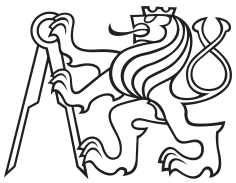
Bachelor Project



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Evolutionary Algorithms for Optimization Problems with Permutative Representation

David Pažout

Supervisor: Ing. David Woller
Study program: Open Informatics
Specialisation: Artificial Intelligence and Computer Science
May 2023

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Pažout David** Personal ID number: **499201**

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

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Evolutionary Algorithms for Optimization Problems with Permutative Representation

Bachelor’s thesis title in Czech:

Evoluční algoritmy pro optimalizační problémy s permutativní reprezentací

Guidelines:

1. Get familiar with metaheuristic optimization algorithms, especially population-based. Research state of the art metaheuristic algorithms for specific problems with solutions representable by permutation or similar sequenced structures.
2. Identify low-level components (parent and survival selection mechanisms, crossover and mutation operators..) and high-level evolutionary algorithm(s) suitable for the addressed class of problems. Implement them in C++ and incorporate them in the generic metaheuristic solver permutator.
3. Apply the implemented algorithms to at least two different problems selected by the supervisor. Benchmark the generic solver against existing problem-specific algorithms.

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] David Woller, Jan Hrazdíra, Miroslav Kulich. Metaheuristic Solver for Problems with Permutative Representation, Intelligent Computing & Optimization 2022, Lecture Notes in Networks and Systems, vol 569. Springer

Name and workplace of bachelor’s thesis supervisor:

Ing. David Woller Intelligent and Mobile Robotics CIIRC

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

Date of bachelor’s thesis assignment: **03.02.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

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 bachelor’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 bachelor’s thesis, the author must state the names of consultants and include a list of references.

. | Date of assignment receipt | Student’s signature

Acknowledgements

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

Declaration

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 26. May 2023

Abstract

In this work, we extend a general-purpose metaheuristic solver for combinatorial problems whose solutions can be represented as an ordered sequence of potentially recurring nodes with arbitrary lengths. We implemented and modified an adaptive penalty evolutionary algorithm ASCHEA and proposed new crossover operators for use with this specific problem representation. We tested the new solver on two real-world problems and compared results with the general-purpose metaheuristic solver and problem-specific solvers.

Keywords: combinatorial optimization; metaheuristics; general-purpose optimizer; evolutionary algorithms; ASCHEA

Supervisor: Ing. David Woller
CIIRC
Intelligent and Mobile Robotics Group

Abstrakt

V této práci rozšiřujeme obecný metaheuristický řešič pro kombinatorické problémy, jejichž řešení lze reprezentovat jako uspořádanou sekvenci potenciálně se opakujících uzlů s libovolnou délkou. Implementovali a upravili jsme adaptivní penalizační evoluční algoritmus ASCHEA a navrhli nové crossover operátory pro použití s touto konkrétní reprezentací. Nový řešič jsme testovali na dvou reálných problémech a porovnali výsledky s obecným metaheuristickým řešičem a řešiči navrženými na tyto specifické problémy.

Klíčová slova: kombinatorická optimalizace; metaheuristika; univerzální optimalizátor; evoluční algoritmy; ASCHEA

Contents

Figures

Tables

Chapter 1

Introduction

The ability to solve Combinatorial Optimization Problems (COPs) is important to many disciplines. The complexity of the COPs ranges from polynomial time solvable to NP-hard problems. General-purpose COP solvers are mainly based on Integer Linear Programming (ILP) or Constraint Satisfaction Programming such as Gurobi [gur23] and CPLEX [IBM23],. The downside of these solvers is that they are based on exact methods and as such don't scale well for NP-hard problems. Often the only viable way to solve large instances of NP-hard problems is to use problem-specific solvers whose development is both time and expertise intensive.

A lot of COPs that are commonly used as testing benchmarks, e.g. Traveling Salesman Problem (TSP), Vehicle Routing Problem (VRP), Quadratic Assignment Problem (QAP) and Flow-Shop Scheduling Problem (FSSP), are unconstrained problems with solutions represented as permutations, meaning any permutation is a valid solution to the problem, even though it isn't a particularly good solution. However, many of the real-world COPs are constrained, e.g. Capacitated VRP (CVRP) and other VRP variants, or don't have a permutation as solution representation, e.g. Non-Permutation Flow-Shop problem (NPFS).

In this thesis, we extend the already implemented General Metaheuristic Solver (GMS) [WHK22] for solving 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 sequence may change during the optimization. Unlike the previously mentioned general COP solvers, GMS is a metaheuristic

and as such scales better for NP-hard problems. The first version of GMS used Local Search (LS) methods with variable local search operators. We will call this version of the solver the GMS-LS. It was compared with ILP solver Gurobi on COPs with few to no constraints (CVRP, QAP, NPFS). The GMS-LS outperformed Gurobi in both the quality and scalability of the solutions in the limited time frame given to solve the problems.

We implemented a new search method based on Adaptive Segregational Constrained Handling Evolutionary Algorithm (ASCHEA) and proposed 18 crossover operators for the dynamic permutative representation. This approach, referred to as Constrained Oriented GMS (GMS-CO), was tested on two real-world problems with numerous constraints: Electric VRP (EVRP) [MMT+20] and ROADEF/EURO challenge 2020: maintenance planning problem (ROADEF) [Roa20]. GMS-CO was compared to the previous GMS-LS as well as to problem-specific solvers on both problems.

Chapter 2

Literature overview

Depending on desired solution quality we can choose from three types of solvers: exact, approximate and metaheuristic.

Exact solvers are guaranteed to find an optimal solution to the problem. However, many COPs are NP-hard and as such exact solvers take exponential time to find the optimal solution. Probably the most widely used representation for combinatorial problems is ILP used in solvers like Gurobi and CPLEX, which implement highly optimized versions of simplex and branch-and-bound algorithms for solving ILP problems.

Approximation solvers are polynomial time solvers that are guaranteed to find a solution of a certain quality. Found solutions are usually guaranteed to be within a multiple of the optimal solution. These solvers are however problem dependent and cannot be generalized. For example Christofides algorithm [Chr76], a polynomial time approximation algorithm for TSP in metric space, which is guaranteed to find a solution below 1.5 multiple of the optimal solution.

Metaheuristic solvers typically do not guarantee anything about the solution quality. In practice, they are often able to find good-quality solutions in a short amount of time. The basic concepts of some common metaheuristics will be introduced in the next section.

2.1 Problem specific metaheuristics

In this section, we go over metaheuristics used by successful competitors of ROADEF and EVRP challenges.

Second place in the ROADEF 2020 challenge was awarded to Hue Chi Lam, Hanyu Gu and Thi Thanh Thu Pham [GLPZ23]. They implemented many algorithms, their most successful was iterated local search (ILS) [LMS03] with self-adaptive perturbation. ILS is an extension of local search. When the local search finds a locally optimal solution a perturbation operator shifts the solution to a different part of the search space where the local search begins again. ILS is highly dependent on the strength of the perturbation operator, if the perturbation is weak it causes the search to fall in the same local optima it has previously found. If the perturbation is strong, it is no different from starting the search from a new, entirely random solution. The self-adaptive perturbation starts as weak and, if no improving solution is found in a certain number of iterations, the strength of the perturbation is increased until an improving solution is found.

Third place was awarded to Francisco Parreño, Ramon Alvarez-Valdes and Consuelo Parreño-Torres [PAVPT]. They first obtain a pool of near-optimal solutions using ILP on a simplified objective function. Found solutions are then improved by Variable Neighborhood Descent (VND) [MH97] and the space near improved solutions is further explored by Path Relinking [Glo97]. VND uses several ranked local search operators, each of which defines its own neighborhood. Starting from an initial solution, VND finds a locally optimal solution with regard to the first operator. VND then iteratively searches the neighborhood of the found solution with the next operator until all operators were exhausted. If an improving solution was found VND start the cycle again from the first operator. If no improving solution was found VND terminates.
Path Relinking takes two good-quality solutions and finds a path of solutions between them by applying a small perturbation operator. If a solution on the path is better than both starting solutions, one of them is replaced.

The winners of the ROADEF 2020 challenge, Mirsad Buljubasic and Michel Vasquez, have not yet published a paper discussing their solution. From their code [BV23], it seems they used a starting population of feasible solutions initialized by ILP that were further improved by VND.

The winners of the IEEE WCCI-2020 Competition on Evolutionary Computation for the Electric Vehicle Routing Problem (EVRP) were David Woller,

Václav Vávra and Viktor Kozák [KWV21]. They used Variable Neighborhood Search (VNS) [RHHM02] initialized by custom solution construction methods. VNS is an extension of ILS which uses a combination of different perturbation and local search operators.

The silver medalists of the EVRP 2020 Competition, V. Mak-Hau and B. Hill, have not yet published a paper discussing their solution. However, the problem website states that they used Simulated Annealing (SA) [VLAvLA87]. SA is a local search procedure that mimics a physical process of cooling hot metals. Local search starts with high temperature, meaning it has a higher probability of accepting a worse solution. As the temperature cools down during the search, the probability of accepting worse solutions decreases toward zero.

Third place was awarded to Vu Quoc Hien, Tran Cong Dao and Huynh Thi Thanh Binh [HDB23]. They used an Evolutionary Algorithm (EA) [BS96] initialized by locally optimal solutions. EAs create a population of potential solutions to the problem and then use operators inspired by natural selection and evolution processes to evolve a better population over time.

2.2 Constraint handling metaheuristics

The aforementioned solvers use problem-specific operators and repair functions to satisfy all problem constraints. Generic metaheuristic solvers must find a feasible solution without inner knowledge of the problem. Several constraint-handling techniques are used in practice, mostly in the population-based solvers and EAs.

The most common, easiest and earliest way to handle constraints in EAs is to use penalties [CC02]. The goal of the penalty is to transform the constrained problem into an unconstrained one by adding penalty $p(X)$ to the infeasible solution's fitness based on the degree of its constraint violation. Some of the commonly used types of penalties are:

- **Death penalty**
  If a solution violates any constraint, it is rejected and generated again. Because we gain no information on constraint violations it is usually limited to problems in which the feasible search space is convex and constitutes a reasonably large portion of the whole search space.

- **Static penalty**
  Penalty coefficients do not depend on the current generation number. There are many ways to configure static penalty, three common ways are:
  - **Number of unsatisfied constraints**
    When an individual is infeasible, their fitness is not computed and all the individuals that violate the same number of constraints receive the same penalty regardless of how close they are to the feasible region [MQ98].
    $$p(X) = K(p - s), \tag{2.1}$$
    where K is a large constant, p is the number of constraints and s is the number of unsatisfied constraints.
  - **Constraint violation**
    A constant coefficient is assigned to each constraint.
    $$p(X) = \sum_{i=1}^{p} r_i v_i(X), \tag{2.2}$$
    where r_i is a penalty coefficient and v_i is a constraint violation function (3.6).
  - **Stepwise constraint violation**
    Penalty coefficient r_i can itself be a function of v_i such that $r_i = r_i(v_i(X))$ is an increasing stepwise function depending on the level of the constraint violation [Mic96].
- **Dynamic penalty**
  Penalty functions in which the current generation number is involved in the computation of the corresponding penalty coefficients. Typically, the penalty coefficients increase over time, pushing the search toward the feasible region. For example [JH94]
  $$p(X) = (Ct)^{\alpha} \sum_{i=1}^{p} v_i(X), \tag{2.3}$$
  where C and α are user-defined constants, t is current generation number and v_i is a constraint violation function.
- **Adaptive penalty**
  The penalty function takes feedback from the search process. For example [BHAB97]
  $$p(X) = r(t) \sum_{i=1}^{p} v_i(X), \tag{2.4}$$
  where penalty coefficient r is updated each generation according to the
  $$r(t+1) = \begin{cases} \frac{r(t)}{\beta_1}, & \text{if } \textbf{case1}, \\ \beta_2 r(t), & \text{if } \textbf{case2}, \\ r(t), & \text{otherwise}, \end{cases} \tag{2.5}$$

where
case1: if the best individuals in the last k generations were always feasible,
case2: if the best individuals in the last k generations were always infeasible,
and $\beta_1, \beta_2, k > 1$ are user defined constants and $\beta_1 \neq \beta_2$ (to avoid cycling).

- **ASCHEA** Adaptive Segregational Constrained Handling Evolutionary Algorithm (ASCHEA) [SH00] [BHS02] is an adaptive penalty evolutionary algorithm for real-space optimization problems. The main idea behind ASCHEA is to maintain both feasible and infeasible solutions in the population and leveraging crossover between the feasible and infeasible individuals, when it is deemed advantageous, to better explore the feasible space and its boundary where the optimum usually lies.

To avoid dealing with penalty coefficients Stochastic ranking (SR) [RY00] separates the fitness function from the fitness violations. SR ranks solutions in the population with a comparison function. The function compares feasible solutions based on their fitness and infeasible solutions based on how many constraints they violate. When comparing feasible and infeasible solutions the feasible solution is ranked better most of the time, but with a small probability they are compared based on their fitness, and thus infeasible solution can rank better than a feasible one. It then selects the best solutions to continue to the next iteration of the search process.

Another approach is to treat each constraint as a separate objective and try to minimize all of the objectives with Multiple Objective EA (MOEA). One of the most popular MOEAs is NSGA-II [DPAM02], which mains a Pareto front [RK14] of solutions, that is a set of solutions that aren't worse in any objective than all other found solutions.

Chapter 3

Theory

3.1 Optimization

Optimization is a collection of mathematical principles and methods used for solving quantitative problems in many disciplines [opt23]. Optimization problems consist of three main parts: set of optimized variables $\mathbb{F}$, fitness function g, which assigns each variable a real number and demand for minimization or maximization of said fitness function [KW22]. This can be formally written as

$$\min_{X \in \mathbb{F}} g(X), \tag{3.1}$$

where $f : \mathbb{V} \to \mathbb{R}$ is a fitness function, $\mathbb{V}$ is domain of g, and $\mathbb{F} \subseteq \mathbb{V}$. Set $\mathbb{V}$ will be called a set of valid solutions, and $\mathbb{F}$ will be a set of feasible solutions. We will focus only on minimization problems since $\max_{X \in \mathbb{F}} g(X) = \min_{X \in \mathbb{F}} -g(X)$.

3.1.1 Constrained Optimization

Constrained optimization is a process of minimizing a fitness function over some variables where the variables must satisfy a set of constraints. Various kinds of constrained problems arise depending on whether the value of the fitness function can be computed even for infeasible individuals, whether the constraints return only a boolean info (fulfilled/unfulfilled, feasible/infeasible), or a more informative degree of constraint violation.

A general constrained minimization problem may be written as follows [MN21].

$$\min_{X \in \mathbb{V}} g(X) \quad \text{s.t.} \tag{3.2}$$

$$u_i(X) \leq 0, \ i = [1, .., m], \tag{3.3}$$

$$v_j(X) = 0, \ j = [m+1, .., p], \tag{3.4}$$

where $u_i(X) \leq 0$ and $v_j(X) = 0$ are set of p constraint functions that must be satisfied. The set of feasible solutions $\mathbb{F}$ is defined as

$$\mathbb{F} = \{x \in \mathbb{V} | u_i(X) \leq 0 \ and \ v_j(X) = 0, \ i = [1, .., m], \ j = [m+1, .., p]\}. \tag{3.5}$$

It is useful to quantify how much the solution x violates constraint i. For this purpose, we use a constraint violation function ν_i, which must satisfy the following equations,

$$\nu_i : \ \mathbb{V} \to \mathbb{R}, \tag{3.6}$$

$$\nu_i(X) \begin{cases} = 0, \ \text{if } X \in \mathbb{F}, \\ > 0, \ \text{otherwise.} \end{cases} \tag{3.7}$$

3.2 Evolutionary Algorithms

Evolutionary algorithms (EAs) are a class of computational methods inspired by the principles of natural selection and evolution [Dar59]. They are used to solve optimization and search problems with large search spaces and complex fitness functions [BS96]. EAs create a population of potential solutions to the problem and then use selection, reproduction (crossover and mutation) and replacement operators to evolve a better population over time.

3.2.1 EA Pipeline

The basic structure of evolutionary algorithms can be described as follows:

- **Initialization** creates a base population of λ solutions and evaluates their fitness. This can be done either by randomly generating valid solutions or using some kind of constructive heuristic.

- **Selection** determines which individuals will go on to reproduce. The purpose of selection is to incentivize the reproduction of stronger individuals. Two of the most popular selection strategies are roulette wheel selection and tournament selection.

- **Crossover** is an analog of sexual reproduction, combining genetic information from parent solutions to hopefully create better offspring. The operator is applied to individuals in the parent population until μ children are produced.

