FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Optimization of the painting placement problem using evolutionary techniques
Student:	Bc. Martin Šafránek
Supervisor:	doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

The aim of this work is to design and validate an algorithm that finds the (sub)optimal placement of images on the wall based on information about paintings of given dimensions and properties (dimensions, genre, age, drawing/painting technique, direction of lighting in the painting etc.) under certain constraints (spacing between paintings, distance from the walls, restrictions on paintings' selection: dimensions, age, genre etc.). The output of the algorithm will be a set of XY coordinates of particular paintings on the wall.

1) Research existing solutions. Take inspiration from the articles listed.
2) Based on the research, propose your own method for solving the problem of optimizing the placement of paintings on the wall based on the given inputs and constraints.
3) Implement the proposed method. Make a visualisation of the results.
4) Prepare your own datasets for the design and validation of the method.
5) Evaluate the results obtained on the datasets in terms of the quality of the solution achieved.
6) Discuss the results obtained and suggest further possible improvements.

[1] BORTFELDT, Andreas; WINTER, Tobias. A genetic algorithm for the two-dimensional knapsack problem with rectangular pieces. International Transactions in Operational Research, 2009, 16.6: 685-713.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 19 September 2022 in Prague.

[2] ANAND, K. Vijay; BABU, A. Ramesh. Heuristic and genetic approach for nesting of two-dimensional rectangular shaped parts with common cutting edge concept for laser cutting and profile blanking processes. Computers & Industrial Engineering, 2015, 80: 111-124.
[3] KANDASAMY, Vijay Anand; UDHAYAKUMAR, S. Effective location of micro joints and generation of tool path using heuristic and genetic approach for cutting sheet metal parts. International Journal of Material Forming, 2020, 13.2: 317-329.

Master’s thesis

OPTIMIZATION OF THE PAINTING PLACEMENT PROBLEM USING EVOLUTIONARY TECHNIQUES

Bc. Martin Šafránek

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: doc. RNDr Ing Marcel Jiřina, Ph.D.
May 4, 2023

Citation of this thesis: Šafránek Martin. *Optimization of the painting placement problem using evolutionary techniques.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

Acknowledgments vi

Declaration vii

Abstract viii

Acronyms ix

1 Introduction 1

2 Literature review 3
- 2.1 Facility layout problem 4
- 2.2 Shelf-space planning problem 5
- 2.3 Sheet metal cutting 6

3 Problem statement and formulation 7

4 Coding and solution construction 9
- 4.1 Genetics 9
  - 4.1.1 Schema Theorem 11
- 4.2 Coding 13
- 4.3 Solution construction 15
  - 4.3.1 Individual decoding 16
  - 4.3.2 Slicing layout construction 17
  - 4.3.3 Placement heuristic 22
- 4.4 Operators 24
  - 4.4.1 Crossover 24
  - 4.4.2 Mutation 28
- 4.5 Genetic algorithm 30
  - 4.5.1 Initial population 30
  - 4.5.2 Reproductive plan 31

5 Computational results 33
- 5.1 Scenarios 33
- 5.2 Dataset 34
  - 5.2.1 Generation parameters 34
  - 5.2.2 Instances 35
- 5.3 Hyper-parameters 36
  - 5.3.1 Max number of iter 37
  - 5.3.2 Population size 37
  - 5.3.3 Maximum wildcard count 37
  - 5.3.4 Orientation weights 38
  - 5.3.5 Population division counts 38
  - 5.3.6 Initial population division counts 39

5.3.7 Overlapping penalization constant 39
5.3.8 Outside of allocated area penalization constant 39
5.4 Results 49
5.4.1 Random scenario 50
5.4.2 Packing scenario 50
5.4.3 Clustering scenario 50
5.4.4 London Gallery Wall 51
5.5 Implementation 56

6 Discussion 58
6.1 Computational result discussion 58
6.2 Improvements, extensions, future work 59
6.2.1 Free space 59
6.2.2 Non-rectangular layouts 60
6.2.3 Non-rectangular paintings 60
6.2.4 Placing heuristic 60
6.2.5 Post-optimization 61
6.2.6 Extension to other problems 61
6.2.7 Deciding which painting to place 61
6.2.8 Multiple walls 61
6.2.9 Human operator assistance 61

