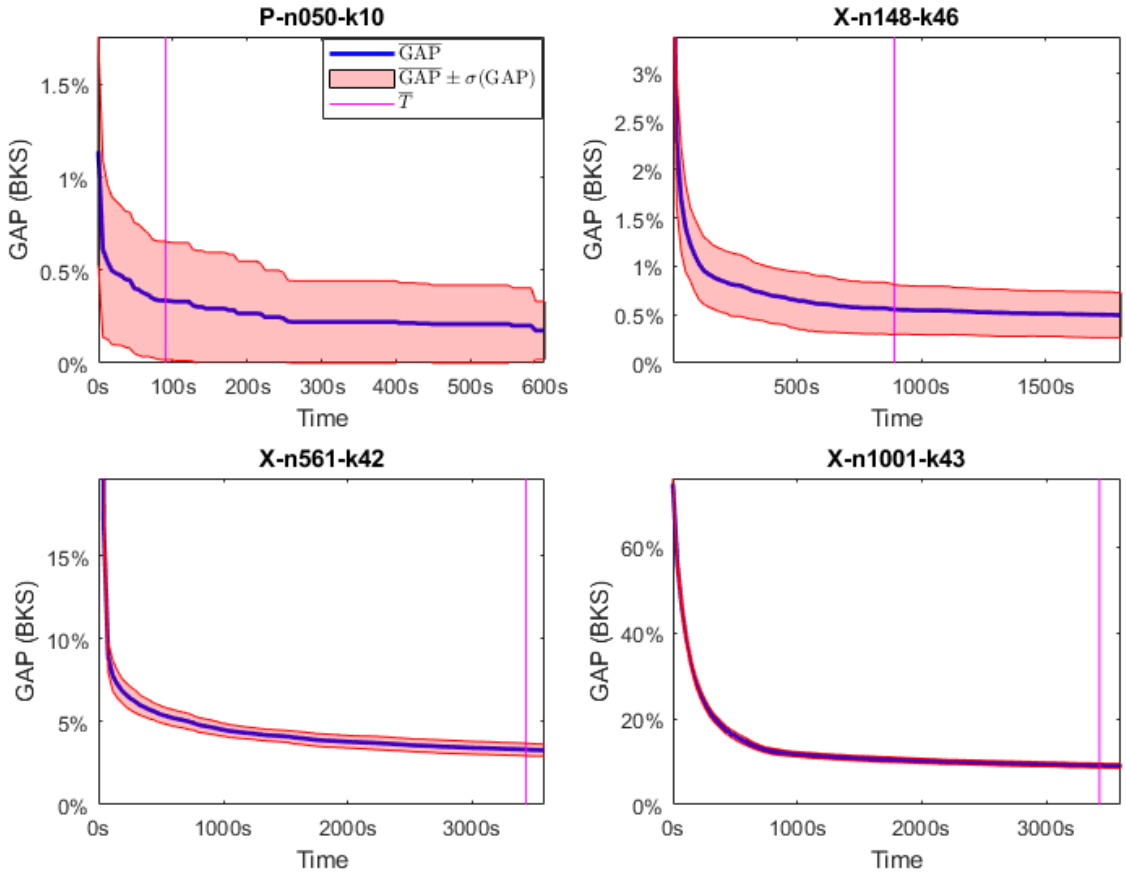

Figure 4.1. CVRP fitness minimization steps of four different instances.

hand, the average of last improvement in X-n148-k46 is suggests that the optimizer was getting stuck before reaching the timeout.

Histogram 4.2 shows that the Two-opt operator was the most improving operator from small to large instances. You can notice a slight increase in the Move(1) and

Swap(1, 1) bins on the instance "X-n1001-k43" but the rest of the operators in both histograms have very similar distribution. Since the Pipe Variable Neighborhood Descent selects the operators sequentially, we cannot simply state that Two-opt is the most effective operator on CVRP because the algorithm always starts with it. To prove this statement, more experiments over more instances and configurations are needed which was not feasible because of the time limitations.

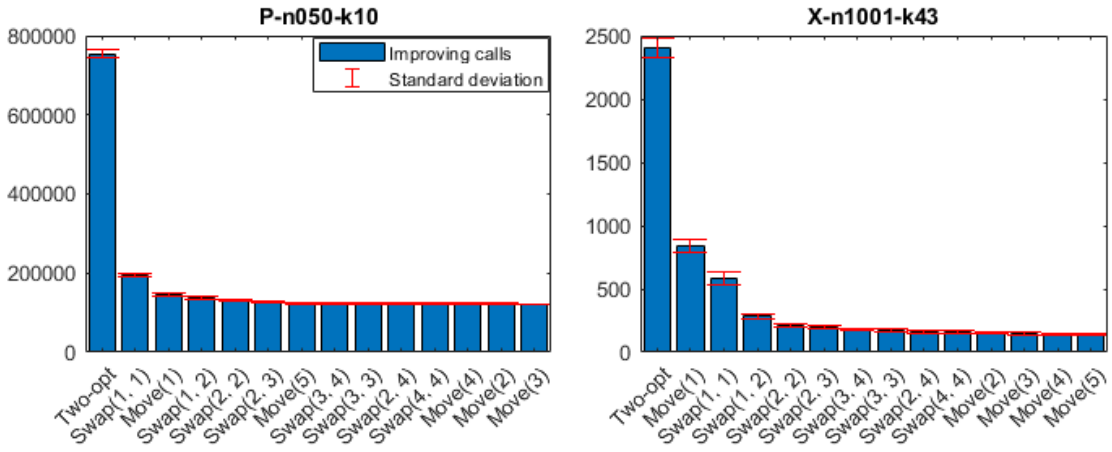

Figure 4.2. CVRP Operators Histogram.