- **Mutation** introduces small genetic variations into the child's genome. The intensity and the occurrence of a mutation are often subject to chance. After this step fitness of all children is computed.

- **Replacement** combines the old base population and the population of children, discarding some individuals in the process. As a general rule replacement should maintain a static population size, although some exceptions to this rule can be found [SWL+03]. The most popular replacement schemes are deterministic, where only the fittest individuals survive, and generational, where the old base population is discarded and replaced by the population of children.

The process of selection, crossover, mutation and replacement is repeated until a stopping criterion is met, e.g. a maximum number of generations has been reached, allocated time has elapsed, or a solution with the desired level of fitness has been found.

3.3 General Metaheuristic Solver (GMS)

This section is an overview of the General Metaheuristic Solver (GMS) as described in Jan Hrazdíra's Master's Thesis [Hra22] on top of which our solver is designed. The GMS consists of the Generic Solver component and the Problem component. Generic Solver is made of three parts: abstract class Instance, and Solution and Generic Optimizer classes. Implementation of our Generic Optimizer class and Problem components will be described in the next chapter.

3.3.1 Problem definition

The GMS accepts minimization problems whose solutions can be encoded as an ordered sequence of potentially recurring nodes with arbitrary lengths. More formally, the problem and its solution must satisfy the following.

$$\text{Given a set of nodes:} \quad A = \{1, ..., n\},\ A \subset \mathbb{Z}_+ \tag{3.8}$$

$$\text{and bounds:} \quad LB = [l_1, l_2, ..., l_n],\ LB \in \mathbb{Z}_+^n, \tag{3.9}$$

$$UB = [u_1, u_2, ..., u_n],\ UB \in \mathbb{Z}_+^n. \tag{3.10}$$

$$\text{Minimize fitness function:} \quad g(X) :\ A^m \to \mathbb{R}, \tag{3.11}$$

$$\text{where:} \quad X = [x_1, x_2, ..., x_m],\ X \in A^m, \tag{3.12}$$

$$F = [f_1, f_2, ..., f_m],\ F \in \mathbb{Z}_+^n, \tag{3.13}$$

$$f_i = \sum_{j=1}^{n} [\![x_k = a_i]\!],\ \forall i \in [1, n], \tag{3.14}$$

$$l_i \leq f_i \leq u_i,\ \forall i \in [1, n]. \tag{3.15}$$

In other words, given a set of nodes A with two vectors of lower bound LB and upper bounds UB, 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 can be an arbitrary function $A^m \to \mathbb{R}$. Solution X can also have variable length when $LB \neq UB$. In the context of the GMS, the solution X is considered valid, i.e. $X \in \mathbb{V}$, if all its nodes lie inside the lower and upped bounds. The feasibility of solution X, i.e. $X \in \mathbb{F}$, is determined by the fitness function g.

3.3.2 Framework

Generic Solver is made of three parts: abstract class Instance, and Solution and Generic Optimizer classes.

The Instance is an abstract class that defines mandatory properties that every Problem instance must have defined in order to follow our formal problem definition. Those are fitness function g, that returns the fitness and information on the feasibility of X, i.e. if $X \in \mathbb{F}$ or $X \notin \mathbb{F}$. Other properties are the number of nodes A and the lower and upper bounds, LB and 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 Problem class, which inherits from this one. It has to set all the mandatory properties and implement a corresponding fitness function g.

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.

Generic Optimizer 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. For the optimizer to function properly together with the Instance class it should only compute the fitness g of valid solutions $X \in \mathbb{V}$.

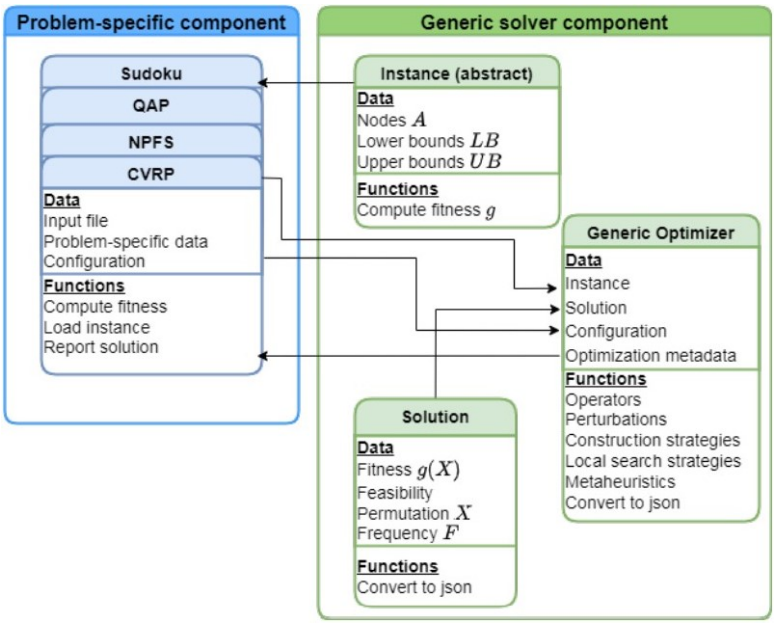


Figure 3.1: Solver class diagram

3.4 Implemented problems

We chose to tackle two real-world problems with numerous constraints that already have their own problem-specific solvers whose performance can be used as a benchmark. First is the Electric Vehicle Routing Problem (EVRP) [MMT+20] a version of the Vehicle Routing Problem (VRP). The second

problem is a subject of the ROADEF 2020 Challenge [Roa20] and concerns grid operation-based outage maintenance planning, a type of Transmission Maintenance Scheduling problem [FGM+16].

3.4.1 Electric Vehicle Routing Problem (EVRP)

EVRP is an extension of Capacitated VRP (CVRP), which itself is an extension of the VRP. In the VRP, the goal is to minimize the total distance traveled by a fleet of vehicles, while visiting each customer exactly once. In the CVRP, the customers are assigned an integer-valued positive demand and the vehicles have limited carrying capacity. Thus an additional constraint of satisfying all customers' demands while respecting the limited vehicle capacity is added to the VRP. In the EVRP vehicles have a charge capacity that is depleted with the distance traveled, and vehicles can be recharged at the depot or charging stations. Thus an additional constraint of not running out of charge while satisfying all other constraints of CVRP is added.

The next section is a slightly reformulated problem definition and description as given in [MMT+20]. Given a fleet of EVs, a set of customers and charging stations, and one depot, the goal is to minimize the total distance traveled by the fleet while the EVs serve all the customers without exceeding their carrying capacity and no EV runs out of charge during the tour. All EVs begin and end at the depot, EVs always leave the charging station fully charged, and the charging stations (including the depot) can be visited multiple times by any EV.

The EVRP can be mathematically formulated as follows:

$$\min \sum_{i \in V, j \in V, i \neq j} d_{ij} x_{ij}, \text{ s.t.} \tag{3.16}$$

$$\sum_{j \in V, i \neq j} x_{ij} = 1, \ \forall i \in I, \tag{3.17}$$

$$\sum_{j \in V, i \neq j} x_{ij} \leq 1, \ \forall i \in F', \tag{3.18}$$

$$\sum_{j \in V, i \neq j} x_{ij} - \sum_{j \in V, i \neq j} x_{ji} = 0, \ \forall i \in V, \tag{3.19}$$

$$u_i \leq u_j - b_i x_{ij} + C(1 - x_{ij}), \ \forall i \in V, \ \forall j \in V, \ i \neq j, \tag{3.20}$$

$$0 \leq u_i \leq C, \ \forall i \in V, \tag{3.21}$$

$$y_j \leq y_i - h d_{ij} x_{ij} + Q(1 - x_{ij}), \ \forall i \in V, \ \forall j \in V, \ i \neq j, \tag{3.22}$$

$$y_j \leq Q - hd_{ij}x_{ij},\ \forall i \in F' \cup \{0\},\ \forall j \in V,\ i \neq j, \tag{3.23}$$

$$0 \leq y_i \leq Q,\ \forall i \in V, \tag{3.24}$$

$$x_{ij} \in \{0,1\},\ \forall i \in V,\ \forall j \in V,\ i \neq j, \tag{3.25}$$

where G is a fully connected weighted graph $G = (V, A)$, $V = \{0 \cup I \cup F'\}$ is a set of nodes and $E = \{e_{i,j} |\ i, j \in V, i \neq j\}$ is a set of edges. Nodes consist of disjoint sets of customers I, charging stations F' and a depot 0. To permit multiple visits to the same charging station, F' consist of β_i node copies of each charging station $i \in F$ (e.g. $|F'| = \sum_{i \in F} \beta_i$). Each customer $i \in I$ is assigned a positive demand b_i. Symbol d_{ij} denotes the weight of the edge $e_{i,j}$ and it is equivalent to the Euclidean distance between its nodes. Car traversing edge $e_{i,j}$ consumes hd_{ij} of the remaining battery charge level of the EV, where h denotes the charge consumption rate. All cars have the same maximal carrying capacity C and maximal battery charge level Q. Variables u_i and y_i denote the remaining carrying capacity and remaining battery charge level of an EV on its arrival at node $i \in V$.

3.4.2 ROADEF 2020 Challenge: Grid operation-based outage maintenance planning

This problem was proposed in 2020 as part of a biannual programming challenge by the French Society for Operational Research and Decision Support [Roa23]. It is a type of Transmission Maintenance Scheduling problem where the goal is to find a maintenance schedule that reduces the probability of grid failure. Scheduled repairs are performed at predetermined intervals or according to prescribed criteria. Maintenance in the electricity industry concerns generating units and transmission lines with either long-term or short-term horizons [FGM+16]. The proposed problem deals with electrical grid repairs over a one-year horizon. The next sections are a slightly reformulated problem description and definition as given in [RTP20].

Introduction

Réseau de Transport d'Électricité (Electricity Transmission Network or RTE for short) is the electricity transmission system operator of France. It is responsible for the operation, maintenance and development of the French high-voltage transmission system. The mission of RTE is to guarantee electricity delivery to French citizens. To achieve its goal the grid needs to be

correctly maintained without interruption to the electricity supply. Factors such as workforce availability, seasonal and unexpected weather, and the size of the grid itself make scheduling these repair operations a highly complex problem.

To solve this planning issue RTE implemented a three-step approach. First, risk values of different future grid scenarios are computed. Second, obtained risk values are inputted into an optimization algorithm which returns a schedule. Third, the proposed schedule is validated. ROADEF challenge focuses on the second step of this approach: given the risk values and problem constraints, the goal is to find an optimal planning regarding a risk-based objective.

Notation, Inputs and Output

This section includes input, output and internal variables used in constraints and fitness functions.

- **Planning horizon H:** The schedule has to be established over a one-year period but the time step of the schedule can vary depending on the needed precision. The number of time steps in a year is $T \in \mathbb{N}$ and the discrete-time horizon is $H = \{1, ..., T\}$ (e.g. $T = 52$ is a weekly schedule).
- **Resources C:** Set of all resources C that may be needed to carry out the different interventions.
  - **Maximum resources u_t^c:** There is an upper bound on available resources that varies in time. The highest possible usage of resource $c \in C$ at time $t \in H$ is denoted by $u_t^c \in \mathbb{R}$.
  - **Minimum resources l_t^c:** For operational reasons, there is also a lower bound on resource consumption that varies in time. The lowest possible usage of resource $c \in C$ at time $t \in H$ is denoted by $l_t^c \in \mathbb{R}$.
- **Interventions I:** Interventions I are tasks that have to be planned in the coming year. They differ in terms of duration and resource requirements.
  - **Time duration $\Delta_{i,t}$:** The duration of a given intervention is not fixed in time and depends on when it starts (because of days off, weekends, holidays, etc.). Duration of intervention $i \in I$ starting at time $t \in H$ is denoted by $\Delta_{i,t} \in \mathbb{N}$.

- **Resource workload $r_{i,t'}^{c,t}$:** Every intervention requires different resources all of which also vary in time. Workload required for resource $c \in C$ at time $t \in H$ by intervention $i \in I$ if i begins at time $t^{'} \in H$ is denoted by $r_{i,t'}^{c,t} \in \mathbb{R}^{+}$.

- **Risk $risk_{i,t'}^{s,t}$:** When an intervention is being performed the corresponding lines have to be disconnected causing the electricity network to be weakened at this time. This implies a financial risk for RTE, because if another close site was to break down (due to extreme weather for example) the grid may not be able to operate correctly. Risk is evaluated according to different scenarios S_t at time $t \in H$ which do not dependent on interventions but are a result of simulating a certain grid operation. However, $risk_{i,t'}^{s,t} \in \mathbb{R}$ depends on the considered intervention $i \in I$ and its start $t^{'} \in H$, the time period $t \in H$ (generally summer is safer than winter) and scenario $s \in S_t$.

- **Solution L:** A schedule (or solution) is a list L of pairs $(i, t) \in I \times H$, where t is the starting time of intervention i. Starting time of intervention $i \in I$ is denoted by $start_i$ and $I_t \subseteq I$ is the set of interventions in process at time $t \in H$.

Constraints

There are three types of constraints. Schedule constraints ensure that all jobs are completed without interruption and finished before the end of the schedule. Workforce constraints ensure that total workforce use doesn't exceed lower and upper limits of available resources. Disjunctive constraints ensure that interruptions that are deemed too risky are not executed at the same time. A planning is said feasible if all constraints presented below hold.

- **Schedule constraints:**
  - Interventions have to start at the beginning of a period.
  - Once an intervention starts, it cannot be interrupted. If intervention $i \in I$ starts at time $t \in H$, then it has to end at $t + \Delta_{i,t}$.
  - All interventions have to be executed.
  - All interventions must be completed no later than the end of the horizon. If intervention $i \in I$ starts at time $t \in H$, then $t + \Delta_{i,t} \leq T + 1$

- **Resource constraints:**

- Given a solution, the workload due to intervention i for resource c at time t is $r^{c,t}_{i,start_i}$. Then the total resource workload for c at time t is

$$r^{c,t}_{total} = \sum_{i \in I_t} r^{c,t}_{i,start_i}. \tag{3.26}$$

- Total resource workload must lie between the limits

$$l^c_t \leq r^{c,t}_{total} \leq u^c_t, \ \forall c \in C, \ \forall t \in H. \tag{3.27}$$

- **Disjunctive constraints:** Some interventions are too close to each other to be carried out at the same time. The set of exclusions is denoted by Exc. It is a set of triplets (i_1, i_2, t), where $i_1, i_2 \in I$ and $t \in H$. The exclusion constraints can formally be written as

$$i_1 \in I_t \implies i_2 \notin I_t, \ \forall (i_1, i_2, t) \in Exc. \tag{3.28}$$

Objective

The score evaluation of a feasible planning only depends on the risk distribution. The overall mean cost appears quickly as an excellent candidate, but it is not enough for taking into account specific behaviors of the risk distribution. In the end, RTE found that expected excess, when in combination with the mean cost, is able to take into account specific behaviors of the risk distribution.

- **Mean cost** obj_1**:**
  - Given a solution, the cumulative planning risk at time $t \in H$ for a scenario $s \in S_t$ is

$$risk^{s,t} = \sum_{i \in I_t} risk^{s,t}_{i,start_i}. \tag{3.29}$$

  - The mean cumulative planning risk at time $t \in H$ is

$$\overline{risk^t} = \frac{1}{|S_t|} \sum_{s \in S_t} risk^{s,t}. \tag{3.30}$$

  - Then the overall planning risk (or mean cost) is

$$obj_1 = \frac{1}{t} \sum_{t \in H} \overline{risk^t}. \tag{3.31}$$

- **Expected excess** obj_2**:**

- Let $E \subset \mathbb{R}$ be a non-empty finite set and $\tau \in [0, 1]$. The τ quantile of E, denoted $Q_t(E)$ is:

$$Q_\tau(E) = \min\{q \in \mathbb{R} : \exists X \subseteq E : |X| \geq \tau \times |E| \; and \; \forall x \in X, x \leq q\} \tag{3.32}$$

- For every time period $t \in H$, we define the quantile value Q_τ^t as

$$Q_\tau^t = Q_\tau(\{risk^{s,t}\}_{s \in S_t}). \tag{3.33}$$

- The expected excess at time $t \in H$ is then defined as

$$Excess_\tau(t) = \max(0, Q_\tau^t - \overline{risk^t}). \tag{3.34}$$

- The expected excess of a planning is

$$obj_2(\tau) = \frac{1}{t} \sum_{t \in H} Excess_\tau(t). \tag{3.35}$$

- **Planning ranking** $obj(\tau)$**:** The two metrics described above are in euros. However, they cannot necessarily be compared directly, as they depend on risk aversion (or risk policies). That is why a scaling factor $\alpha \in [0, 1]$ is provided. The final score of a planning then is

$$obj(\tau) = \alpha \; obj_1 + (1 - \alpha) \; obj_2(\tau). \tag{3.36}$$

  The goal is to find a feasible planning with the lowest possible score regarding this objective.