7 Conclusion 62

Bibliography 63

A Appendix 66

Contents of the attached medium 69

List of Figures

1.1 Painting placement at the London National Gallery 2

3.1 Example of a painting placement solution 8

4.1 Population example 10
4.2 Example of a genetic approach 11
4.3 Example of a schema 12
4.4 Example of an individual representation 13
4.5 Transformation of an individual to the painting placement solution 15
4.6 Individual decoding example 17
4.7 Slicing tree construction 20
4.8 Slicing layout construction 21
4.9 Examples of the painting placement points created by the corner-placing heuristic 23
4.10 Painting placement example 23
4.11 Crossover example for painting sequence and slicing order random keys 26
4.12 Crossover example for orientation probabilities 27
4.13 Example of a mutation on all three parts of a chromosome 29
4.14 Initial population generation strategy and transition 31

5.1 Testing maximum number of iterations 41
5.2 Testing population scaling factor 42
5.3 Testing maximum wildcard count 43
5.4 Testing orientation weights performance 44
5.5 Testing wildcard spread 45
5.6 Testing population division counts 46
5.7 Testing initial population division counts 47
5.8 Testing overlapping penalization constant 48
5.9 Painting placement solution for the random_10 instance 52
5.10 Painting placement solution for the random_20 instance 52
5.11 Painting placement solution for the packing_10 instance 53
5.12 Painting placement solution for the packing_20 instance 53
5.13 Painting placement solution for the cluster_3_6 instance 54
5.14 Painting placement solution for the cluster_4_5 instance 54
5.15 Painting placement solution for the london_gallery_wall instance 55

6.1 Example of dummy painting injection 59
6.2 Example of working with non-rectangular layout 60

A.1 Flow visualization examples 68

List of Tables

5.1 Parameters used to generate testing scenarios 34
5.2 Painting placement instances 35
5.3 Hyperparameters of the genetic algorithm 36
5.4 Percentage of paintings placed outside allocated area 40
5.5 Hyperparameter values used to obtain results 49
5.6 Statistics of the last iteration 49

List of Algorithms

1 Placement heuristic 22
2 Genetic algorithm 30

List of code listings

1 Starting a computation server 56
2 Successful computation submission response 56
3 Unsuccessful computation submission response 56
4 Example of computation submission with instance name 57
5 Example of computation submission without instance name 67

I want to thank my supervisor doc. RNDr Ing Marcel Jiřina, Ph.D. for the opportunity to discuss my thesis with him. Also, I want to thank my family for their support during my studies and during writing this thesis.

Declaration

In Prague on May 4, 2023 . . . .

Abstract

The thesis proposes a genetic algorithm with a novel chromosome representation and novel crossover implementation for solving the NP-hard problem of placing rectangular paintings on a two-dimensional grid. A chromosome is represented as multiple stochastic vectors (vector that contains non-negative elements that add up to one). Crossover is implemented as vector addition followed by normalization back to the stochastic vector. The proposed solution is tested on a generated dataset.

Keywords genetic algorithms, random keys, chromosome structure, slicing trees, facility layout, FLP, shelf-space planning, optimization

Abstrakt

Tato práce navrhuje genetický algoritmus s novou reprezentací chromozomu a novou implementací křížení pro řešení NP těžkého problému umístění obdélníkových obrazů na dvoudimenzionální mřížku. Chromozom je reprezentován jako několik stochastických vektorů (vektor obsahující nezáporné prvky, které se sčítají na jedničku). Křížení je implementováno jako sčítání vektorů následované normalizací zpět na stochastický vektor. Navržené řešení je testováno na vygenerovaném datasetu.

Klíčová slova genetické algoritmy, náhodné klíče, struktura chromozomu, řezové stromy, uspořádání pracovišť, FLP, plán polic, optimalizace

Acronyms

2D-KP	Two-Dimensional Knapsack
BL	Bottom Left
BL-F	Bottom Left Fill
BRKGA	Biased Random Key Genetic Algorithm
FLP	Facility Layout Problem
GA	Genetic Algorithm
I/O	Input,Output
NP	Nondeterministic Polynomial
RKGA	Random Key Genetic Algorithm
UA-FLP	Unequal Area Facility Layout Problem

Chapter 1 Introduction

Placement of the images or paintings on the wall may seem trivial at first. However, it is not true. There are different arrangements and constraints to each particular placement. For example, an art gallery might want to place paintings on the wall, grouping them by their author or style. One example of such placement can be seen in figure 1.1. In addition, the room's lighting and the dimensions of the wall and paintings need to be considered. Together, these requirements pose a complex problem to solve.