4.2 Quadratic Assignment Problem

The QAP experiments were conducted on instances taken from Taillard's [64] dataset downloaded from QAPLIB [65]. The instances Taixxa have uniformly generated flow and distance matrices while the instances Taixxb are asymmetric and randomly generated. The "xx" in the instance name represents the number of nodes in that instance. The following solver configuration was used on all QAP instances:

- Metaheuristic: CVNS ($k_{\min} = 2, k_{\max} = 5$)
- Local search: RPVND
- Construction: Random
- Perturbation: Random Swap
- Operators: Two-opt, Move(1), Move(2), Reverse Move(2), Move(3), Reverse Move(3), Swap(1, 1), Swap(1, 2), Reverse Swap(1, 2), Swap(2, 2), Reverse Swap(2, 2), Swap(2, 3), Reverse Swap(2, 3), Swap(3, 3), Reverse Swap(3, 3), Swap(3, 4), Reverse Swap(3, 4)

Table 4.2 shows the optimization results from both optimizers. The $\mathrm{GAP}_{\mathrm{BKS}}$ and $\overline{\mathrm{GAP}}_{\mathrm{BKS}}$ values were computed based on best known solutions reported on QAPLIB website [65]. The results show that the proposed solver achieved significantly better results in the given time window on all ten QAP instances. The proposed solver's $\overline{\mathrm{GAP}}_{\mathrm{BKS}}$ is ranging between 0-2% on all the instances and two of the small instances (tai20b and tai30b) were solved optimally in every run. Interesting observation is that when two instances had the same number of nodes but one was symmetrical and the other was asymmetrical, the Gurobi Optimizer achieved better result on the symmetrical while the proposed solver performed better on the asymmetrical. On the other hand, there were only two such pairs in the used instances and so more experiments are needed to make such conclusion.

Instance	T[s]	Proposed solver			Gurobi Optimizer		
		$\overline{g(X)}$	$\sigma(g(X))$	$\overline{\text{GAP}}_{\text{BKS}}$	$g(X)$	GAP_{LB}	GAP_{BKS}
*tai20b	600	122455319.00	0.00	0.00%	123009513	65.99%	0.45%
*tai25a	600	1174472.16	3626.37	0.62%	1224160	20.51%	4.88%
*tai30b	600	637117113.00	0.00	0.00%	665957832	93.54%	4.53%
tai40a	1800	3179448.64	9252.10	1.28%	3301646	24.37%	5.17%
tai50b	1800	459578778.16	835883.75	0.17%	493998253	91.56%	7.67%
tai60a	1800	7335806.12	20420.90	1.80%	7612652	27.01%	5.64%
tai80a	3600	13754440.64	31644.24	1.89%	14618165	28.07%	6.39%
tai80b	3600	828040446.16	5216147.84	1.18%	924292399	90.21%	12.94%
tai100a	3600	21427931.28	45379.38	1.78%	23526234	32.73%	11.75%
tai100b	3600	1197988925.88	12875802.95	1.01%	1422944733	87.67%	19.98%

Table 4.2. QAP optimization results. The rows starting with the symbol * are instances with known optimal solution.

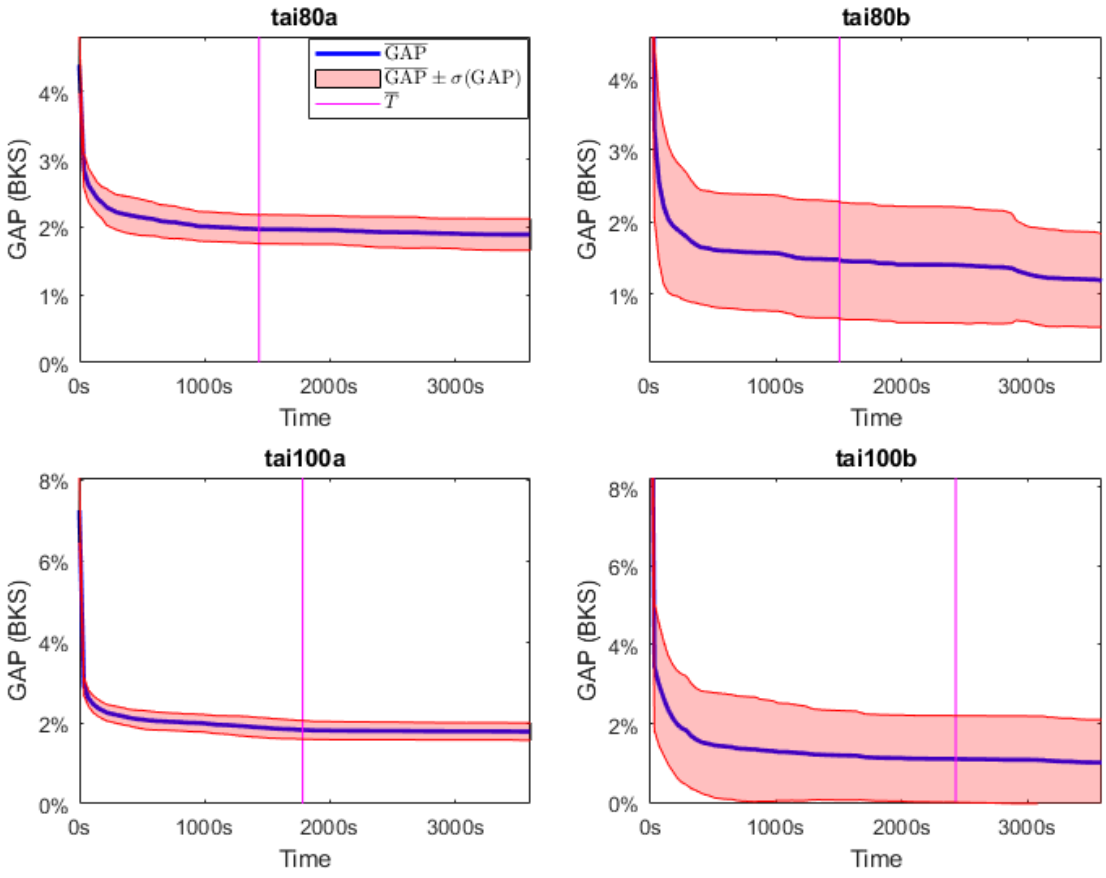

Figure 4.3. QAP fitness minimization steps of four different instances.