Chapter 4

Implementation

In the first part of this chapter, we will introduce our changes to the GMS [WHK22]. Those include modifications to the Solution class, which was extended to include a penalty vector $P = [p_0, .., p_p]$, and the implementation of a new Generic Optimizer class in the form of modified ASCHEA with population sizing scheme and newly proposed crossover operators. The second part describes the implementation of the EVRP and ROADEF problems.

4.1 Generic Optimizer

The structure of the Generic Optimizer mimics the structure of ASCHEA. The operators and other static parameters are the same for all populations. A single instance of ASCHEA has a similar structure as standard EA described in EA Pipeline section, and it can be summarized by the flowchart in Figure 4.1.

ASCHEA itself only defines selection, replacement, and penalty initialization and update mechanisms. Population initialization, crossover and mutation operators were chosen by us. The algorithm terminates after a user-determined amount of time has elapsed. If parameter $stop\_on\_feasible$ is true algorithm can terminate sooner if a feasible solution has been found first. If the population sizing scheme is active, several instances of ASCHEA, each with its unique population, are run in parallel. Parameters that depend on the state of the population are not shared between instances, more on

that later in this chapter.

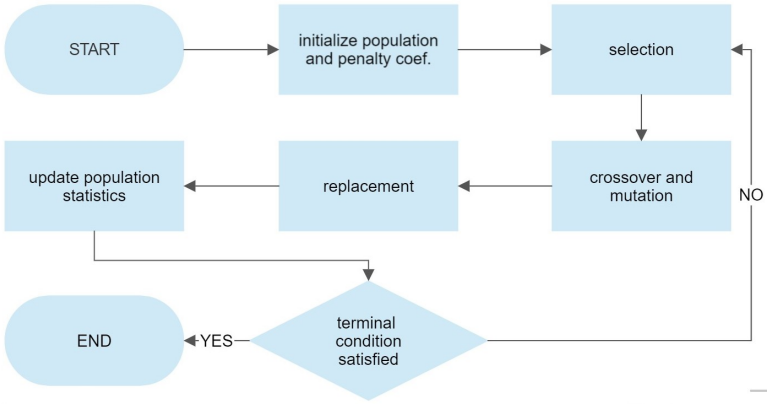


Figure 4.1: ASCHEA flowchart

4.1.1 Fitness and penalties

Standard ASCHEA operates on problems in $\mathbb{R}^n$ space with constraints of two types, inequality constraints

$$u_i(X) \leq 0, \ i = [1, .., m], \tag{4.1}$$

and equality constraints

$$v_j(X) = 0, \ j = [m+1, .., p], \tag{4.2}$$

which are often relaxed to $|v_j(X) - \epsilon| \leq 0$ for some small ϵ.
From these constraints their constraint violation functions $\nu_i, \ i = [1, .., p]$ are defined

$$\nu_i(X) = \begin{cases} \max(0, u_i(X)), & \text{if } 1 \leq i \leq m, \\ |v_i(X)|, & \text{if } m < i \leq p. \end{cases} \tag{4.3}$$

Since our problems aren't in $\mathbb{R}^n$ space we cannot use these constraints and their constraint violation functions. Constraint violations are instead returned in the penalty vector $P = [p_0, .., p_p]$ by the user-implemented Compute fitness function g of the Problem class, which is inherited from the abstract class Instance. Penalty vector $P = [p_0, .., p_p]$ consists of un-penalized fitness p_0 and the solutions constraint violations $p_1, .., p_p$. To ensure better performance constraint violations shouldn't be binary, but should indicate if possible, an amount of violation.

In the first version of ASCHEA [SH00] single penalty coefficient was used for all constraint violations. However, it has been shown that constraint

satisfaction difficulty differs from one constraint to another. So in the second version [BHS02] each constraint has its own penalty coefficient. Penalized fitness $g_p(X)$ is computed independently inside our Generic Optimizer as

$$g_p(X) = \sum_{i=0}^{p} \alpha_i p_i(X), \tag{4.4}$$

where $\alpha_1, .., \alpha_p$ are penalty coefficients and α_0 is the fitness coefficient. Term $p_i(X)$ is equal to the i^{th} element of the penalty vector P of the solution X. By fitness of the solution X, it is from now on meant penalized fitness $g_p(X)$ unless stated otherwise.

Coefficients initialization

At the start of the algorithm penalty coefficients $\alpha_i(0),\ i = [1, .., p]$ are initialized from the first generation of solutions $[X_1, .., X_n]$ as

$$\alpha_i(0) = \begin{cases} 1, \text{ if } \sum_{j=1}^{n} p_i(X_j) = 0, \\ \frac{\sum_{j=1}^{n} |p_0(X_j)|}{\sum_{j=1}^{n} |p_i(X_j)|} \times 100, \text{ otherwise,} \end{cases} \qquad i = [1, .., p]. \tag{4.5}$$

The initial value of α_0 is set to $\alpha_0(0) = 1$.

Coefficients update

All penalty coefficients $\alpha_1, .., \alpha_p$ are adapted at the end of each generation t by the following rule:

$$\alpha_i(t+1) = \begin{cases} \alpha_i(t)/fact, \text{ if } \tau_t(i) > \tau_{target}, \\ \alpha_i(t) \times fact, \text{ otherwise,} \end{cases} \qquad \text{for } i \in [1, .., p], \tag{4.6}$$

where τ_{target} is a user-defined parameter that represents the desired proportion of individuals satisfying a constraint, $\tau_t(i)$ is the proportion of individuals satisfying constraint i at generation t and $fact$ is a penalty scaling factor (usually $fact = 1.1$). It holds that $\tau_{target}, \tau_t, fact \in \mathbb{R}$ and $\tau_{target}, \tau_t \in [0, 1],\ fact > 1$.

Linear coefficient rescaling scheme

With the introduction of the adaptive penalty coefficients, we encounter two problems that didn't arise in the GMS-LS because it used static, user-defined penalties.
The first problem is less significant of the two and could be avoided by using floats to store fitness/penalties. We store the penalties as unsigned long integers, the same as fitness in the previous implementation. The problem occurs when penalty coefficient α_k has easily satisfiable constraints p_k. The penalty coefficient α_k is repeatedly divided by $fact$ and after $\alpha_k p_k < 0.5$ the product is rounded to 0 and it doesn't contribute to the final penalty function.
The second problem is the exact opposite and it occurs when penalty coefficient α_k has a difficult constraint p_k. As a result, the penalty coefficient α_k is repeatedly multiplied by $fact$. For example, if starting with $p_k = 1$, $\alpha_k = 1$ and $fact = 1.1$, after approximately $\log_{fact}(4,294,967,295) \approx 233$ generations the product $\alpha_k p_k$ will exceed the maximal size of an unsigned long integer of 4,294,967,295. Since the second problem would persist even with different choice of data type we decided to implement a linear coefficient rescaling scheme.

Rescaling is meant to preserve relations between coefficients without them exceeding the lower limit of 1 and user-defined upper limit $\alpha_{ceiling}$. The upper limit can also be generated automatically by the program depending on the problem as

$$\alpha_{ceiling} = \max_{i=[1,..p]} \alpha_i(0). \tag{4.7}$$

The rescaling scheme is then applied to all coefficients and occurs after Coefficients update is performed.

$$\alpha_i = \begin{cases} 1 + \alpha_i - \alpha_{min}, & \text{if } \alpha_{max} < \alpha_{ceiling}, \\ 1 + (\alpha_i - \alpha_{min})\frac{\alpha_{ceiling}}{\alpha_{max}}, & \text{otherwise,} \end{cases} \tag{4.8}$$

where α_{min} and α_{max} are the smallest and the biggest coefficient in the current generation.

4.1.2 Initialization

Users can choose one of the four methods for initializing the population, all of which create solutions inside the lower and upper bounds. The core functionality of all initialization operators was already implemented and is

described in detail in [Hra22, pg. 24,25]. We modified Greedy and Nearest-Neighbor operators to begin their process from unique starting points.

- **Random** starts with an empty solution. It then inserts nodes below the lower bound to random locations in the solution until all lower bounds are satisfied. As a result the returned solution always has $F = LB$.

- **Random-replicate** generates a random permutation of all nodes. It then repeatedly appends the permutation to itself until all the lower bounds are satisfied. If a node reaches its upper bound before all lower bounds are satisfied it is excluded from the next copies of the permutation.

- **Greedy** starts with incomplete solution X of k random nodes that are below upper bound. It then inserts a node $a \in [1, .., |A|]$ into the previous solution thus creating a new solution (node a must be below the upper bound). After iterating over all possible node and position combinations, the new solution with the smallest non-penalized fitness g continues to the next iteration. Insertions are repeated until a valid solution is obtained. Random starting solution of length $k = \lceil \frac{\lambda}{|A|} \rceil$ is used as a seed for the generation of unique solutions.

- **Nearest-Neighbor** is similar to greedy construction but instead of insertion, it appends the node a at the end of the solution.

4.1.3 Selection

To ensure better exploration of the space near the constraints' boundaries and to attract infeasible individuals to the feasible region, without restricting the search too much, a constraint-driven selection/seduction mechanism is used [Ron95]. It chooses a mate of a feasible individual to be an infeasible one, but only if the number of feasible individuals in the population is bellow $\tau_{target}\lambda$. It only limits from which solutions the mate can be selected and not how the mate is chosen. This means that any selection operator can be used in combination with this mechanism. We used tournament selection with user-defined $tournament\_size \in \mathbb{N}$. The size of the parent population $number\_of\_parents$ is set to population size λ.

Algorithm 1 *Tournament_selection*(*population*)

```
1: parents ← [ ]
2: for i = 0 to number_of_parents do
3:     tournament ← [ ]
4:     for j = 0 to tournament_size do
5:         tournament.append(random(population))
6:     end for
7:     parents.append(fittest_solution(tournament))
8: end for
```

The algorithm creates *number_of_parents* tournaments (1-3). All tournaments are filled with *tournament_size* random solutions from the population (4-6). The fittest individual from each tournament is then added to the parent population (7).

4.1.4 Crossover

In this section, we describe 18 binary crossover operators, most of them being extensions of classical crossovers for problems with permutation representation (e.g. TSP, VRP, QAP...) [LKM+99], [PM13], [MK05]. We divided them into two main groups, positional and neighboring crossovers. We also introduce a concept of alignment and implement 6 alignment operators to be used in combination with positional crossovers.

All crossover operators accept two parent solutions X,Y and return one child solution Z. The user can define a set of crossover operators C which will be used in reproduction.

Algorithm 2 *Crossover*(*parents*)

```
1: children ← []
2: for i = 1 to number_of_parents do
3:     X ← parents[i]
4:     Y ← parents[i + 1]
5:     crossover_operator ← random(C)
6:     children.append(crossover_operator(X, Y))
7:     children.append(crossover_operator(Y, X))
8:     i ← i + 2
9: end for
```

Crossover selects adjacent parents (3-4) and chooses a random crossover operator from the user-provided set of operators C (5). Crossover is then applied to the parents and the resulting child solutions are added to the child population (6-7). The process is repeated $number\_of\_parents$-times with the next parents (2,8).

Alignment operators

For positional crossovers to work both solutions X, Y have to have the same length, this is guaranteed only if $LB = UB$. To enable the crossover of differently-sized solutions we introduce a concept of alignment. It inserts copies of a gap node a_{gap} into the solutions and thus equalizes their length. Six alignment operators are described in the next section.

Let the length of the solution X be $|X|$, $|X| \geq |Y|$ and $a_{gap} = |A| + 1$ (e.g. one bigger than the biggest node in the solution). The lower and upper bounds of the gap node are set $l_{a_{gap}} = 0$ and $u_{a_{gap}} = \infty$.

- **Front-fill one-gap** prepends a string of gap nodes of length $|X| - |Y|$ at the start of the shorter solution.
- **Back-fill one-gap** appends a string of gap nodes of length $|X| - |Y|$ at the end of the shorter solution.
- **Random one-gap** inserts a string of gap nodes of length $|X| - |Y|$ at a random position into the shorter solution.
- **Greedy one-gap** inserts a string of gap nodes of length $|X| - |Y|$ into the shorter solution, thus creating $\hat{X}, \hat{Y}$. The string is inserted into the position k which minimizes

$$\sum_{i=1}^{|X|} [\![\hat{x}_i \neq \hat{y}_i]\!] \tag{4.9}$$

- **Random uniform** inserts $|X| - |Y|$ gap nodes at random positions into the shorter solution.
- **Greedy uniform** inserts gap nodes possibly into both solutions thus creating $\hat{X}, \hat{Y}$. Alignment returns $\hat{X}, \hat{Y}$ that minimize the penalty function

$$\sum_{i=1}^{k} p_{neq} [\![\hat{x}_i \neq \hat{y}_i]\!] + p_{gap}([\![\hat{x}_i = a_{gap}]\!] + [\![\hat{y}_i = a_{gap}]\!]), \tag{4.10}$$

where $p_{neq}, p_{gap} \in \mathbb{N},\ p_{neq}, p_{gap} \geq 1$ are user-defined penalty coefficients. The number of mismatched pairs between the solution is multiplied by p_{neq}, and the number of gaps by p_{gap}. Constant k is the length of both solutions after the insertion of gap nodes. This operator is a version of the Needleman-Wunsch algorithm [NW70], also knowns as the global alignment algorithm.

Positional crossovers

Positional crossover works with absolute positions e.g. it knows that in solution [1 2 3] node 1 is in the first position. Chosen alignment operator is applied to both parents X and Y before the crossover operator. The gap node is removed afterward from both parents and child. The frequency of node a in the child solution Z is further denoted as just f_a.

- **PBX** (Position Based Crossover) [SVG87] is a standard TSP operator. The child inherits the absolute positions of nodes on random positions from the first parent. Our variation of PBX is described with an example below.

  The child is initialized as a copy of parent Y. The operator then selects a random set of positions ($\{4, 6, 8\}$).

  $$X: \mathbf{3\ 1\ 4\ \underline{1}\ 5\ \underline{9}\ 2\ \underline{6}\ 5\ 3}$$

  $$Y: \mathbf{2\ 7\ 1\ 8\ 2\ 8\ 1\ 8\ 2\ 8}$$

  Nodes on the selected positions in the parent X ($[1, 9, 6]$) are removed from the child in the same frequencies as they have in the selected set. (only one of the 1s is removed, all are as likely to be chosen).

  $$Z: \mathbf{2\ 7\ 1\ 8\ 2\ 8\ 8\ 2\ 8}$$

  Selected nodes are reinserted back into the child at the same positions they have in parent X.

  $$Z: \mathbf{2\ 7\ 1\ \underline{1}\ 8\ \underline{9}\ 2\ \underline{6}\ 8\ 8\ 2\ 8}$$

- **NBX** (Node Based Crossover) is our operator inspired by PBX. The child inherits all absolute positions of random nodes from the first parent. NBX is described with an example below.

  The child is initialized as a copy of parent Y.

  $$X: \mathbf{3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3}$$

$$Y: \mathbf{2\ 7\ 1\ 8\ 2\ 8\ 1\ 8\ 2\ 8}$$

$$Z: \mathbf{2\ 7\ 1\ 8\ 2\ 8\ 1\ 8\ 2\ 8}$$

The operator then selects a set of random nodes ($\{1, 2, 4\}$), removes all of their occurrences from the child

$$Z: \mathbf{7\ 8\ 8\ 8\ 8}$$

and then reinserts them back into the child at the positions they have in parent X.

$$Z: \mathbf{7\ 1\ 4\ 1\ 8\ 8\ 2\ 8\ 8}$$

- **OBX** (Order Based Crossover) [SVG87] is a TSP operator. The child inherits the relative positions of nodes on random positions from the first parent. Our variation of OBX is described with an example below.

  The operator starts by selecting a set of random nodes ($\{3, 5, 8\}$) whose sums of frequencies are the same in both parents.

  $$X: \mathbf{\underline{3}\ 1\ 4\ 1\ \underline{5}\ 9\ 2\ 6\ \underline{5}\ \underline{3}}$$

  $$Y: \mathbf{2\ 7\ 1\ \underline{8}\ 2\ \underline{8}\ 1\ \underline{8}\ 2\ \underline{8}}$$

  The child is then set as a copy of parent Y. The selected nodes in the child are then replaced by the selected nodes as ordered in X.

  $$Z: \mathbf{2\ 7\ 1\ \underline{3}\ 2\ \underline{5}\ 1\ \underline{5}\ 2\ \underline{3}}$$

- **OX** (Ordered Crossover) [D+85] is a standard TSP and VRP operator. It is a mix between the positional and neighboring operators, but since it needs same-sized solutions we included it in this section. The child inherits a substring, with its position, from the first parent. The rest of the child is filled by the second parent. Our variation of OX is described with an example below.