Furthermore, a solution that places paintings on the wall can be used in many other fields. For example, the facility layout problem places a set of facilities on a grid while having the same constraints as the painting placement – grouping related facilities and considering their dimensions [2]. Another field is retail shelf-space planning, which tries to partition a shelf in a store into rectangles [3]. Subsequently, the partitioned shelf is filled with goods that the customers can buy. Similarly to the lighting conditions for paintings, the placement of goods depends on the particular placement on the shelf – goods close to the customer's eye level have increased visibility, leading to more sales [4].

The goals of the thesis are:

1. Define the painting placement problem, its inputs, constraints, and what a solution to the painting placement problem is.
2. Create a dataset for the painting placement problem.
3. Propose and implement a genetic approach for solving the painting placement problem.
4. Evaluate the performance of the proposed genetic approach.
5. Discuss the results and suggest further improvements, extensions, and future work.

The thesis is structured as follows. Chapter 2 describes similar problems to the painting placement problem and their solution methods. They are facility layout problem, shelf-space planning, and sheet metal cutting. Chapter 3 defines the painting placement problem, its inputs, constraints, and what a solution to the painting placement problem is. The central part of the thesis is in chapter 4. It describes the proposed genetic approach and the construction of the solution to the painting placement problem. Chapter 5 evaluates the performance of the proposed genetic approach and presents the created dataset. Also, it describes the implementation of the computation server for the painting placement problem. Chapter 6 summarizes and further discusses the computational results and suggests further improvements, extensions, and future work. Lastly, chapter 7 concludes the whole thesis. There is also an appendix A, which contains figures and listings outside the thesis's main part.

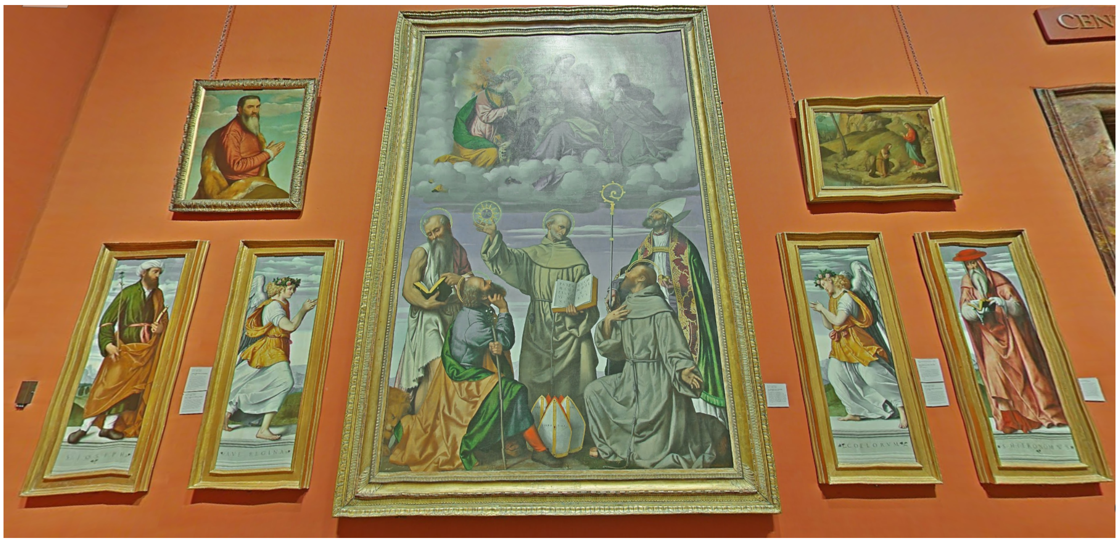


Figure 1.1 Painting placement at the London National Gallery. Source: [1]

Chapter 2
Literature review

This chapter describes methods used in different fields to solve a similar problem to the painting placement problem. It follows the methods mentioned in the previous chapter 1 with a more precise and in-depth explanation – facility layout problem (FLP) in section 2.1, shelf-space planning problem in section 2.2 and sheet metal cutting problem in section 2.3.