Figure 4.3 shows the $\overline{\text{GAP}}_{\text{BKS}}$ over time of two pairs of instances with the same node count. It is interesting that the asymmetrical instances resulted in larger standard deviation $\sigma(\text{GAP})$ than the symmetrical. The average last improvement times $\overline{T}$ suggests that the optimizer could have been getting stuck in some local optima so maybe a more suitable perturbation could be selected for this problem.s

Histogram 4.4 shows that the Swap(1, 1) operator was the most improving operator on two instances; one small and one large. On the larger instance the other operators become less efficient than on the smaller one. The Random Pipe Variable Neighborhood Descent selects the operators randomly so we can state that Swap(1, 1) is the most effective operator on QAP instances.

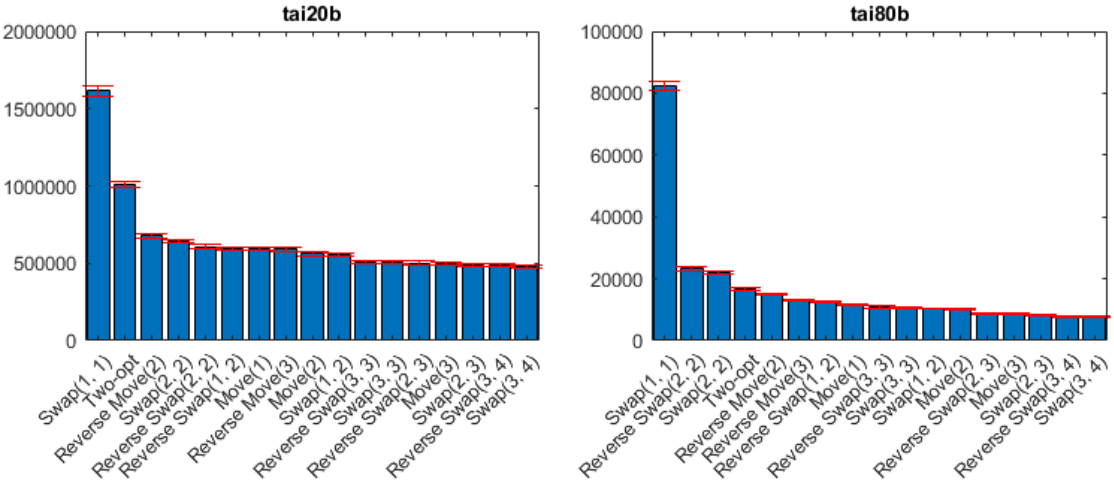

Figure 4.4. QAP Operators Histogram.

4.3 Non-permutation Flowshop

Instances from the data set proposed by Vallada et al. [66] were used on the NPFS experiments. It contains problems ranging from very small instances (10 jobs on 5 machines) to large instances (800 jobs on 60 machines). The data set is originally proposed for permutation flowshop scheduling problem (FSSP), which is almost the same problem as NPFS with the only one difference that each machine must use the same permutation of jobs. The input of FSSP is the same as NPFS, and therefore can be used for NPFS as well. The following solver configuration was used on all NPFS instances:

- Metaheuristic: ILS ($k = 3$)
- Local search: BVND
- Construction: Random Replicate
- Perturbation: Random Move All
- Operators: Exchange Nodes, Move all (10), Exchange First N Nodes

The optimization results of the proposed solver and the Gurobi Optimizer are listed in the Table 4.3. The instance names have following format: “VFRxx_yy_id” where xx is the number of jobs, yy is the number of machines and the id is an integer in range 1-10 as the VFR dataset contains 10 different instances for every job and machine number selection. The best known solutions for computation of the percentage difference were taken from Libralesso’s iterative beam algorithm results [67] which are, to our best knowledge, the best known solutions to FSSP. Even though these values are not NPFS solutions they can be used as upper bound because every feasible solution of FSSP is feasible solution to NPFS as well, that is, the fitness of optimal solution of FSSP is larger or equal to NPFS on the same input. That is why for “VFR20_15_1” the proposed solver was able to find even better solution and has negative $\overline{\text{GAP}}_{\text{BKS}}$.

Instance	T[s]	Proposed solver			Gurobi Optimizer		
		$\overline{g(X)}$	$\sigma(g(X))$	$\overline{\text{GAP}}_{\text{BKS}}$	$g(X)$	GAP_{LB}	GAP_{BKS}
VFR20_5_1	600	1193.72	1.69	0.14	1267	36.11	6.29
VFR20_15_1	600	1916.08	11.85	-1.03	2090	32.49	7.95
VFR40_5_1	600	2396.00	0.00	0.00	2554	43.15	6.59
VFR100_20_1	1800	6363.86	22.20	3.11	7860	55.78	27.35
VFR100_20_2	1800	6466.70	24.27	3.19	7902	73.17	26.09
VFR200_20_1	1800	11577.96	31.60	3.25	13731	32.49	22.46
VFR400_40_1	3600	24253.24	62.60	4.72	-	-	-
VFR400_60_1	3600	26549.72	73.63	4.70	-	-	-
VFR600_40_1	3600	34905.18	95.97	4.55	-	-	-
VFR600_60_1	3600	37573.98	106.20	4.60	-	-	-

Table 4.3. NPFS optimization results.

Table 4.3 shows that the proposed solver performs better on NPFS than the Gurobi Optimizer. The Gurobi Optimizer was able to find feasible solutions for only the small and medium problems and failed to even construct the model for the large instances because it ran out of memory. The proposed solver was able to find better solutions on all of the tested instances.