  First, a random substring of X starting at i and ending before j ($i = 4, j = 8$) is copied to the child.

  $$X: \mathbf{3\ 1\ 4\ |\ 1\ 5\ 9\ 2\ |\ 6\ 5\ 3}$$

  $$Y: \mathbf{2\ 7\ 1\ |\ 8\ 2\ 8\ 1\ |\ 8\ 2\ 8}$$

  $$Z: \mathbf{*\ *\ *\ |\ 1\ 5\ 9\ 2\ |\ *\ *\ *}$$

  A list of nodes L, taken from Y starting at position j, wrapping around the end of the Y and ending before j, is then copied to Z ($L = [8, 2, 8, 2, 7, 1, 8, 2, 8, 1]$). The same number of nodes are inserted after the substring in Z as are in X and Y. Nodes from the substring $(1, 2, 5, 9)$ are removed from the list L in the same frequencies as they have in the substring ($L = [8, 2, 8, 7, 8, 2, 8, 1]$).

  $$Z: \mathbf{7\ 8\ 2\ 8\ 1\ |\ 1\ 5\ 9\ 2\ |\ 8\ 2\ 8}$$

- **MPX** (Maximally Preserving Crossover) [MGSK88] is a TSP operator. The child inherits a substring S from the first parent. The rest is appended to the substring from the second parent. If viewed as a standard TSP operator, it is designed to break at most $|S|$ edges between the parent solutions. Our variation of MPX is described with an example below.

  The child is initialized as a random substring S of X. The size of the substring must be between 10 and $\frac{|X|}{2}$ for solutions $|X| > 20$.

  $$X: \mathbf{3\ 1\ 4\ |\ 1\ 5\ 9\ 2\ |\ 6\ 5\ 3}$$

  $$Y: \mathbf{2\ 7\ 1\ 8\ 2\ 8\ 1\ 8\ 2\ 8}$$

  $$Z: \mathbf{1\ 5\ 9\ 2}$$

  Nodes in the substring ($[1, 2, 5, 9]$) are removed from the copy of parent Y in the same frequencies as they have in the substring.

  $$Y_{copy}: \mathbf{2\ 7\ 8\ 8\ 1\ 8\ 2\ 8}$$

  The copy of Y is then appended to the child.

  $$Z: \mathbf{1\ 5\ 9\ 2\ 2\ 7\ 8\ 8\ 1\ 8\ 2\ 8}$$

- **APX** (Alternating Position Crossover) [LKPM97] is a TSP operator. Our variation of APX is described with an example below.

  It zips together the parents $X = [x_1, ..., x_{|X|}], Y = [y_1, ..., y_{|Y|}]$.

  $$X: \mathbf{3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3}$$

  $$Y: \mathbf{2\ 7\ 1\ 8\ 2\ 8\ 1\ 8\ 2\ 8}$$

  The child starts as an empty solution. Nodes are sequentially appended to the child from list $[x_1, y_1, x_2, y_2, ..., x_{|X|}, y_{|Y|}]$ only if $f_{x_i} < u_{x_i}$ (or $f_{y_i} < u_{y_i}$).

  $$Z: \mathbf{3\ 2\ 1\ 7\ 4\ 1\ 8\ 5\ 2\ 9\ 8\ 2\ 6\ 8\ 5\ 3\ 8}$$

- **CX** (Cyclic crossover) [MF99] is a standard TSP and VRP operator. Every position in the child is assigned a node from one of the parents with the same position. To achieve this, the operator founds a cycle of nodes between the parents. Our variation of CX is described with an example below.

  First, a directed graph $G = (V, E)$ of node relations between the two parents is created. Set of nodes $V = A \cup gap$. Edge (i, j) belongs to the graph if nodes i and j are in the same position in X and Y respectively. Or formally $E = \{(i, j) \in \mathbb{N}^2 |\ \exists k \in \mathbb{N} : x_k = i \text{ and } y_k = j\}$. For parents

  $$X: \mathbf{3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3}$$

  $$Y: \mathbf{2\ 7\ 1\ 3\ 2\ 8\ 1\ 8\ 2\ 8}$$

  graph G can be described by the following edge map:

node	neighbors
1	7,3
2	1
3	2,8
4	1
5	2
6	8
9	8

Breath First Search is then started on the graph beginning with the edge x_1, y_1 $(3, 2)$ and ending when we first revisit the starting node x_1 $(3 \rightarrow 2 \rightarrow 1 \rightarrow 3)$. If no such cycle exists within this graph crossover returns X as the child. Otherwise, nodes on the positions corresponding to the found cycle (positions $1, 4, 7$) are copied from X to child

$$X: \mathbf{3} * * \mathbf{1} * * \mathbf{2} * * *$$

and the rest of the nodes are copied from Y.

$$X: \mathbf{3\ 7\ 1\ 1\ 2\ 8\ 2\ 8\ 2\ 8}$$

- **ULX** (Uniform Like Crossover) [TS95] is a QAP crossover. It is a variation of uniform crossover for permutations. Our variation of ULX is described in the pseudo-code below.

Algorithm 3 $ULX(X, Y)$

1: $Z \leftarrow [blank, ..., blank]$.
2: $indexes \leftarrow [1, ..., |X|]$
3: **for** $i \in indexes$ **do**
4: **if** $x_i = y_i$ **then**
5: $z_i = x_i$
6: **end if**
7: **end for**
8: **for** $i \in indexes$ **do**
9: **if** $z_i = blank$ and $f_{x_i} < l_{x_i}$ and $f_{y_i} < l_{y_i}$ **then**
10: $z_i \leftarrow \text{random}(x_i, y_i)$
11: **else if** $z_i = blank$ and $f_{x_i} < l_{x_i}$ **then**
12: $z_i \leftarrow x_i$
13: **else if** $z_i = blank$ and $f_{y_i} < l_{y_i}$ **then**
14: $z_i \leftarrow y_i$
15: **end if**
16: **end for**
17: **for** $i \in indexes$ **do**
18: **if** $z_i = blank$ and $f_{x_i} < u_{x_i}$ and $f_{y_i} < u_{y_i}$ **then**
19: $z_i \leftarrow \text{random}(x_i, y_i)$
20: **else if** $z_i = blank$ and $f_{x_i} < u_{x_i}$ **then**
21: $z_i \leftarrow x_i$
22: **else if** $z_i = blank$ and $f_{y_i} < u_{y_i}$ **then**
23: $z_i \leftarrow y_i$
24: **end if**
25: **end for**
26: **for** $i \in indexes$ **do**
27: **if** $z_i = blank$ **then**
28: $z_i = \text{random}(a \in A | f_a < l_a)$
29: **end if**
30: **end for**
31: **for** $a \in A$ **if** $f_a < l_a$ **do**
32: Z.append(a)
33: **end for**
34: $Z \leftarrow Z \backslash blank$

The operator first fills the child Z of size $|X|$ with empty spaces (1). It then iterates four times over the positions from left to right (2). In the first iteration, all nodes in the same positions are copied to the child (3-7). In the second iteration, all nodes below the lower bounds are copied to the child (8-16). In the third iteration, all nodes below the upper bounds are copied to the child (17-25). In the fourth iteration, all the empty spaces in the child are filled with nodes below the lower bound (26-30). Finally, all nodes still below the lower bound are appended at the end (31-33) and the empty spaces are removed from the child (34).

- **RULX** (Random Uniform Like Crossover) [MK05] is a QAP crossover and it is a variation of ULX. The algorithm for RULX is the same as the algorithm for ULX shown above. Only change occurs on line (2)

$$2: \ indexes \leftarrow \text{shuffle}([1, ..., |X|])$$

  where the indexes are shuffled. As a result, the operator goes thru positions in random order instead of from left to right.

- **EULX** (Extended Uniform Like Crossover) [MK05] is a QAP crossover and it is an extension of ULX. The goal of this operator is to improve the child solution found by ULX with local search. Our variation of ULX is described in the pseudo-code below.

Algorithm 4 $EULX(X, Y)$

```
1:  Z ← ULX(X, Y).
2:  CL ← {}
3:  for i ∈ [1, ..., Z] do
4:      if x_i ≠ z_i and y_i ≠ z_i then
5:          CL ← CL ∪ i
6:      end if
7:  end for
8:  improved ← true
9:  while improved do
10:     improved ← false
11:     Z' ← Z
12:     for i ∈ CL do
13:         for j ∈ CL if j > i do
14:             Z'' ← Z
15:             swap(z''_i, z''_j)
16:             if fitness(Z'') < fitness(Z') then
17:                 Z' ← Z''
18:             end if
19:         end for
20:     end for
21:     if fitness(Z') < fitness(Z) then
22:         CL ← CL\{i ∈ ℕ|z_i ≠ z'_i}
23:         Z ← Z'
24:         improved ← true
25:     end if
26: end while
```

First ULX is applied to X and Y producing the child Z (1). Then a candidate list CL of all positions, where the child doesn't equal either of its parents, is created (2-7). A local search, using a swap operator, is performed until no improving solution is found (8-26). First, the search founds the most improving swap between the positions from

the candidate list (11-20). If the newly found solution Z' has lower fitness than the previous child Z (21-25) then the swapped positions are removed from the candidate list (22), and the child is replaced by the better solution (23).

- **ERULX** is a QAP crossover and it is a variation of EULX. The algorithm for ERULX is the same as the algorithm for EULX shown above. The only change occurs on line (1)

$$1: \ Z \leftarrow RULX(X, Y)$$

where the RULX operator is used, instead of ULX, to obtain the child.

- **UPMX** (Uniform Partially Mapped Crossover) [MTKT96] is a version of Partially Mapped Crossover (PMX) for QAP. Since we were unable to successfully implement a standard PMX, we resorted to adapting UPMX instead. UPMX uses a similar mapping principle as PMX, but instead of a continuous mapping segment, UPMX uses several random mapping positions. Our variation of UPMX is described in the pseudo-code below.

Algorithm 5 $UPMX(X, Y)$

```
1: Z ← X
2: for k = 1 to |X|/3 do
3:     i ← random({1, ..., |X|})
4:     S ← {j ∈ ℕ| x_i = y_j}
5:     if |S| > 0 then
6:         j ← random(S)
7:         swap(z_i, z_j)
8:     end if
9: end for
```

Child Z is set as a clone of X (1). The following cycle is repeated $\frac{|X|}{3}$ times (2-9). Random position i in X is chosen (3). Then we find a position j in Y where $x_i = y_j$ (4). If at least one such position exists choose one, j, at random (5-6). Swap z_i, z_j (7).

- **SPX** (Swap Path Crossover) [Glo94], sometimes referred to as Path Relinking Crossover, is a QAP crossover. Our variation of SPX is described in the pseudo-code below.

Algorithm 6 $UPMX(X, Y)$

1: $children \leftarrow \{\}$
2: $start \leftarrow \text{random}(\{1, ..., |X|\})$
3: **for** $offset = 0$ **to** $|X| - 1$ **do**
4: $\quad Z^x \leftarrow X$
5: $\quad Z^y \leftarrow Y$
6: $\quad i \leftarrow (start + offset) \bmod |X|$
7: $\quad \alpha \leftarrow z_i^x$
8: $\quad \beta \leftarrow z_i^y$
9: $\quad$ **if** $f_\beta^{Z^x} > 0$ **then**
10: $\quad\quad S \leftarrow \{j \in \mathbb{N} |\ z_j^x = \beta\}$
11: $\quad\quad j \leftarrow \text{random}(S)$
12: $\quad\quad z_j^x \leftarrow \alpha$
13: $\quad\quad z_i^x \leftarrow \beta$
14: $\quad$ **end if**
15: $\quad$ **if** $f_\alpha^{Z^y} > 0$ **then**
16: $\quad\quad S \leftarrow \{j \in \mathbb{N} |\ z_j^y = \alpha\}$
17: $\quad\quad j \leftarrow \text{random}(S)$
18: $\quad\quad z_j^y \leftarrow \beta$
19: $\quad\quad z_i^y \leftarrow \alpha$
20: $\quad$ **end if**
21: $\quad children \leftarrow children \cup Z^x \cup Z^y$
22: $\quad$ **if** $fitness(Z^x) < fitness(Z^y)$ **then**
23: $\quad\quad X \leftarrow Z^x$
24: $\quad$ **else**
25: $\quad\quad Y \leftarrow Z^y$
26: $\quad$ **end if**
27: **end for**
28: $Z \leftarrow \text{fittest}(children)$

First, an empty set of *children* is created (1) and a random starting position $i = start + offset$ is chosen (2). The following process is repeated $|X|$ times until all positions were visited (3-27). The operator clones the parents X, Y to temporary children Z^x and Z^y (4-5). The values of z_i^x, z_i^y before the swap are α, β (7,8). If Z^x has at least one node of value β (9), take a random node from Z^x with the value β and assign it the value α (10-12), next assign z_i^x value β (13). The same swap is done for Z^y (15-20). Both children are added to the set of children (21). The fitter of the modified temporary children then replaces its parent, e.g. child Z^x replaces X or Z^y replaces Y (22-26). The fittest of all *children* is returned (29).

Neighboring crossovers

Neighboring crossovers only utilize information about the relative positions of nodes e.g. it knows that in solution [1 2 3] node 1 is a left neighbor of node 2.

Vector of all neighbors of node x in solution X (with repetitions) is $\overleftrightarrow{N_X}(x)$. Vector of right-neighbors is $\overrightarrow{N_X}(x)$. Neighbors wrap around the solution e.g. right neighbor of the last node is the first node.
For example $\overleftrightarrow{N_X}(1) = [3, 4, 4, 5, 4, 3]$ for $X = [3, 1, 4, 1, 5, 9, 2, 6, 5, 4, 1]$

- **ERX** (Edge Recombination Crossover) [WSF89] is a standard TSP and VRP operator. It treats the solutions as an undirected sequence of vertexes in a graph. The child should inherit as many edges from the parents as possible, and edges common to both parents should have priority. Our variation of ERX is described in the pseudo-code below.

Algorithm 7 $ERX(X, Y)$

1: $Z \leftarrow [x_1]$.
2: **for** $a \in A$ **do**
3: $\quad M_a \leftarrow \overleftrightarrow{N_X}(a) + \overleftrightarrow{N_Y}(a)$
4: **end for**
5: **while** true **do**
6: $\quad \beta \leftarrow z_{-1}$
7: $\quad S \leftarrow \{a \in M_\beta |\ f_a < u_a\}$
8: $\quad$ **if** $S = \emptyset$ **then**
9: $\quad\quad S \leftarrow \{a \in A |\ f_a < l_a\}$.
10: $\quad$ **end if**
11: $\quad$ **if** $S = \emptyset$ **then**
12: $\quad\quad$ break
13: $\quad$ **end if**
14: $\quad \Gamma \leftarrow \{\gamma \in S \mid |M_\gamma| \leq |M_a|\ \forall a \in S\}$
15: $\quad \gamma \leftarrow \text{random}(\Gamma)$
16: $\quad Z \leftarrow Z.\text{append}(\gamma)$
17: $\quad M_\beta \leftarrow M_\beta \backslash \gamma$
18: $\quad M_\gamma \leftarrow M_\gamma \backslash \beta$
19: **end while**

ERX first initializes the child with the first node of X (1) and then creates vectors of neighbors from both parents (2-4). Following steps are repeated until the stopping condition (11-13) is met. At every iteration, take the last node of the child $\beta = z_{-1}$ (5-6). Selects from β's neighbors set of nodes S which don't exceed the upper bound (7). If no neighbors

are below the upper bound, fill S with all nodes below the lower bound (8-10). If S is still empty stop the algorithm (11-13). Append a node from S with the fewest neighbors to the child. If there is more than one node in S choose one at random (14-16). Remove the nodes from each other's vectors of neighbors (17-18).

- **AEX** (Alternating Edge Crossover) [GGRVG14] is a standard TSP and VRP operator. It treats the solutions as a directed sequence of vertexes in a graph. The child should inherit a mix of edges from both parents, alternating between edges from the first and second parent. Our variation of AEX is described in the pseudo-code below.

Algorithm 8 $AEX(X, Y)$

1: $Z \leftarrow [x_1]$.
2: **while** true **do**
3: $\beta \leftarrow z_{-1}$
4: $S \leftarrow \{a \in \overrightarrow{N_X}(\beta) |\ f_a < u_a\}$
5: **if** $S = \emptyset$ **then**
6: $S \leftarrow \{a \in A |\ f_a < l_a\}$.
7: **end if**
8: **if** $S = \emptyset$ **then**
9: break
10: **end if**
11: $s \leftarrow \text{random}(S)$
12: $Z \leftarrow Z.\text{append}(s)$
13: $\overrightarrow{N_X}(\beta) \leftarrow \overrightarrow{N_X}(\beta) \backslash s$
14: $\text{Swap}(X,\ Y)$
15: **end while**

AEX first initializes the child with the first node of X (1). Following steps are repeated until the stopping condition (8-10) is met. At every iteration, take the last node of the child $\beta = z_{-1}$ (2-3). In the first parent, selects from β's right-neighbors set of nodes S which don't exceed the upper bound (4). If no neighbors are below the upper bound, fill S with all nodes below the lower bound (5-7). If S is still empty stop the algorithm (8-10). Append a random node from S to the child (11-12). Remove appended node from the β's vectors of right-neighbors (13). Swap the parents (14).

- **HX** (Heuristic Crossover) [GGRVG14] is a TSP and VRP operator similar to the AEX. It treats the solutions as a directed sequence of vertexes in a graph. The child inherits edges from both parents based on the fitness of the partial solution. Our variation of HX is described in the pseudo-code below.

Algorithm 9 $HX(X, Y)$

1: $Z \leftarrow [x_1]$.
2: **for** $a \in A$ **do**
3: $\quad \overrightarrow{M}_a \leftarrow \overrightarrow{N_X}(a) + \overrightarrow{N_Y}(a)$
4: **end for**
5: **while** true **do**
6: $\quad \beta \leftarrow z_{-1}$
7: $\quad S \leftarrow \{a \in \overrightarrow{M}_\beta |\ f_a < u_a\}$
8: $\quad$ **if** $S = \emptyset$ **then**
9: $\quad\quad S \leftarrow \{a \in A |\ f_a < l_a\}$.
10: $\quad$ **end if**
11: $\quad$ **if** $S = \emptyset$ **then**
12: $\quad\quad$ break
13: $\quad$ **end if**
14: $\quad s \leftarrow$ random(S) with probability distribution P
15: $\quad Z \leftarrow Z$.append(s)
16: $\quad \overrightarrow{M}_\beta \leftarrow \overrightarrow{M}_\beta \backslash s$
17: **end while**

HX first initializes the child with a first node of X (1) and then creates vectors of right-neighbors from both parents (2-4). Following steps are repeated until the stopping condition (11-13) is met. At every iteration, take the last node of the child $\beta = z_{-1}$ (5-6). Selects from β's right-neighbors set of nodes S which don't exceed the upper bound (7). If no neighbors are below the upper bound, fill S with all nodes below the lower bound (8-10). If S is still empty stop the algorithm (11-13). Append a random node $s \in S$ with probability P_s, given by one of the probability distributions described below, to the child (14-15). Remove appended node from the β's vectors of right-neighbors (16).

Three ways of defining the probability distribution are

- **HRndX:** Probability of choosing s is uniform. $P_s = \frac{1}{|S|}$.
- **HGreX:** Node s with the lowest fitness is chosen. Constructs $|S|$ temporary solutions $T_s \leftarrow Z$.append(s) $\forall s \in S$. Assign $P_s = 1$ to $s = arg\min_{s \in S} g_p(T_s)$, others are assigned 0.
- **HProX:** Probability of choosing s is proportional to its fitness. Construct the same temporary solutions T_s, then $P_s = \frac{g_p(T_s)}{\sum_{i \in S} g_p(T_i)}$

4.1.5 Mutation

Every child has the probability of *mutation_rate* of undergoing random mutation from a mutation list. The mutation list is provided by the user and

must contain at least one mutation operator. Next are listed all mutation operators. They are repurposed and randomized local search operators taken from the GMS-LS and are described in more detail at [Hra22, pg. 15-21]. The numerical parameters of the operators are defined by the user if not mentioned otherwise. By substrings we mean continuous parts of the solution that don't wrap around the ends of the solution.

$$X: \ \mathbf{3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3\ 3\ 5\ 8}$$

- **Insert mutation** inserts a random node $a \in A,\ f_a < u_a$ into a random position. For example, insert node 7 at position 2.

$$X': \ \mathbf{3\ \underline{7}\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3\ 3\ 5\ 8}$$

- **Append mutation** appends a random node $a \in A,\ f_a < u_a$ to the solution. For example, append node 7.

$$X': \ \mathbf{3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3\ 3\ 5\ 8\ \underline{7}}$$

- **Remove mutation** removes a node $a \in A,\ f_a > l_a$ at a random position. For example, remove node 3 at position 10.

$$X': \ \mathbf{3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ \_\ 3\ 5\ 8}$$

- **Relocate mutation** moves a random substring of length x to a random position. If parameter *reverse* is true it also reverses the substring. For example, move a substring from position 2 of length 3 ($[1, 4, 1]$) to position 5.

$$X': \ \mathbf{3\ 5\ 9\ 2\ \underline{1\ 4\ 1}\ 6\ 5\ 3\ 3\ 5\ 8}$$

- **Exchange mutation** swaps two random non-overlapping substrings of length x and y. If parameter *reverse* is true it also reverses the substrings. For example, exchange and reverse substrings at positions 2 and 8 of lengths 3 and 4 ($[1, 4, 1]$ and $[6, 5, 3, 3]$)

$$X': \ \mathbf{3\ \underline{3\ 3\ 5\ 6}\ 5\ 9\ 2\ \underline{1\ 4\ 1}\ 5\ 8}$$

- **Central exchange mutation** reverses a random substring of odd length x around its center node. For example, reverse string of length 2, centered at position 5 ($[4, 1, 5, 9, 2]$).

$$X': \ \mathbf{3\ 1\ \underline{2\ 9\ 5\ 1\ 4}\ 6\ 5\ 3\ 3\ 5\ 8}$$

- **Move all mutation** moves all occurrences of a random node by k positions. Nodes moved beyond the first/last position wrap around back to the end/beginning. For example, move node 3 by 4 positions to the right

$$X': \ \mathbf{1\ \underline{3}\ \underline{3}\ 4\ 1\ 5\ \underline{3}\ 9\ 2\ 6\ 5\ 5\ 8}$$

- **Exchange nodes mutation** swaps all occurrences of two random nodes if the swap doesn't break the lower/upper bounds. If parameter N is provided it swaps the first N occurrences of both nodes. For example, exchange nodes 1 and 5

$$X' : \mathbf{3\ \underline{5}\ 4\ \underline{5}\ \underline{1}\ 9\ 2\ 6\ \underline{1}\ 3\ 3\ \underline{1}\ 8}$$

- **Two-opt mutation** reverses random substring of random length. For example, reverse string at position 6 of length 6 ($[9, 2, 6, 5, 3, 3]$).

$$X' : \mathbf{3\ 1\ 4\ 1\ 5\ \underline{3\ 3\ 5\ 6\ 2\ 9}\ 5\ 8}$$

4.1.6 Replacement

To maintain the same number of solutions in the next generation we need a way to replace the old solutions with the new child solutions. Deterministic replacement combines all old and new solutions and then selects λ of the fittest solutions to continue to the next generation.

ASCHEA uses segregational replacement which is a combination of two deterministic replacements. First $t_{select}\lambda$ fittest solutions from the feasible solutions are promoted to the next generation, where t_{select} is a user-defined proportion. The rest of the next generation is then filled using standard deterministic replacement with the remaining individuals based on their current penalized fitness. Duplicates of solutions are not allowed in the population with the intent to increase the population's diversity.

We added a third deterministic replacement between the two original replacements. It triggers only if the population contains no feasible individuals. For every problem constraint, the fittest solution that doesn't violate said constraint is promoted to the next generation. This should have the effect of keeping the genetic information, which allows a solution to pass certain constraint, in the population.

Niching

To better handle multimodal functions (functions with several local optima) niching [Pet96] has been added to the replacement operator in the second

version of ASCHEA. A niche is defined as a ball centered on an individual with a user-defined radius r. Each niche contains some dominant individuals, called leaders, that include its center. The maximum number of leaders allowed in a niche is another user-defined parameter called the niche capacity. When a niche has reached its full capacity, other individuals that fall within that niche, called followers, will be discarded from further selections. Levenshtein distance [L+66] was used to compute the distance between solutions, because it can calculate the distance between solutions of different sizes.

Once all individuals are classified, the segregational replacement is first applied to the leaders only. If there are less than $t_{select}\lambda$ feasible individuals or not enough individuals to complete the population it is applied in a second step to the followers. To avoid manual tuning of the niche radius r, the adaptive procedure similar to the population-level adaptive penalty is used

$$r(t+1) = \begin{cases} r/fact, & \text{if } \tau_{leaders} > \tau_{clear}, \\ r(t) \times fact, & \text{if } \tau_{clear} > \tau_{leaders}, \end{cases} \tag{4.11}$$

where t_{clear} is the desired proportion of leaders in the population defined by the user (usually 0.4), $\tau_{leaders}$ and $\tau_{followers}$ are the proportion of leaders and followers in the population at generation t and $fact$ is the same as in the penalty adaptation function.

Since niching adds computational overhead that scales $O(n^2)$ with both the length of the solution and population size, we give the user a choice of disabling the niching process and using only the standard segregational replacement.

Niching is described in Algorithm 10. It iterates over all solutions in the population (3). If the solution X isn't a follower (4), it is designated as a leader of a niche (5). If some other solution Y, not belonging to followers or leaders, is closer to X than distance r it belongs to the X's niche (7-8). If the niche has less than *niche_capacity* leaders Y is classified as a leader, otherwise as a follower (9-12).

Algorithm 10 *Niche(population)*

```
sort population by decreasing fitness
leaders, followers = {}
for i = 0 to μ − 1 do
    if X_i ∉ followers then
        leaders ← leaders ∪ X_i
        nbLeaders = 1
        for j = i + 1 to μ − 1 do
            if X_j ∉ followers and distance(X_i, X_j) < r then
                if nbLeaders < niche_capacity then
                    nbLeaders ← nbLeaders + 1
                else
                    followers ← followers ∪ X_j
                end if
            end if
        end for
    end if
end for
```

4.1.7 Population sizing scheme

One of the most important parameters in EAs is the size of the population λ. To avoid possible problems with early population conversion or stagnation we choose to implement a population sizing method from a parameter-less genetic algorithm [HL+99].

First, we need to define what is an average fitness of a population and a stagnant population. Average fitness g_{avg} of population $Pop = [X_1, ..., X_n]$ is average fitness of its feasible solutions

$$g_{avg}(Pop) = \begin{cases} \frac{\sum_{X \in Pop} g_p(X) [\![X \in \mathbb{F}]\!]}{\sum_{X \in Pop} [\![X \in \mathbb{F}]\!]}, & \text{if } \sum_{X \in Pop} [\![X \in \mathbb{F}]\!] > 0, \\ \infty, & \text{otherwise.} \end{cases} \tag{4.12}$$

The population is stagnant if solutions in generation t are the same as solutions in generation $t - 1$.

The sizing scheme maintains a list $L = [Pop_1, Pop_2, ...]$ of increasingly larger populations. Population Pop_{i+1} is twice as large as population Pop_i. Since it takes a larger population longer to complete one generation, more evaluations are awarded to smaller populations. We start with a population Pop_1 of size eight ($|Pop_1| = 8$) and a counter in user-defined base n set to zero. Every cycle the counter is incremented by one and the largest flipped digit determines which population from the list is run. An example of a run

counter	population	generation
1	1	1
2	1	2
3	1	3
10	2	1
11	1	4
12	1	5
13	1	6
20	2	2
...	...	...
33	1	12
100	3	1
101	1	13

Table 4.1: populations sizing example

with the counter in base 4 is given in Table 4.1. If the population hasn't run before, it is initialized before being run for one generation. If a population has higher average fitness than a bigger population or is stagnant, it is removed from the list of populations.

What this means, is that several instances of ASCHEA are run in parallel, each with its own population, penalty coefficients etc. The operators and other static parameters ($\tau_{target}, \tau_{select}$) are the same for all populations. The best solution from all populations is returned as a result of the search.

The sizing scheme can be turned off by the user by setting parameter *populatiton_type* = "*static*" and setting *population_size* to desired integer value. In this mode, if the population stagnates, the p fittest individuals are kept and the rest is discarded. The remaining empty space is filled with solutions initialized by the user's chosen operator.

4.2 Problem implementation

In this section, we outline the implementation of EVRP and ROADEF problems in the context of GMS. Mainly the lower and upper bounds LB and UB, a conversion between the solver's representation X and problem-specific solution, fitness function g and penalty vector P. Fitness function g consists of actual fitness p_0, penalized sum of constraint violations $p_1, ..., p_p$. Fitness g is used in Greedy and Nearest-Neighbor initializations and for compatibility

with the GMS-LS.

Electric Vehicle Routing Problem

Let us assume that we are given distance matrix $C = (c_{ij}) \in \mathbb{R}^{n \times n}$, number of vehicles k, depot node D, a set of charging stations S of size s, list of customer demands B of size l ($1 + l + s = n$ must hold) and maximal carrying capacity and charge C and Q. Permutation with repetition $X = [x_1, ..., x_m]$ used in the solver is directly translated to a sequence of visited customers, chargers and depot. The problem class is initialized by the following

$$A = \{1, ..., n\} \tag{4.13}$$

$$LB = [k + 1, 1..., 1, 0, ..., 0] \tag{4.14}$$

$$UB = [k + 1, 1..., 1, 2l, ..., 2l] \tag{4.15}$$

$$D = 1 \tag{4.16}$$

$$B = [b_2, ..., b_{l+1}] \tag{4.17}$$

$$S = \{n - s + 1, ..., n\} \tag{4.18}$$

$$p_0 = \sum_{i=1}^{m-1} c_{x_i, x_{i+1}} \tag{4.19}$$

$$p_1 = \sum_{a \in A \setminus (S \cup \{D\})} |1 - f_a| \tag{4.20}$$

$$p_2 = [\![x_1 \neq D]\!] + [\![x_m \neq D]\!] \tag{4.21}$$

$$p_3 = |k - (f_D - 1)| \tag{4.22}$$

$$p_4 = \sum_{a \in A} b_a [\![b_a > u_a]\!] \tag{4.23}$$

$$p_5 = \sum_{i=1}^{m-1} \max(0, c_{x_i, x_{i+1}} - \frac{y_i}{h}) \tag{4.24}$$

$$g(X) = p_0 + \sum_{i=1}^{5} M_i p_i \tag{4.25}$$

where A is the set of nodes including depot a_1, customers $a_2, ..., a_{l+1}$, and charging stations $a_{l+2}, ..., a_n$. Upper and lower bounds $l_i = 1$ and $u_i = 1$ for $i \in \{2, .., l + 1\}$ and $l_j = 0$ and $u_j = 2l$ for $j \in \{l + 2, .., n\}$. Pure fitness p_0 is a sum of the total traveled distance. The first constraint p_1 ensures that all customers are visited exactly once. Second constraint p_2 checks if the solution starts and ends at the depot. Third constraint p_3 counts the

number of tours starting and ending at the depot. Fourth constraint p_4 sums unsatisfied customers' demand, where b_a is the demand of customer a and u_a is the maximal demand the vehicle is able to satisfy when it arrives at node a. Fifth constraint p_5 sums distance traveled on an empty battery. From available charge y_a at node a, we are able to determine vehicles range $\frac{y_i}{h}$. If the range doesn't exceed the distance between the nodes then the distance traveled on an empty battery is added to the penalty. In fitness function g symbols $M_1, ..., M_5$ are large constants defined by the programmer.