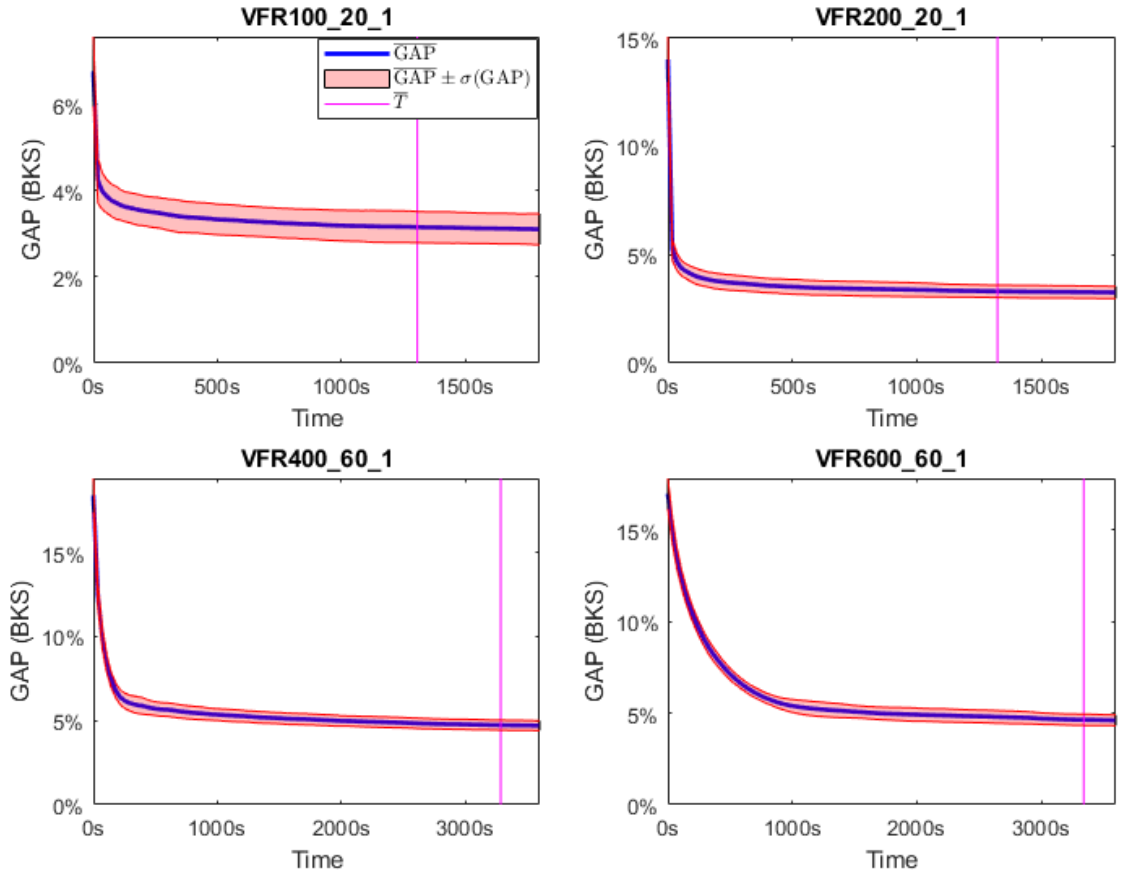

Figure 4.5. NPFS fitness minimization steps of four different instances.

Figure 4.5 displays $\overline{\text{GAP}}_{\text{BKS}}$ over time obtained by the proposed solver on four different problems including the average time of a last improving local search call $\bar{T}$. The displayed plots suggest that the solver was making improving local search steps until the timeout was reached and would have probably continued improving the fitness if a bigger timeout was set even though the slope seems to be very flat when approaching the timeout.

Histogram 4.6 shows that the most improving operator was Exchange Nodes. On the small instances the other operators also played important role while on the large ones they become less significant. The Basic Variable Neighborhood Descent selects the operators sequentially so the dominance of Exchange Nodes operator can be caused just by this. We empirically found out that the other available operators were significantly less efficient on this problem in the terms of time complexity and the number of improvements. Therefore, only these three operators were used for the final experiments.

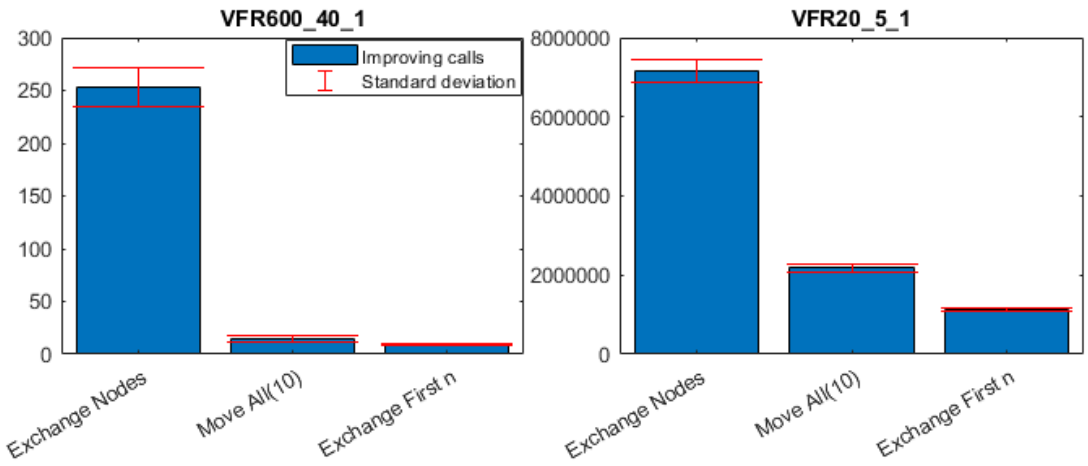

Figure 4.6. NPFS Operators Histogram.