4.2.1 ROADEF

Let us assume we are given all values as described in ROADEF input section, which include planning horizon $H = \{1, ...T\}$, set of resources C and their bounds l_t^c, u_t^c, set of interventions I and their time duration $\Delta_{i,t}$ and workload $r_{i,t'}^{c,t}$. Next, we are given risk assessment of different scenarios $risk_{i,t}^{s,t}$, a set of exclusions Exc from disjunctive constraints and constants $\alpha, \tau \in [0, 1]$ needed in the final objective. The problem class is initialized by the following

$$A = \{1, ..., |I| + 1\} \tag{4.26}$$

$$LB = [1, ..., 1, 0] \tag{4.27}$$

$$UB = [1..., 1, T] \tag{4.28}$$

$$p_0 = \alpha obj_1 + (1 - \alpha) obj_2(\tau) \tag{4.29}$$

$$p_1 = \sum_{c \ \in C} \sum_{t \in H} \max(0, \ l_t^c - r_{tot}(c, t)) \tag{4.30}$$

$$p_2 = \sum_{c \ \in C} \sum_{t \in H} \max(0, \ r_{tot}(c, t) - u_t^c) \tag{4.31}$$

$$p_3 = \sum_{t \in H} \sum_{i_1 \in I} \sum_{\substack{i_2 \in I \\ i_1 < i_2}} [\![(i_1, i_2, t) \in Exc]\!] \tag{4.32}$$

$$g(X) = p_0 + \sum_{i=1}^{3} M_i p_i \tag{4.33}$$

where A is the set of nodes representing interventions $i_1, ..., i_{|I|}$ and a fill node $|I| + 1$. Lower and upper bounds of interventions are set to one. Fill node may be present up to T times. Pure fitness p_0 is derived directly from (3.36). The first constraint p_1 is the resource underuse. Function $r_{tot}(c, t)$ defined in (3.26) returns a total use of a resource c at a time t. If the total use of a resource at certain time is below its lower bound, the difference is

added to the resource underuse p_1. The second constraint p_2 is the resource underuse and functions the same as resource underuse. Third constraint p_3 counts the number of unmet exclusions. In fitness function g symbols $M_1, ..., M_3$ are large constants defined by the programmer.

Before fitness and constraint violations are computed we need to convert the solution X to a schedule list L of pairs $(i, t) \in I \times H$. Before solving the problem, interventions I are assigned nodes $a_1, ...a_{|I|}$, node $a_{|I|+1}$ is reserved as a filler node.

Algorithm 11 *Permutation_to_schedule*(X)

```
1: t ← 1.
2: L ← []
3: for i ∈ X do
4:     if i = |I| + 1 then
5:         t ← t + 1
6:     else
7:         L.append((i, t))
8:     end if
9: end for
```

First we set time $t = 1$ (1) and empty the schedule $L = []$ (2). We iterate over the solution X. (3) If node i is a filler node then time t is increased by 1 (4-5), otherwise its corresponding intervention i staring at time t is appended to L (6-7).

Chapter 5

Results

The GMS and both problem-specific solvers are implemented in C++. All experiments mentioned below ran on a single thread. We performed two kinds of experiments. First, performance comparison experiments, comparing our GMS-CO to the GMS-LS and a problem-specific solver. Second, operator comparison experiments, comparing proposed crossovers against each other. All GMS parameters were selected by the Irace parameter tuning tool [LIDLC+16]. Operator comparison experiments used the same parameters as the performance comparison experiment, but instead of a mixture of crossovers only a single fixed crossover operator was used.

All experiments were run on MetaCentrum's [met] distributed computing clusters. The operator comparison experiments and tuning of parameters were run across all available clusters due to the volume of computation needed. All performance comparison experiments were run on a Nympha cluster to guarantee better consistency. The cluster runs the Linux Debian 11 operating system, 16 GB of RAM and per-core performance in SPECfp2017 benchmark [SPE] of SPECfp2017=3.7 or SPECfp2017=3.8. Since our solver is stochastic all experiments were run $R = 50$ times, each with a random seed. Statistics were calculated only from feasible solutions $\mathbb{X}$. They include a percentage $\overline{\mathbb{F}}$ of feasible solutions found, mean $\overline{g}$, standard deviation $\overline{\sigma}$ and percentage difference $\overline{GAP}$ of the proposed solver's average fitness w.r.t. best-known solution BKS.

$$\overline{\mathbb{F}} = 100 \, \frac{|\mathbb{X}|}{R} \tag{5.1}$$

$$\overline{g} = \frac{\sum_{X \in \mathbb{X}} g(X)}{|\mathbb{X}|} \tag{5.2}$$

$$\overline{\sigma} = \sqrt{\frac{\sum_{X \in \mathbb{X}} (\overline{g} - g(X))^2}{|\mathbb{X}| - 1}} \tag{5.3}$$

$$\overline{GAP} = 100 \, \frac{\overline{g} - BKS}{BKS} \tag{5.4}$$

The values of BKS were taken from the relevant literature. The timeout for the experiments, $t(min)$ in minutes, is specified in the following sections.

5.1 ROADEF

The dataset used in the experiment is dataset 'A', which was used for the qualification round of the ROADEF competition. The timeouts were set at 15 and 90 minutes, the same as in the ROADEF competition. The best-known solutions were taken from the results of the qualification round.

instance	A01	A02	A03	A04	A05	A06	A07	A08
T	90	90	90	365	182	182	17	17
$\|I\|$	181	89	91	706	180	180	34	18
$\|Exc\|$	81	32	12	1377	87	87	4	4
instance	A09	A10	A11	A12	A13	A14	A15	
T	17	52	52	52	90	52	52	
$\|I\|$	18	108	54	54	179	108	108	
$\|Exc\|$	0	40	4	0	136	22	22	

Table 5.1: parameters of ROADEF instances

5.1.1 Performance comparison experiments

After some technical difficulties with most of the problem-specific solvers, we choose the solver by David Woller and Jakub Rada [RW22], which is based on the Adaptive Large Neighborhood Search (ALNS) [MNJ+22]. The

parameters for GMS-CO and GMS-LS found by the Irace tuning tool, with a tuning budget of 10000 problem runs, are listed below.

- **GMS-CO parameters:**
  - **Population:**
    - **Type:** dynamic
    - **Parameters:** $\tau_{select} = 0.2943$, $\tau_{target} = 0.6861$, $counter_{base} = 26, tournament_{size} = 18$
  - **Construction:** random
  - **Crossover:**
    - **Alignment:** randomUniform
    - **Operators:** ERX, OBX, ULX, RULX, UPMX, MPX
  - **Mutation:**
    - **Rate:** 0.9355
    - **Operators:** insert, exchangeNIds, centralExchange(1), centralExchange(2), centralExchange(4), relocate(1),
  - **Replacement:** segregational
- **GMA-LS parameters:**
  - **Metaheuristic:** BVNS ($k_{min} = 7,\ k_{max} = 11$)
  - **Local search:** PVND
  - **Construction:** randomReplicate
  - **Perturbation:** randomMove
  - **Operators:** insert, two-opt, exchangeIds, exchangeNIds, exchange(1,1), exchange(2,4), exchange(3,3), exchange(4,4), reverseExchange(2,3), reverseExchange(3,3), reverseExchange(4,4), centralExchange(3), centralExchange(4), centralExchange(5), relocate(5), moveAll(1), moveAll(2)

GMS-LS								
	$t(min) = 15$				$t(min) = 90$			
Instance	$\overline{g}$	$\overline{\sigma}$	$\overline{GAP}$	$\overline{\mathbb{F}}$	$\overline{g}$	$\overline{\sigma}$	$\overline{GAP}$	$\overline{\mathbb{F}}$
A01	∅	∅	∅	0%	∅	∅	∅	0%
A02	∅	∅	∅	0%	2758495.1	15809901	58955.8%	80%
A03	∅	∅	∅	0%	915.8	26.1	8%	96%
A04	∅	∅	∅	0%	∅	∅	∅	0%
A05	∅	∅	∅	0%	∅	∅	∅	0%
A06	∅	∅	∅	0%	∅	∅	∅	0%
A07	2274	0.3	0.04%	100%	2273.7	0.5	0.02%	100%
A08	745	0	0.1%	100%	745	0.2	0.1%	100%
A09	1508	0	0.1%	100%	1507.7	0.5	0.04%	100%
A10	3490.6	873.7	16.5%	100%	3060.7	17	2.2%	100%
A11	7739.2	6662.6	1463.5%	100%	524.1	20	5.9%	100%
A12	1151.4	44.2	45.8%	100%	957	100.1	21.1%	100%
A13	∅	∅	∅	0%	∅	∅	∅	0%
A14	∅	∅	∅	0%	∅	∅	∅	0%
A15	∅	∅	∅	0%	∅	∅	∅	0%
mean	∅	∅	254.3%	40%	∅	∅	7374,15%	52%

Table 5.2: ROADEF results with 15 and 90 minute timeouts using GMS-LS

GMS-CO								
	$t(min) = 15$				$t(min) = 90$			
Instance	$\overline{g}$	$\overline{\sigma}$	$\overline{GAP}$	$\overline{\mathbb{F}}$	$\overline{g}$	$\overline{\sigma}$	$\overline{GAP}$	$\overline{\mathbb{F}}$
A01	1920	54.4	8.6%	100%	1808.5	12.9	2.3%	100%
A02	4704.7	11.6	0.7%	100%	4704.4	12.3	0.7%	100%
A03	879.4	13.4	3.7%	100%	859.6	5.4	1.4%	100%
A04	∅	∅	∅	0%	∅	∅	∅	0%
A05	759.3	32.2	19.6%	100%	683.8	14.4	7.7%	100%
A06	775.7	25.9	31.2%	96%	693.1	19.3	17.3%	100%
A07	2274.8	1.3	0.1%	100%	2274	0.5	0.04%	100%
A08	745.6	0.5	0.2%	100%	745.4	0.5	0.2%	100%
A09	1507	0.1	0%	100%	1507	0	0%	100%
A10	3022.9	11.7	0.9%	100%	3014.1	10.3	0.6%	100%
A11	497.7	1	0.5%	100%	497.5	0.8	0.5%	100%
A12	869	107.4	10%	100%	833.1	95.6	5.5%	100%
A13	2428.6	970.5	21.5%	100%	2040.3	57.1	2.1%	100%
A14	2452.3	96.6	8.3%	50%	2336.6	32.6	3.2%	94%
A15	2498.8	125.5	10.1%	40%	2351.3	45.8	3.6%	88%
mean	∅	∅	8.14%	86%	∅	∅	3.21%	92%

Table 5.3: ROADEF results with 15 and 90 minute timeouts using GMS-CO

problem-specific solver								
	$t(min) = 15$				$t(min) = 90$			
Instance	$\overline{g}$	$\overline{\sigma}$	$\overline{GAP}$	$\overline{\mathbb{F}}$	$\overline{g}$	$\overline{\sigma}$	$\overline{GAP}$	$\overline{\mathbb{F}}$
A01	1769.5	0.6	0.09%	100%	1768.6	0.3	0.04%	100%
A02	4671.5	0.4	0.0%	100%	4671.4	0.2	0.0%	100%
A03	848.2	0.0	0.0%	100%	848.2	0.0	0.0%	100%
A04	2105.8	6.4	0.95%	100%	2094.3	1.5	0.4%	100%
A05	636	0.2	0.1%	100%	635.7	0.2	0.1%	100%
A06	593.7	1.3	0.5%	100%	592.1	0.8	0.2%	100%
A07	2272.8	0.0	0.0%	100%	2272.8	0.0	0.0%	100%
A08	744.3	0.0	0.0%	100%	744.3	0.0	0.0%	100%
A09	1507.3	0.0	0.0%	100%	1507.3	0.0	0.0%	100%
A10	2994.9	0.0	0.0%	100%	2994.9	0.0	0.0%	100%
A11	495.3	0.1	0.0%	100%	495.3	0.0	0.0%	100%
A12	789.6	0.0	0.0%	100%	789.6	0.0	0.0%	100%
A13	1999.3	0.2	0.03%	100%	1999.1	0.2	0.02%	100%
A14	2286.2	12.5	0.97%	100%	2274.2	10.3	0.4%	100%
A15	2297.4	14.4	1.2%	100%	2286.4	14.4	0.8%	100%
mean	$\emptyset$	$\emptyset$	0.26%	100%	$\emptyset$	$\emptyset$	0.13%	100%

Table 5.4: ROADEF results with 15 and 90 minute timeouts using problem-specific solver

As seen in Table 5.2, the GMS-LS managed to find feasible solutions in under 15 minutes to instances A07,..., A12 whose time horizon is $T \leq 52$ and the number of exclusions is generally low. With 90 minute timeout, it also found a feasible solution to instances A02 and A03 with $T = 90$.

The problem-specific solver unsurprisingly performed best on all instances across all metrics, which can be seen in Table 5.4. It is also the only solver that found a feasible solution to instance A04, which is also the instance that occupied most memory while being solved. Both are probably caused by the large time horizon $T = 365$, the number of interruptions $|I| = 706$ and the number of exclusions $|Exc| = 1377$ of this instance, as can be seen in Table 5.1.

The instances A05 and A06 have the second largest time horizon $T = 182$, and those are also instances in which the proposed solver performed the worst. For harder instances with longer time horizons $T \geq 90$ the GMS-CO's $\overline{GAP}$ improved significantly with the increased 90-minute timeout as seen in Table 5.3, suggesting that the solver's performance could improve further with an even longer timeout. This is supported by Figure 5.1 where the average last improvements occur towards the end of the timeout. In the graph of instance

A15 we can see that the average first feasible solution found is around the 25-minute mark. This would corelate with low feasibility $\overline{\mathbb{F}} = 0.4$ for 15 minute timeout and $\overline{\mathbb{F}} = 0.88$ for 90 minute timeout. The spikes in the graph of instance A15 correspond to times when the first feasible solutions were frequently found.

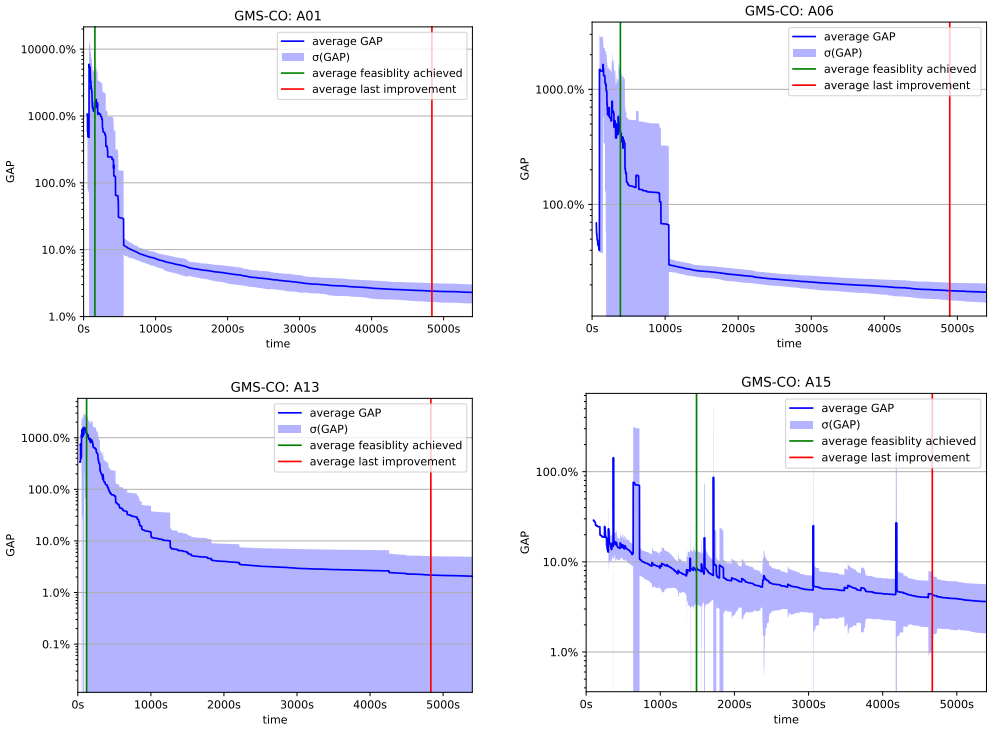


Figure 5.1: ROADEF fitness minimization steps of four different instances.

5.1.2 Operator comparison

If we disregard the A04 instance, we can see in Figures 5.2 and 5.3 that half of the operators found at least some valid solutions to all instances in the 15-minute timeout and 10 found feasible solutions in the 90-minute timeout. Best performing operators were APX, CX, NBX, OBX, RULX, ULX and UPMX, finding at least 90% of all feasible solutions in 90 minutes. From those ULX seems as the best single operator for solving this problem. However, all of them performed sub-optimally in the A02, A05, A06 and A13 instances. GMS-CO performed at most 200 times better than the best-performing individual operators depending on the instance. This suggests that the interplay between different crossovers is crucial for effectively solving this problem.

t(min)=15

	feasibility																GAP														
15m	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	avg	15m	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	avg
AEX	0	74	96	0	2	0	100	100	100	100	100	100	0	26	10	53,87	AEX		642	98		554		0	0	2	4	9	29		205	295	167,09
APX	98	100	100	0	100	100	100	100	100	100	100	100	100	54	38	86,00	APX	4	1824	4		526	68	0	0	0	1	2	30	81	4	6	182,14
CX	100	100	100	0	98	100	100	100	100	100	100	100	100	54	38	86,00	CX	4	1932	4		694	31	0	0	0	1	3	30	76	6	7	199,14
ERULX	0	2	0	0	0	0	100	100	100	94	100	100	0	0	0	39,73	ERULX		5310					1	0	0	41	1211	37				942,86
ERX	0	0	0	0	0	0	100	100	100	100	100	100	0	0	0	40,00	ERX							10	13	6	10	210	47				49,33
EULX	0	52	74	0	0	0	100	100	100	100	100	100	0	0	0	48,40	EULX		4028	914				0	0	0	4	170	27				642,88
HGrX	0	0	0	0	0	0	100	100	100	44	100	100	0	0	0	36,27	HGrX							10	8	6	106	2228	47				400,83
HPrX	0	0	0	0	0	0	100	100	100	86	100	100	0	0	0	39,07	HPrX							3	0	3	49	1669	44				294,67
HRnX	0	100	100	0	38	0	100	100	100	100	100	100	0	0	0	55,87	HRnX		52	20		351		0	0	0	3	3	19				49,78
MPX	0	0	0	0	0	0	100	100	100	100	100	100	0	0	0	40,00	MPX						1	0	1	3	532	46					97,17
NBX	100	100	100	0	98	98	100	100	100	100	100	100	100	38	22	83,73	NBX	4	2420	4		553	127	0	0	0	1	3	28	106	91	3	238,57
OBX	100	100	100	0	94	96	100	100	100	100	100	100	98	44	24	83,73	OBX	4	2303	4		755	46	0	0	0	1	3	25	136	6	5	234,86
OX	0	88	98	0	0	0	100	100	100	100	100	100	0	8	8	53,47	OX		3277	10				0	0	0	1	7	43		18	21	337,70
PBX	0	100	100	0	0	0	100	100	100	100	100	100	0	4	2	53,73	PBX		2998	7				0	0	0	1	4	39		19	9	307,70
RULX	100	100	100	0	92	96	100	100	100	100	100	100	100	32	16	82,40	RULX	17	2653	4		652	84	0	0	0	1	2	30	89	8	6	253,29
SPX	8	96	98	0	0	4	100	100	100	100	100	100	6	6	16	55,60	SPX	863	3183	8			1175	0	0	0	2	62	30	3765	21	18	702,08
ULX	100	100	100	0	100	92	100	100	100	100	100	100	100	84	80	90,40	ULX	4	702	2		349	28	0	0	0	1	1	21	26	4	4	81,57
UPMX	100	100	100	0	94	96	100	100	100	100	100	100	100	28	22	82,67	UPMX	5	2184	4		588	31	0	0	0	1	2	32	90	7	50	213,86

Figure 5.2: $\overline{\mathbb{F}}$ and $\overline{GAP}$ of ROADEF instances by operator with 15-minute timeout

t(min)=90

	feasibility																GAP															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	avg		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	avg
AEX	0	96	98	0	2	0	100	100	100	100	100	100	0	70	50	61,07	AEX		156	10		67		0	0	0	3	6	25		97	23	35,18
APX	100	100	100	0	100	100	100	100	100	100	100	100	100	100	98	93,20	APX	1	546	0		38	20	0	0	0	1	1	29	23	1	1	47,21
CX	100	100	100	0	100	100	100	100	100	100	100	100	100	100	100	93,33	CX	1	233	0		54	20	0	0	0	1	4	26	15	1	1	25,43
ERULX	0	12	40	0	0	100	100	100	100	100	100	0	0	0	0	43,47	ERULX		3618	237			287	0	102	121	0	109					559,25
ERX	0	0	0	0	0	0	100	100	100	100	100	100	0	2	4	40,40	ERX							6	10	5	6	16	46		198	1598	235,63
EULX	0	96	100	0	0	0	100	100	100	100	100	100	0	2	2	53,33	EULX		3236	57				0	0	0	2	3	19		25	14	335,60
HGrX	0	0	0	0	0	0	100	100	100	100	100	100	0	0	0	40,00	HGrX							2	4	1	17	395	44				77,17
HPrX	0	4	2	0	0	0	100	100	100	100	100	100	0	0	0	40,40	HPrX		1924	48				1	0	1	6	260	34				284,25
HRnX	0	100	100	0	100	0	100	100	100	100	100	100	0	0	0	60,00	HRnX		17	17		105		0	0	0	2	1	12				17,11
MPX	0	10	32	0	0	0	100	100	100	100	100	100	0	4	4	43,33	MPX		3353	1308				0	0	0	2	8	47		20	19	475,70
NBX	100	100	100	0	100	100	100	100	100	100	100	100	100	98	94	92,80	NBX	1	933	1		39	21	0	0	0	1	2	30	27	1	2	75,57
OBX	100	100	100	0	98	100	100	100	100	100	100	100	100	92	96	92,40	OBX	1	291	1		41	20	0	0	0	1	2	28	20	1	1	29,07
OX	32	100	100	0	2	12	100	100	100	100	100	100	20	66	50	65,47	OX	1377	2815	5		3761	350	0	0	0	1	8	42	1744	15	11	723,50
PBX	70	100	100	0	30	48	100	100	100	100	100	100	62	50	32	72,80	PBX	676	2933	4		839	148	0	0	0	1	5	40	912	7	9	398,14
RULX	100	100	100	0	100	100	100	100	100	100	100	100	100	92	96	92,53	RULX	1	716	1		58	21	0	0	0	1	2	28	38	1	2	62,07
SPX	86	100	100	0	50	44	100	100	100	100	100	100	84	34	34	75,47	SPX	522	2932	4		1517	309	0	0	0	2	4	27	315	6	6	403,14
ULX	100	100	100	0	100	100	100	100	100	100	100	100	100	100	100	93,33	ULX	1	116	1		32	20	0	0	0	1	1	22	2	1	1	14,14
UPMX	100	100	100	0	100	100	100	100	100	100	100	100	92	96	92,53	UPMX	1	642	1		39	21	0	0	0	1	3	28	20	1	1	54,14	

Figure 5.3: $\overline{\mathbb{F}}$ and $\overline{GAP}$ of ROADEF instances by operator with 90-minute timeout

5.2 EVRP

The dataset used in the experiment is the same as in the EVRP competition. The number after the letter 'n' in the names of the instances refers to the number of customers and the number after the letter 'k' to the number of vehicles available. The best-known solutions were taken from the best results found in the competition. The competition timeout was set at a number of fitness function evaluations. To obtain a real-time timeout t in minutes we used an equation from a previously proposed EVRP solver [JMZ21],

$$t(min) = \beta \frac{|I| + |F|}{100} 60, \tag{5.5}$$

where I is a set of customers, F is a set of charging stations and β is a scaling factor. For 'E' instances $\beta = 1$, for bigger 'X' instances $\beta = 2$.

5.2.1 Performance comparison experiments

The winning solution from the EVRP competition [KWV21] was used as the problem-specific solver. Due to technical difficulties with the tuning software, we were only able to run 5000 problems for each GMS, half of the tuning budget that was used on the ROADEF problem. The GMS-CO and GMS-LS configurations are listed below.

- **GMS-CO parameters:**
  - **Population:**
    - Type: dynamic
    - Parameters: $\tau_{select} = 0,3$, $\tau_{target} = 0,5$, $counter_{base} = 18$
  - **Construction:** nearestNeighbor
  - **Crossover:**
    - Alignment: greedyOneGap
    - Operators: NBX, PBX, ERX, AEX, OX, OBX, CX, HGreX, HRndX, HProX
  - **Mutation:**
    - Rate: 0.4410
    - Operators: insert, remove, twoOpt, exchangeIds, exchangeNIds
  - **Replacement:** segregational
- **GMS-LS parameters:**
  - **Metaheuristic:** ILS ($k = 8$)
  - **Local search:** PVND
  - **Construction:** greedy
  - **Perturbation:** randomMoveAll
  - **Operators:** insert, two-opt, exchangeIds, exchange(1,1), exchange(2,3), exchange(2,4), exchange(3,4), exchange(4,4), reverseExchange(3,3), reverseExchange(3,4), centralExchange(1), centralExchange(2), centralExchange(3), relocate(1), reverseRelocate(2), moveAll(2), modeAll(3), modeAll(4), modeAll(10),

		GMS-LS				GMS-CO			
Instance	$t(min)$	$\overline{g}$	$\overline{\sigma}$	$\overline{GAP}$	$\overline{\mathbb{F}}$	$\overline{g}$	$\overline{\sigma}$	$\overline{GAP}$	$\overline{\mathbb{F}}$
E-n22-k4	17	385	0	0%	100%	385.2	1	0.1%	100%
E-n23-k3	18	572	0	0%	100%	572	0	0%	100%
E-n30-k3	21	509	0	0%	100%	509	0	0%	100%
E-n33-k4	23	840	0	0%	100%	844.06	5.7	0.5%	100%
E-n51-k5	33	530.2	0.9	0.04%	100%	549.6	13.6	3.7%	100%
E-n76-k7	49	695	1.4	0.3%	100%	719.8	13.2	3.9%	100%
E-n101-k8	65	843.3	6.3	0.5%	100%	913.8	125	8.9%	100%
X-n143-k7	175	16312	119.3	1.8%	100%	17444.2	373.9	8.8%	100%
X-n214-k11	266	11802.1	240.8	4.2%	100%	19268.6	1189.5	70.2%	96%
X-n351-k40	462	29222.8	474.1	8%	100%	∅	∅	∅	0%
X-n459-k26	573	26062.4	140.2	2.7%	100%	53925.6	2508.3	112.5%	16%
X-n573-k30	694	52875.8	185.7	1.3%	100%	73719.1	1799.8	41.2%	92%
X-n685-k75	851	∅	∅	∅	0%	∅	∅	∅	0%
X-n749-k98	934	∅	∅	∅	0%	∅	∅	∅	0%
X-n819-k171	1011	∅	∅	∅	0%	∅	∅	∅	0%
X-n916-k207	1109	∅	∅	∅	0%	∅	∅	∅	0%
X-n1001-k43	1253	185730	10653.3	139.7%	76.4%	203294.5	12208	162.4%	8%
mean	∅	∅	∅	12%	75.1%	∅	∅	22.7%	59.6%

Table 5.5: EVRP results using GMS-LS and GMS-CO

problem-specific solver					
Instance	$t(min)$	$\overline{g}$	$\overline{\sigma}$	$\overline{GAP}$	$\overline{\mathbb{F}}$
E-n22-k4	17	385	0.0	0.0%	100%
E-n23-k3	18	572	0.0	0.0%	100%
E-n30-k3	21	509	0.0	0.0%	100%
E-n33-k4	23	840	0.0	0.0%	100%
E-n51-k5	33	530	0.0	0.0%	100%
E-n76-k7	49	693	0.0	0.0%	100%
E-n101-k8	65	834.7	0.7	-0.5%	100%
X-n143-k7	175	15888.7	4.2	-0.9%	100%
X-n214-k11	266	11133.2	18.6	-1.7%	100%
X-n351-k40	462	26582.8	330.4	-1.8%	100%
X-n459-k26	573	24763.6	48.3	-2.4%	100%
X-n573-k30	694	51496.6	69	-1.3%	100%
X-n685-k75	851	69624.3	1266.9	-2.4%	100%
X-n749-k98	934	79220.1	1707.6	-2.2%	100%
X-n819-k171	1011	160657.3	5796	-2.2%	100%
X-n916-k207	1109	336108.2	447.9	-1.6%	100%
X-n1001-k43	1253	75499.3	208.9	-2.5%	100%
mean	∅	∅	∅	-1.15%	100%

Table 5.6: EVRP results using problem-specific solver

As seen in Table 5.6, the problem-specific solver unsurprisingly performed best on all instances across all metrics. It also found new best-known solutions to instances E-n101-k8,..., X-n1001-k43. This is probably because it was able to run for a longer time than in the EVRP competition. As seen in Table 5.5, the GMS-LS performed well on instances with less than 600 customers, finding a feasible solution with good fitness on average. For larger instances with more than 50 vehicles, it failed to find any feasible solutions, but it managed to find feasible solutions for the largest instance, X-n1001-k43, with large $\overline{GAP}$ over 100%.

In Table 5.5 we can see that in the smaller instances (E-n22-k4,..., X-n143-k7) the proposed solver achieved feasibility $\overline{\mathbb{F}}$ of 100% and $\overline{GAP}$ bellow 10%. It found feasible solutions with large $\overline{GAP}$ for medium instances (X-n214-k11,..., X-n573-k30) except for instance X-n351-k40, where it failed to find any solutions. GMS-CO failed to find any feasible solutions for larger instances (X-n685-k75,...) except for instance X-n1001-k43, for which it found 8% of feasible solutions. The poor results may be a result of not using a high enough tuning budget. Although, the GMS-LS had the same tuning budget and it performed significantly better. Another reason for the poor performance might be that none of the implemented crossover operators are suitable for this kind of problem. A solution might also be to let the solver run longer. As we see in figure 5.4 the GMS-CO was improving on the solutions in the X-n214-k11 and X-n459-k26 instances until the very end of its run. This slow conversion could be caused by poor scalability with the increased number of vehicles.

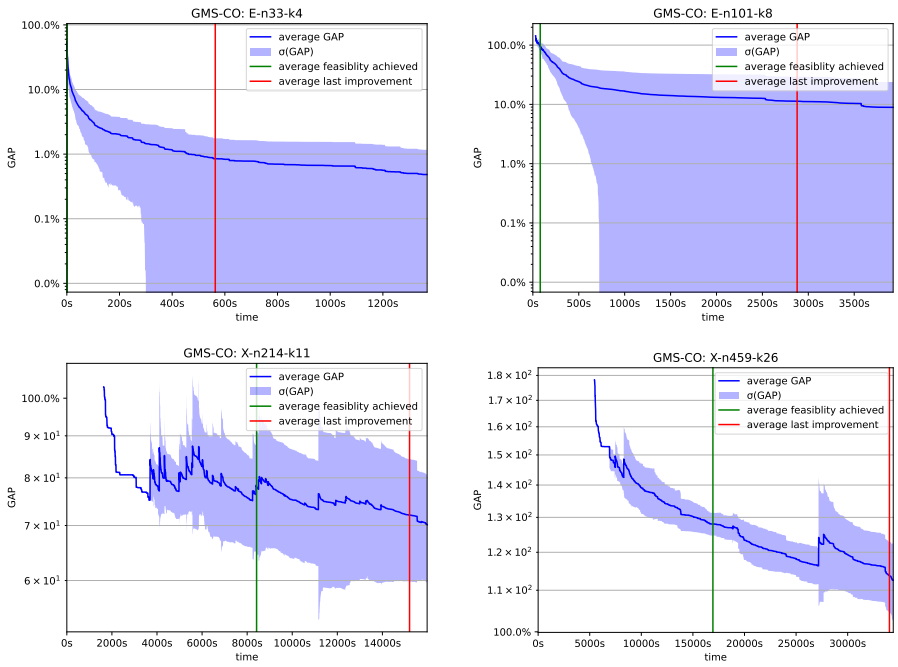


Figure 5.4: EVRP fitness minimization steps of four different instances.