4.4 Sudoku

Sudoku instances used for the experiments were taken from a data set created by Huw Lloyd [68]. There are three types of sudoku instances: 9x9, 16x16, and 25x25. The Names of the instances have following format: "inst$N$xN_p_id" where N is the size of the Sudoku and p is the percentage of the assigned fields. The following solver configuration was used on all Sudoku experiment:

- Metaheuristic: BVNS ($k_{\min} = 2, k_{\max} = 6$)
- Local search: PVND
- Construction: Greedy
- Perturbation: Double bridge
- Operators: Move(1), Two opt, Swap(1, 1)

From the results in Table 4.4 it is apparent that the Gurobi Optimizer performed better on all Sudoku instances. Not only it found the the optimal solution of each instance, it only needed couple of seconds to achieve that. The number of times in which the proposed solver was able to find optimum is denoted |Opt|. This is also equal to the number of times in which a feasible solution was found as the only feasible solution is optimal. The proposed solver was able to find optimum of all 9x9 instances

in all runs. The 16x16 instances were optimally solved in only some cases and 25x25 instances were never solved. The reason for the poor results of the proposed solver might be the fact that heuristic algorithms generally do not aim to find optimal solution. Another interesting observation is that while Gurobi Optimizer needed more time to solve Sudoku with lower assignment percentage p, the proposed solver performed better when p was lower.

		Proposed solver			Gurobi Optimizer		
Instance	T[s]	$\overline{g(X)}$	$\sigma(g(X))$	|Opt|	$g(X)$	Gap	Runtime[s]
inst9x9_5_0	600	0.00	0.00	50	0.00	0.00	0.01
inst9x9_5_1	600	0.00	0.00	50	0.00	0.00	0.01
inst9x9_5_2	600	0.00	0.00	50	0.00	0.00	0.01
inst16x16_5_0	1800	0.40	0.81	40	0.00	0.00	1.51
inst16x16_10_0	1800	0.88	1.29	32	0.00	0.00	1.10
inst16x16_15_1	1800	1.92	1.56	14	0.00	0.00	0.02
inst25x25_5_0	3600	13.92	4.01	0	0.00	0.00	33.25
inst25x25_10_0	3600	16.40	4.49	0	0.00	0.00	15.67
inst25x25_15_0	3600	17.00	3.75	0	0.00	0.00	15.80
inst25x25_20_0	3600	23.28	4.69	0	0.00	0.00	11.07

Table 4.4. Sudoku optimization results.

Figure 4.7 shows fitness $g(X)$ over time obtained by the proposed solver on four different problems. On the 25x25 instances, we can see that the proposed solver was making improving steps until reaching the timeout. On the 16x16 instances, the average last improvement $\bar{T}$ appears to be in the first half of the allocated time. This is probably caused by the fact that some of the experiments reached the optimum prior to the timeout.

Histogram 4.8 shows that the most improving operator was Move(1) but Swap(1, 1) and Two-opt also made significant amount of improving calls. We can see that this distribution doesn't shift between 9x9 and 25x25 instances. This operator histogram is also affected by the fact that the PVND selects operators sequentially. Again, we empirically found out that the other available operators were less efficient and therefore only the three operators were used for the final experiments.

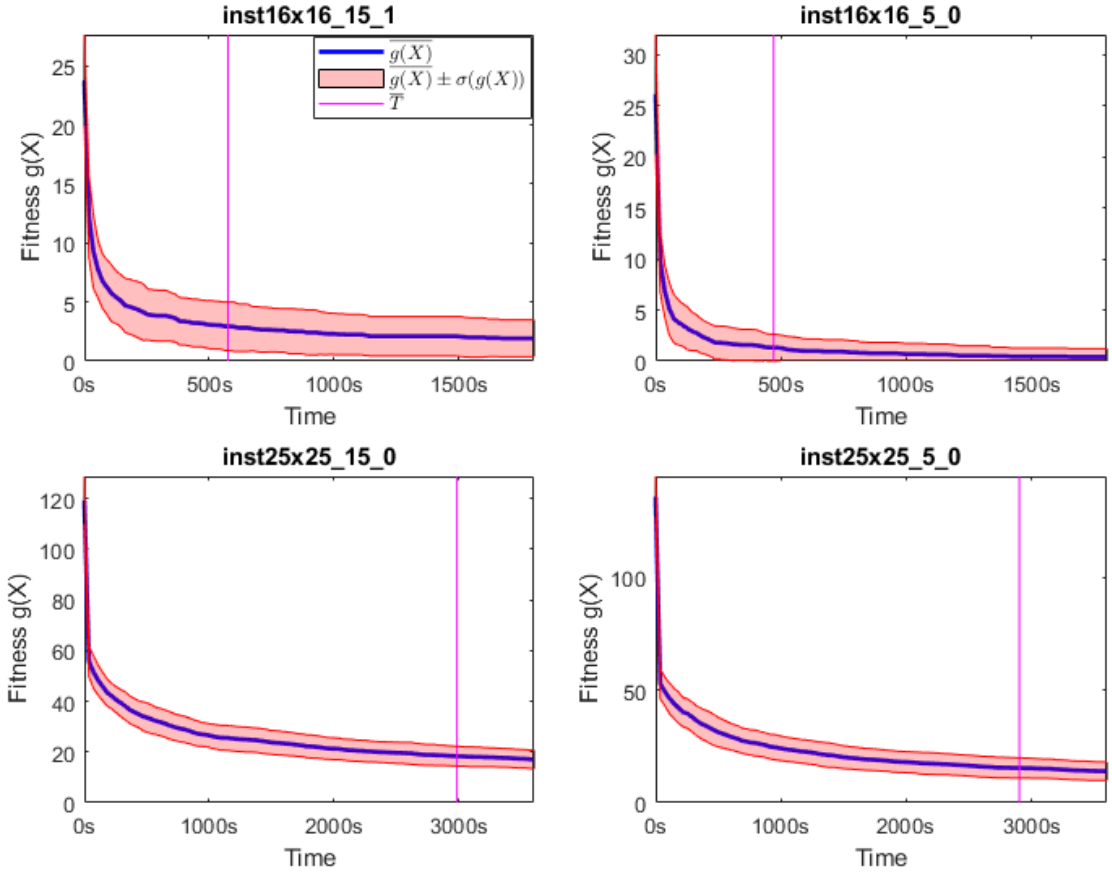

Figure 4.7. Sudoku fitness minimization steps of four different instances.