5.2.2 Operator comparison

We can see in Figure 5.5 that seven operators found at least some valid solutions to instances under 750 customers and 100 vehicles, but they failed to find any solutions for instances with more vehicles. Best performing operators were APX, CX, NBX, OBX, RULX, SPX, ULX and UPMX with above 70% of feasible solutions found on average. From those APX seems as the best single operator for solving this problem with 77% of feasible solutions found on average and a 38% average $\overline{GAP}$. However, all of the operators struggled with X-n351-k40 and X-n685-k75,..., X-n916-k207 instances. This is probably caused by the larger amount of vehicles compared to other instances. Good performance on instance X-n1001-k43 is probably caused by a relatively small number of vehicles. Strangely, when using one of the seven well-performing operators independently, the solver performed the same or even better than when using a combination of the tuned operators.

feasibility

	22	23	30	33	51	76	101	143	214	351	459	573	685	749	819	916	1001	avg
AEX	100	100	100	100	62	16	0	12	0	0	0	0	0	0	0	0	0	28,82
APX	100	100	100	100	100	100	100	100	100	32	100	100	42	44	0	0	100	77,53
CX	100	100	100	100	100	100	100	100	98	12	96	100	34	16	0	0	100	73,88
ERULX	100	100	100	100	100	100	100	100	38	0	0	0	0	0	0	0	0	49,29
ERX	96	100	100	98	2	0	0	0	0	0	0	0	0	0	0	0	0	23,29
EULX	100	100	100	100	100	100	100	100	96	0	4	0	0	0	0	0	0	52,94
HGrX	2	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,24
HPrX	0	6	18	2	0	0	0	0	0	0	0	0	0	0	0	0	0	1,53
HRnX	0	48	46	22	0	0	0	0	0	0	0	0	0	0	0	0	0	6,82
MPX	100	100	100	100	100	94	6	2	0	0	0	0	0	0	0	0	0	35,41
NBX	100	100	100	100	100	100	100	100	94	2	70	100	4	0	0	0	88	68,12
OBX	100	100	100	100	100	100	100	100	100	10	94	100	32	6	0	0	100	73,06
OX	100	100	100	100	100	100	100	100	36	0	0	0	0	0	0	0	0	49,18
PBX	100	100	100	100	100	100	100	100	28	0	0	0	0	0	0	0	0	48,71
RULX	100	100	100	100	100	100	98	100	96	14	96	100	14	10	0	0	100	72,24
SPX	100	100	100	100	100	100	100	100	100	0	100	100	10	2	0	0	100	71,29
ULX	100	100	100	100	100	100	100	100	98	12	90	100	24	6	0	0	98	72,24
UPMX	100	100	100	100	100	100	100	100	86	4	88	100	10	0	0	0	94	69,53

Figure 5.5: $\overline{\mathbb{F}}$ of EVRP instances by operator

GAP

	22	23	30	33	51	76	101	143	214	351	459	573	685	749	819	916	1001	avg
AEX	8	2	2	14	149	247		816										176,86
APX	0	0	0	1	8	8	10	16	53	69	70	41	164	78			49	37,80
CX	0	0	0	2	8	8	12	17	58	76	82	44	196	85			69	43,80
ERULX	0	0	0	0	8	13	27	33	63									16,00
ERX	18	1	1	13	86													23,80
EULX	0	0	0	1	9	8	12	18	46		79							17,30
HGrX	45		111															78,00
HPrX		98	112	80														96,67
HRnX		111	112	70														97,67
MPX	1	0	0	3	20	87	150	303										70,50
NBX	0	0	0	1	9	11	13	19	63	92	115	57	216				128	51,71
OBX	0	0	0	1	9	9	12	17	56	71	77	46	184	82			70	42,27
OX	0	0	0	1	14	14	23	24	111									20,78
PBX	0	0	0	2	15	16	28	27	123									23,44
RULX	0	0	0	2	9	10	13	18	59	74	90	46	193	84			77	45,00
SPX	0	0	0	1	8	9	11	18	56		81	52	169	79			103	41,93
ULX	0	0	0	1	9	10	12	19	53	79	67	45	203	81			67	43,07
UPMX	0	0	0	1	10	10	14	19	62	84	98	51	202				107	47,00

Figure 5.6: $\overline{GAP}$ of EVRP instances by operator

Chapter 6

Conclusion

We extended the General Metaheuristic Solver (GMS) for optimization problems with the dynamic permutative representation with a new Constrained Oriented search method (GMS-CO) inspired by ASCHEA and we proposed 18 new crossover operators that work with the dynamic permutative representation.

We compared the GMS-CO with previously proposed GMS based on Local Search (GMS-LS) and with problem-specific solvers on two real-world problems, EVRP and ROADEF, both having more constraints than previously tested problems. Problem-specific solvers unsurprisingly performed better than both GMSs on both problems. Our GMS-CO performed better than GMS-LS on the ROADEF problem, which has more constraints than the EVRP. In the given 90-minute timeout, GMS-CO found 90% of all feasible solutions with average fitness within 5.5% of best-known solutions. GMS-LS found only 50% of all feasible solutions in the same time period with average fitness within 20% of best-known solutions for all but one successfully solved instance.
Results of GMS-CO and GMS-LS were more even in the EVRP problem. GMS-LS failed to find 25% of all feasible solutions and GMS-CO didn't find 40% of all feasible solutions. They both solved instances having up to 1001 customers and 50 vehicles. GMS-LS found solutions with fitness within 10% of best-known solutions for all instances smaller than 600 customers. The average fitness of GMS-CO for instances with less than 200 customers was within 10% of best-known solutions, for larger instances it was much higher, between 41% and 160% of best-known solutions. The underperformance of GMS-CO in the EVRP might be caused by the inability of ASCHEA to properly exploit the constraints with the increased vehicle count, or by the

use of unsuitable crossover operators.

A possible way to increase the performance of GMS-LS on ROADEF, and both GMSs on EVRP, would be to use a hybrid approach, combining GMS with problem-specific repair mechanisms. However, we feel this would go against the main idea of the generality of GMS. New population-based constrained handling strategies that might be better suited for EVRP, such as Stochastic Ranking or MOEAs, could also be added to the GMS utilizing proposed crossover operators.

Bibliography

[BHAB97] Atidel Ben Hadj-Alouane and James C Bean, *A genetic algorithm for the multiple-choice integer program*, Operations research **45** (1997), no. 1, 92–101.

[BHS02] S. Ben Hamida and M. Schoenauer, *Aschea: New results using adaptive segregational constraint handling*, Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600) (2002).

[BS96] T. Back and H.-P. Schwefel, *Evolutionary computation: an overview*, Proceedings of IEEE International Conference on Evolutionary Computation, 1996, pp. 20–29.

[BV23] Mirsad Buljubasic and Michel Vasquez, *mbmv7/rc roadef solver; https://github.com/mbmv7/rc*, 2023.

[CC02] Carlos A Coello Coello, *Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: A suthervey of state of the art*, Computer Methods in Applied Mechanics and Engineering **191** (2002), no. 11-12, 1245–1287.

[Chr76] Nicos Christofides, *Worst-case analysis of a new heuristic for the travelling salesman problem*, Tech. report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[D+85] Lawrence Davis et al., *Applying adaptive algorithms to epistatic domains.*, IJCAI, vol. 85, Citeseer, 1985, pp. 162–164.

[Dar59] Charles Darwin, *On the origin of species by means of natural selection*, Murray, London, 1859, or the Preservation of Favored Races in the Struggle for Life.

[DPAM02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan, *A fast and elitist multiobjective genetic algorithm: Nsga-ii*, IEEE transactions on evolutionary computation **6** (2002), no. 2, 182–197.

[FGM+16] Aurélien Froger, Michel Gendreau, Jorge E. Mendoza, Éric Pinson, and Louis-Martin Rousseau, *Maintenance scheduling in the electricity industry: A literature review*, European Journal of Operational Research **251** (2016), no. 3, 695–706.

[GGRVG14] John Grefenstette, Rajeev Gopal, Brian Rosmaita, and Dirk Van Gucht, *Genetic algorithms for the traveling salesman problem*, Proceedings of the first International Conference on Genetic Algorithms and their Applications, Psychology Press, 2014, pp. 160–168.

[Glo94] Fred Glover, *Genetic algorithms and scatter search: unsuspected potentials*, Statistics and Computing **4** (1994), 131–140.

[Glo97] ———, *Tabu search and adaptive memory programming—advances, applications and challenges*, Interfaces in computer science and operations research: Advances in metaheuristics, optimization, and stochastic modeling technologies (1997), 1–75.

[GLPZ23] Hanyu Gu, Hue Chi Lam, Thi Thanh Thu Pham, and Yakov Zinder, *Heuristics and meta-heuristic to solve the roadef/euro challenge 2020 maintenance planning problem*, Journal of Heuristics (2023), 1–37.

[gur23] *Gurobi, https://www.gurobi.com*, GUROBI (2023).

[HDB23] Vu Quoc Hien, Tran Cong Dao, and Huynh Thi Thanh Binh, *A greedy search based evolutionary algorithm for electric vehicle routing problem*, Applied Intelligence **53** (2023), no. 3, 2908–2922.

[HL+99] Georges R Harik, Fernando G Lobo, et al., *A parameter-less genetic algorithm.*, GECCO, vol. 99, 1999, pp. 258–267.

[Hra22] Jan Hrazdíra, *Metaheuristic algorithms for optimization problems sharing representation; diploma thesis; https://dspace.cvut.cz/handle/10467/102112.*

[IBM23] IBM, *Cplex,https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer*, IBM (2023).

[JH94] Jeffrey A Joines and Christopher R Houck, *On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with ga's*, Proceedings of the first IEEE conference on evolutionary computation. IEEE world congress on computational intelligence, IEEE, 1994, pp. 579–584.

[JMZ21] Ya-Hui Jia, Yi Mei, and Mengjie Zhang, *A bilevel ant colony optimization algorithm for capacitated electric vehicle routing problem*, IEEE Transactions on Cybernetics **52** (2021), no. 10, 10855–10868.

[KW22] Tomas Kroupa and Tomas Werner, *Optimalizační úlohy a jejich formulace; https://cw.fel.cvut.cz/b212/_media/courses/b0b33opt/01uvod.pdf*, 2022.

[KWV21] Viktor Kozák, David Woller, and Václav Vávra, *Vns evrp 2020 solver; https://github.com/wolledav/vns-evrp-2020*, 2021.

[L+66] Vladimir I Levenshtein et al., *Binary codes capable of correcting deletions, insertions, and reversals*, Soviet physics doklady, vol. 10, Soviet Union, 1966, pp. 707–710.

[LIDLC+16] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle, *The irace package: Iterated racing for automatic algorithm configuration*, Operations Research Perspectives **3** (2016), 43–58.

[LKM+99] Pedro Larranaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic, *Genetic algorithms for the travelling salesman problem: A review of representations and operators*, Artificial intelligence review **13** (1999), 129–170.

[LKPM97] Pedro Larranaga, Cindy MH Kuijpers, Mikel Poza, and Roberto H Murga, *Decomposing bayesian networks: triangulation of the moral graph with genetic algorithms*, Statistics and Computing **7** (1997), 19–34.

[LMS03] Helena R Lourenço, Olivier C Martin, and Thomas Stützle, *Iterated local search*, Springer, 2003.

[met] *Metacentrum; https://metavo.metacentrum.cz/.*

[MF99] Peter Merz and Bernd Freisleben, *A comparison of memetic algorithms, tabu search, and ant colonies for the quadratic assignment problem*, Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406), vol. 3, IEEE, 1999, pp. 2063–2070.

[MGSK88] Heinz Mühlenbein, Martina Gorges-Schleuter, and Ottmar Krämer, *Evolution algorithms in combinatorial optimization*, Parallel computing **7** (1988), no. 1, 65–85.

[MH97] Nenad Mladenović and Pierre Hansen, *Variable neighborhood search*, Computers & operations research **24** (1997), no. 11, 1097–1100.

[Mic96] Zbigniew Michalewicz, *Genetic algorithms + data structures = evolution programs*, Springer-Verlag, 1996.

[MK05] Alfonsas Misevičius and Bronislovas Kilda, *Comparison of crossover operators for the quadratic assignment problem*, Information Technology and Control **34** (2005), no. 2.

[MMT+20] Michalis Mavrovouniotis, Charalambos Menelaou, Stelios Timotheou, Christos Panayiotou, Georgios Ellinas, and Marios Polycarpou, *Benchmark set for the ieee wcci-2020 competition on evolutionary computation for the electric vehicle routing problem*, March 2020.

[MN21] Joaquim Martins and Andrew Ning, *Engineering design optimization*, October 2021.

[MNJ+22] Setyo Tri Windras Mara, Rachmadi Norcahyo, Panca Jodiawan, Luluk Lusiantoro, and Achmad Pratama Rifai, *A survey of adaptive large neighborhood search algorithms and applications*, Computers & Operations Research (2022), 105903.

[MQ98] Angel Kuri Morales and Carlos Villegas Quezada, *A universal eclectic genetic algorithm for constrained optimization*, Proceedings of the 6th European congress on intelligent techniques and soft computing, vol. 1, 1998, pp. 518–522.

[MTKT96] Victor V Migkikh, AA Topchy, Victor M Kureichik, and Alexander Y Tetelbaum, *Combined genetic and local search algorithm for the quadratic assignment problem*, Proceedings of IC on evolutionary computation and its applications, EvCA **96** (1996), 335–341.

[NW70] Saul B. Needleman and Christian D. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, Journal of Molecular Biology **48** (1970), no. 3, 443–453.

[opt23] *Encyclopedia britannica: 'optimization' definition; https://www.britannica.com/science/optimization*, 2023.

[PAVPT] Francisco Parreño, Ramon Alvarez-Valdes, and Consuelo Parreño-Torres, *A matheuristic algorithm for the maintenance*

planning problem at an electricity transmission system operator, Available at SSRN 4263885.

[Pet96] A. Petrowski, *A clearing procedure as a niching method for genetic algorithms*, Proceedings of IEEE International Conference on Evolutionary Computation, 1996, pp. 798–803.

[PM13] Krunoslav Puljić and Robert Manger, *Comparison of eight evolutionary crossover operators for the vehicle routing problem*, Mathematical Communications **18** (2013), no. 2, 359–375.

[RHHM02] Celso C Ribeiro, Pierre Hansen, Pierre Hansen, and Nenad Mladenović, *Developments of variable neighborhood search*, Springer, 2002.

[RK14] Julia Roberts and Mykel Kochenderfer, *Pareto optimality presentation; https://web.stanford.edu/group/sisl/k12/optimization/mo-unit5-pdfs/5.8pareto.pdf*, 2014.

[Roa20] Roadef, *Roadef problem website; https://www.roadef.org/challenge/2020/en/index.php*, 2020.

[Roa23] _____, *Societe francaise recherche operationnelle aide decision; https://roadef.org/societe-francaise-recherche-operationnelle-aide-decision*, 2023.

[Ron95] Edmund MA Ronald, *When selection meets seduction*, Proceedings of the 6th International Conference on Genetic Algorithms, 1995, pp. 167–173.

[RTP20] Manuel Ruiz, Pascal Tournebise, and Patrick Panciatici, *Roadef/euro hallenge 2020: Maintenance planning problem!ů problem description*, March 2020.

[RW22] Jakub Rada and David Woller, *Roadef 2020 solver; https://github.com/wolledav/roadef_2020*.

[RY00] T.P. Runarsson and Xin Yao, *Stochastic ranking for constrained evolutionary optimization*, IEEE Transactions on Evolutionary Computation **4** (2000), no. 3, 284–294.

[SH00] Marc Schoenauer and Sana Ben Hamida, *An adaptive algorithm for constrained optimization problems*, Proceedings of the 6th Conference on Parallel Problems Solving from Nature (2000), 529–538.

[SPE] *Standard performance evaluation corporation; https://www.spec.org/cpu2017/*.

[SVG87] JY Suh and D Van Gucht, *Incorporating heuristic information into genetic search. genetic algorithms and their applications: Proceedings of the second international conference on genetic algorithms*, 1987.

[SWL+03] X.H. Shi, L.M. Wan, H.P. Lee, X.W. Yang, L.M. Wang, and Y.C. Liang, *An improved genetic algorithm with variable population-size and a pso-ga based hybrid evolutionary algorithm*, Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.03EX693), vol. 3, 2003, pp. 1735–1740 Vol.3.

[TS95] David M Tate and Alice E Smith, *A genetic approach to the quadratic assignment problem*, Computers & Operations Research **22** (1995), no. 1, 73–83.

[VLAvLA87] Peter JM Van Laarhoven, Emile HL Aarts, Peter JM van Laarhoven, and Emile HL Aarts, *Simulated annealing*, Springer, 1987.

[WHK22] David Woller, Jan Hrazdíra, and Miroslav Kulich, *Metaheuristic solver for problems with permutative representation*, Intelligent Computing & Optimization: Proceedings of the 5th International Conference on Intelligent Computing and Optimization 2022 (ICO2022), Springer, 2022, pp. 42–54.

[WSF89] L Darrell Whitley, Timothy Starkweather, and D'Ann Fuquay, *Scheduling problems and traveling salesmen: The genetic edge recombination operator*, ICGA, vol. 89, 1989, pp. 133–40.

Appendix A

Attachments

- allStats.zip - all of the collected results, used in creating graphs, figures and tables, allStats contains the following folders:
  - EVRP-EA: results of GMS-CO's Performance comparison experiments on EVRP
  - EVRP-LS: results of GMS-LS's Performance comparison experiments on EVRP
  - EVRP-solver: results of problem-specific solver's Performance comparison experiments on EVRP
  - EVRP-op: results of Operator comparison experiments on EVRP
  - ROADEF-EA: results of GMS-CO's Performance comparison experiment ROADEF
  - ROADEF-LS: results of GMS-LS's Performance comparison experiment ROADEF
  - ROADEF-solver: results of problem-specific solver's Performance comparison experiments on ROADEF
  - ROADEF-op: results of Operator comparison experiments on ROADEF
- permutator-light.zip - project containing GMS without some of the ROADEF instances to save memory. The whole project can be found in this git-repository https://github.com/wolledav/permutator under the 'evolutionary-algorithm' branch.
- solvers.zip - contains slightly modified EVRP and ROADEF specific solvers. Links to their respective git-repositories:
  EVRP: https://github.com/wolledav/VNS-EVRP-2020,
  ROADEF: https://github.com/wolledav/ROADEF_2020