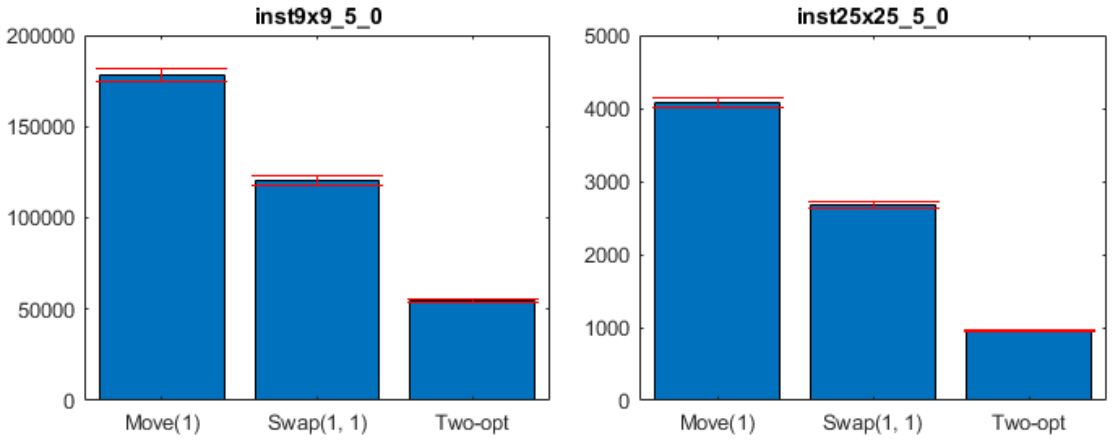

Figure 4.8. Sudoku Operators Histogram.

Chapter 5
Conclusion

General-purpose optimizers are convenient tools that can be applied to wide classes of optimization problems with shared representation. A typical examples are integer programming solvers like Gurobi or constraint satisfaction solvers like CPLEX. We proposed a representation formalism which allows to address a large group of problems whose solutions can be represented using the dynamic permutative representation. Then we designed a neighborhood-oriented heuristic solver that exploits this representation. It adapts multiple well-established metaheuristics that can be used on virtually any combinatorial optimization problem and combines them into one powerful optimizer.

We compared the solver to the commercial IP solver Gurobi Optimizer on four problems: CVRP, QAP, NPFS, and Sudoku. Each problem was formulated using both the proposed solver framework and the integer programming paradigm. The experimental results showed that the proposed solver implemented in C++ [61] outperforms the Gurobi Optimizer on CVRP, QAP, and NPFS problems in terms of scalability and solution quality, given a fixed computational budget. However it did not outperform Gurobi on Sudoku. This might be caused by a poor choice of the proposed solver components (operators, perturbation, etc.) or because heuristic algorithms generally do not excel in finding the optimum solution, and the only feasible solution of Sudoku is the optimum.

Concerning future work, we plan to extend the list of operators and perturbations. For example, NPFS has a very specific structure that was hard to exploit using the currently implemented neighborhoods. Additionally, other metaheuristics like Tabu Search or Simulated Annealing could be easily incorporated in the solver and might improve the performance on some problems.

Our solver contains a number of metaheuristics, local search strategies, initial solution construction strategies, perturbations, and operators. It can be assumed that the solver performs best with a different setup for each problem. We selected the configuration manually using our intuition and several small experiments however this could also be done by an automated parameter tuning tool. This would release the user from the responsibility of providing the configuration. On the other hand, such tools are usually very computationally intensive.

We performed experiments with a fixed timeout but Gurobi is exact solver and by definition it will find better or equal solution than the proposed solver given unlimited amount of time. Another interesting question is how much time it takes the Gurobi Optimizer to obtain such solution on the CVRP, QAP, and NPFS problems. Therefore we suggest experiments with longer timeouts should be made.

The proposed solver is primarily designed for COPs and Sudoku experiments showed that the solver might not be suitable for solving CSPs. The tested optimization problems do not have many constraints so it is reasonable to carry out experiments with more constrained optimization problems e.g. Capacitated Vehicle Routing with Time Windows or Flowshop Scheduling with Resources Constraints.

References

[1] Hejiao Huang, Hongwei Du, and Farooq Ahmad. *Job Shop Scheduling with Petri Nets*. 2013.
http://link.springer.com/10.1007/978-1-4419-7997-1_51.

[2] Chrysafis Vogiatzis, and Panos M Pardalos. *Combinatorial Optimization in Transportation and Logistics Networks*. 2013.
http://link.springer.com/10.1007/978-1-4419-7997-1_63.

[3] Congchao Wang, Yizhi Wang, Yinxue Wang, Chiung-Ting Wu, and Guoqiang Yu. *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. 2019.
http://papers.nips.cc/paper/8334-mussp-efficient-min-cost-flow-algorithm-for-multi-object-tracking.

[4] Combinatorial optimization applied to variable scale 2d model matching. *10th International Conference on Pattern Recognition*. 1990, 1 18-23.

[5] Shouhei Hanaoka, Akinobu Shimizu, Mitsutaka Nemoto, Yukihiro Nomura, Soichiro Miki, Takeharu Yoshikawa, Naoto Hayashi, Kuni Ohtomo, and Yoshitaka Masutani. Automatic detection of over 100 anatomical landmarks in medical CT images. *Medical Image Analysis*. 2017, 35 192-214. DOI 10.1016/j.media.2016.04.001.

[6] Vladimir Kolmogorov, and Ramin Zabih. *Multi-camera Scene Reconstruction via Graph Cuts*. 2002-4-29.
http://link.springer.com/10.1007/3-540-47977-5_6.

[7] Dan Gusfield. *Integer Linear Programming in Computational Biology*. 2019.
http://link.springer.com/10.1007/978-3-030-10837-3_15.

[8] *Benchmark TSP Information*.
https://www.math.uwaterloo.ca/tsp/concorde/benchmarks/bench99.html.

[9] Sergio Nesmachnow. An overview of metaheuristics. *International Journal of Metaheuristics*. 2014, 3 (4), 320-347. DOI 10.1504/IJMHEUR.2014.068914.

[10] Fabrizio Grandoni, and Giuseppe F. Italiano. *Algorithms and Constraint Programming*. 2006.
http://link.springer.com/10.1007/11889205_2.

[11] Nicos Christofides. Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem. *Operations Research Forum*. 2022, 3 (1), DOI 10.1007/s43069-021-00101-z.

[12] Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. *Integer Programming*. 1st ed.. Cham: Springer International Publishing, 2014. ISBN 978-3-319-11007-3.
https://link.springer.com/book/10.1007/978-3-319-11008-0.

[13] Nenad Mladenovic, and Pierre Hansen. Variable neighborhood search. *Computers and operations research*. 1997, 24 (11), 1097-1100.

[14] Thomas Stützle, and Rubén Ruiz. *Iterated Local Search*. 2018. http://link.springer.com/10.1007/978-3-319-07124-4_8.

[15] Thomas Bartz-Beielstein, Jürgen Branke, Jörn Mehnen, and Olaf Mersmann. Evolutionary Algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*. 2014, 4 (3), 178-195. DOI 10.1002/widm.1124.

[16] Fred Glover. Tabu Search—Part I. *ORSA Journal on Computing*. 1989, 1 (3), 190-206. DOI 10.1287/ijoc.1.3.190.

[17] Marco Dorigo. Optimization, learning and natural algorithms. *Ph. D. Thesis*. 1992,

[18] Xin-She Yang. *Engineering optimization*. Hoboken: Wiley, [2010].. ISBN 978-0-470-58246-6.

[19] Pradnya A. Vikhar. *Evolutionary algorithms*. In: *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*. IEEE, 2016. 261-265. ISBN 978-1-5090-0467-6. http://ieeexplore.ieee.org/document/7955308/.

[20] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*. 1983-05-13, 220 (4598), 671-680. DOI 10.1126/science.220.4598.671.

[21] Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM (JACM)*. 2004, 51 (3), 385–463.

[22] David R. Morrison, Sheldon H. Jacobson, Jason J. Sauppe, and Edward C. Sewell. Branch-and-bound algorithms. *Discrete Optimization*. 2016, 19 79-102. DOI 10.1016/j.disopt.2016.01.005.

[23] Toshihide Ibaraki. Theoretical comparisons of search strategies in branch-and-bound algorithms. *International Journal of Computer & Information Sciences*. 1976, 5 (4), 315–344.

[24] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*. 2005, 33 (1), 42-54. DOI 10.1016/j.orl.2004.04.002.

[25] Hugues Marchand, Alexander Martin, Robert Weismantel, and Laurence Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*. 2002, 123 (1-3), 397-446. DOI 10.1016/S0166-218X(01)00348-1.

[26] *IBM ILOG CPLEX Optimization Studio*. https://www.ibm.com/products/ilog-cplex-optimization-studio.

[27] *Generic constraint development environment*. https://www.gecode.org/.

[28] Francesca Rossi, Peter van Beek, and Toby Walsh. *Hanbook of Constraint Programming*. 1st ed.. Elsevier, 2006. ISBN 978-0-444-52726-4. https://www.sciencedirect.com/science/bookseries/15746526/2.

[29] Roman Bartak. Constraint programming: What is behind. *Proceedings of CPDC99*. 1999, 1 (1), 7-15.

[30] Francesca Rossi, Peter van Beek, and Toby Walsh. *Chapter 4 Constraint Programming*. 2008. https://linkinghub.elsevier.com/retrieve/pii/S1574652607030040.

[31] Alexander Bockmayr, and John N. Hooker. *Constraint Programming*. 2005. https://linkinghub.elsevier.com/retrieve/pii/S0927050705120106.

[32] Michel Gendreau, and Jean-Yves Potvin. *Handbook of metaheuristics*. 3rd ed.. Montreal, QC, Canada: Springer, 2019. ISBN 978-3-319-91086-4.

[33] Rafael Martí, Panos M. Pardalos, and Mauricio G. C. Resende. *Handbook of Heuristics*. 1st ed. ed.. Gewerbestrasse 11, 6330 Cham, Switzerland: Springer, 2018. ISBN 978-3-319-07123-7.

[34] *LocalSolver*. https://www.localsolver.com/overview.html.

[35] *OptaPlanner*. https://www.optaplanner.org/.

[36] *ECF - Evolutionary Computation Framework*. http://ecf.zemris.fer.hr/.

[37] *DEAP documentation*. https://deap.readthedocs.io/en/master/.

[38] *Evolving Objects*. http://eodev.sourceforge.net/.

[39] *LKH-3*. http://webhotel4.ruc.dk/~keld/research/LKH-3/.

[40] Keld Helsgaun. An effective implementation of the Lin–Kernighan traveling salesman heuristic. *European Journal of Operational Research*. 2000, 126 (1), 106-130. DOI 10.1016/S0377-2217(99)00284-2.

[41] Keld Helsgaun. An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*. 2017, 24-50.

[42] Milos Seda. Mathematical models of flow shop and job shop scheduling problems. *International Journal of Applied Mathematics and Computer Sciences*. 2007, 4 (4), 241-246.

[43] Marino Widmer, and Alain Hertz. A new heuristic method for the flow shop sequencing problem. *European Journal of Operational Research*. 1989, 41 (2), 186-193. DOI 10.1016/0377-2217(89)90383-4.

[44] Charles E. Noon, and James C. Bean. An Efficient Transformation Of The Generalized Traveling Salesman Problem. *INFOR: Information Systems and Operational Research*. 2016-05-25, 31 (1), 39-44. DOI 10.1080/03155986.1993.11732212.

[45] G. B. Dantzig, and J. H. Ramser. The Truck Dispatching Problem. *Management Science*. 1959, 6 (1), 80-91. DOI 10.1287/mnsc.6.1.80.

[46] Gilbert Laporte. The vehicle routing problem. *European Journal of Operational Research*. 1992, 59 (3), 345-358. DOI 10.1016/0377-2217(92)90192-C.

[47] T.K. Ralphs, L. Kopman, W.R. Pulleyblank, and L.E. Trotter. On the capacitated vehicle routing problem. *Mathematical Programming*. 2003-1-1, 94 (2-3), 343-359. DOI 10.1007/s10107-002-0323-0.

[48] Andreas Konstantinidis, and Savvas Pericleous. Adaptive Evolutionary Algorithm for a Multi-Objective VRP. *International Journal on Engineering Intelligent Systems*. 2014, 22 145-162.

[49] Tjalling C. Koopmans, and Martin Beckmann. Assignment Problems and the Location of Economic Activities. *Econometrica*. 1957, 25 (1), 53–76. DOI 10.2307/1907742.

[50] Eliane Maria Loiola, Nair Maria Maia de Abreu, Paulo Oswaldo Boaventura-Netto, Peter Hahn, and Tania Querido. A survey for the quadratic assignment problem. *European Journal of Operational Research*. 2007, 176 (2), 657-690. DOI 10.1016/j.ejor.2005.09.032.

[51] Daniel Alejandro Rossit, Fernando Tohmé, and Mariano Frutos. The Non-Permutation Flow-Shop scheduling problem. *Omega*. 2018, 77 143-153. DOI 10.1016/j.omega.2017.05.010.

[52] Alexander J. Benavides, and Marcus Ritt. Two simple and effective heuristics for minimizing the makespan in non-permutation flow shops. *Computers & Operations Research*. 2016, 66 160-169. DOI 10.1016/j.cor.2015.08.001.

[53] Andrew Barlett, Timothy P. Chartier, Amy Langville, and Timothy D. Rankin. An integer programming model for the sudoku problem. *Journal of Online Mathematics and its Applications*. 2008, 8 (1),

[54] J. Scott Provan. Sudoku. *American Mathematical Monthly*. 2009-10-01, 116 (8), 702-707. DOI 10.4169/193009709X460822.

[55] *Sudoku Generator*.
`https://www.sudokuweb.org/`.

[56] *The irace Package: Iterated Race for Automatic Algorithm Configuration*.
`https://iridia.ulb.ac.be/irace/`.

[57] Abraham Duarte, Jesús Sánchez-Oro, Nenad Mladenović, and Raca Todosijević. *Variable Neighborhood Descent*. 2018.
`http://link.springer.com/10.1007/978-3-319-07124-4_9`.

[58] B. Farhang Moghadam, S. J. Sadjadi, and S. M. Seyedhosseini. Comparing mathematical and heuristic methods for robust vehicle routing problem. *IJRRAS*. 2010, 2 (2), 108-116.

[59] Yong Xia, and Ya-Xiang Yuan. A new linearization method for quadratic assignment problems. *Optimization Methods and Software*. 2006, 21 (5), 805-818. DOI 10.1080/10556780500273077.

[60] Yong Xia. Gilmore-Lawler bound of quadratic assignment problem. *Frontiers of Mathematics in China*. 2008, 3 (1), 109-118. DOI 10.1007/s11464-008-0010-4.

[61] *Metaopt*.
`https://gitlab.com/Dirys/metaopt`.

[62] *CVRPLIB*. 2022.
`http://vrp.atd-lab.inf.puc-rio.br/index.php/en/`.

[63] Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. New benchmark instances for the Capacitated Vehicle Routing Problem. *European Journal of Operational Research*. 2017, 257 (3), 845-858. DOI 10.1016/j.ejor.2016.08.012.

[64] E. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*. 1991, 17 (4-5), 443-455. DOI 10.1016/S0167-8191(05)80147-4.

[65] *QAPLIB-Problem Instances and Solutions*. 2012.
`https://coral.ise.lehigh.edu/data-sets/qaplib/qaplib-problem-instances-and-solutions/`.

[66] Eva Vallada, Rubén Ruiz, and Jose M. Framinan. New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research*. 2015, 240 (3), 666-677. DOI 10.1016/j.ejor.2014.07.033.

[67] Luc Libralesso, Pablo Andres Focke, Aurélien Secardin, and Vincent Jost. Iterative beam search algorithms for the permutation flowshop. *European Journal of Operational Research*. 2022, 301 (1), 217-234. DOI 10.1016/j.ejor.2021.10.015.

[68] Huw Lloyd, and Martyn Amos. Solving Sudoku With Ant Colony Optimization. *IEEE Transactions on Games*. 2020, 12 (3), 302-311. DOI 10.1109/TG.2019.2942773.

Appendix A
Glossary

BKS	Best Known Solution
B&B	Branch-and-Bound
BVND	Basic Variable Neighborhood Descent
BVNS	Basic Variable Neighborhood Search
COP	Combinatorial Optimization Problem
CSOP	Constraint Satisfaction Optimization Problem
CSP	Constraint Satisfaction Problem
CVNS	Calibrated Variable Neighborhood Search
CVRP	Capacitated Vehicle Routing Problem
DEAP	Distributed Evolutionary Algorithms in Python
EA	Evolutionary algorithms
ECF	Evolutionary Computation Framework
EO	Evolving Objects
FSSP	Flow Shop Scheduling problem
GUB	Global upper bound
ILP	Integer Linear Programming
ILS	Iterated Local Search
IP	Integer Programming
IQP	Integer Quadratic Programming
LB	Lower bound
LKH	Lin-Kernighan-Helsgaun
MILP	Mixed Integer Linear Programming
NPFS	Non-Permutation Flow Shop
PVND	Pipe Variable Neighborhood Descent
QAP	Quadratic Assignment Problem
RPVND	Random Pipe Variable Neighborhood Descent
RVND	Random Variable Neighborhood Descent
TSP	Traveling Salesman Problem
UB	Upper bound
VRP	Vehicle Routing Problem