The similarity in all methods is that each place objects and evaluates the particular placement. The main difference is their domain, which determines the objective function, i.e., the measure by which different solutions can be compared. Also, there are differences between the given constraints, as some of them are more loose or strict, depending on the intended use of the result. A concise comparison, further explained in the following sections, follows.

Facility layout problem defines the flow between every facility pair and metric for measuring the distance between facilities. The goal is to minimize the flow sum between all facilities. Good results are mainly compactly placed facilities, where the ones with the highest flow between them are placed closer together.

Shelf-space planning problem has two main parts – dividing a shelf into rectangles and assigning a product to them. This product placement on a shelf is evaluated using a profit function. Similar to the sheet metal cutting problem, it differs from FLP in not having any mutual relationship between placed products, i.e., there is no flow. Another difference is that there are more products than available shelf space. It implies that product choice must also be part of the shelf-space planning problem. It is unique in using all shelf space in the first step. However, empty space can still exist if the product has smaller dimensions than the shelf it is placed on. Good results are mainly shelves that contain the largest amount of the most desired products at customer eye level.

Sheet metal cutting problem evaluates two aspects – compactness of the layout and distance taken by the path-cutting tool to cut all placed parts. Similar to the shelf-space planning problem, placed parts have no mutual relationship. However, unlike FLP and shelf-space planning, placed parts can have arbitrary shapes. Good results depend on the balance between the two evaluated aspects. If compactness is preferred, results will be more compactly packed, but the distance of the path-cutting tool might increase. If, on the other hand, the lower cutting distance is preferred, it results in more common edges between the placed parts or clustering.

2.1 Facility layout problem

The goal of the FLP (Facility Layout Problem) is to place facilities, which are often represented as a rectangle, in a given two-dimensional grid that is also rectangular. In addition, a flow exists between each pair of facilities, i.e., the number that defines whether it is advantageous to place facilities close together.

Authors in [2] define facility layout problem using a cost function c as

$$\arg\min_{x \in K} c(x) = \sum_{i=1}^{i=N} \sum_{j=1}^{j=N} c_{i,j}\, f_{i,j}\, d_{i,j} \,, \tag{2.1}$$

where K is the set of all possible facility placements, N is the number of facilities, $f_{i,j} \in \mathbb{R}^+$ is the flow between facility i and j, $c_{i,j} \in \mathbb{R}^+$ is price for a unit of distance between i, j and $d_{i,j}$ is their distance. Flow f is defined to be symmetric, i.e., $f_{i,j} = f_{j,i}$. The constraint to the facility placement problem is that no facilities can overlap. The FLP defined as such is NP-hard [5, 2, 6].

An important part of the input to the FLP is the facility dimensions. They are not defined as w_i, h_i pairs, where w_i is the width and h_i height of a facility i. Each facility is defined using its area a_i and maximum aspect ratio $r_i \in \mathbb{R}^+$ which must satisfy equations 2.2 and 2.3.

$$a_i = w_i h_i \tag{2.2}$$

$$\frac{\arg\max(w_i, h_i)}{\arg\min(w_i, h_i)} < r_i \tag{2.3}$$

Thus, determining the width w_i and height h_i of the facility i is part of the facility layout problem. Furthermore, no FLP dataset defines facilities in terms of their width and height. Facility records in the datasets are always in the form of a (a_i, r_i) pair [7, 8, 9].

Metric d in equation 2.1 used for measuring the distance between facilities is important in practical applications of the facility layout problem. Authors in [6] argue that using Euclidean $L2$ norm to measure facility distance produces suboptimal results as, for example, transportation of material between facilities hardly ever follows a direct route. Thus, they recommend using a contour-based metric instead. In addition, they also try to assign I/O points to the facilities, which are points from which the distance d is measured. Some authors also consider facility orientation [5, 7] to be part of the facility layout problem.

Solution methods differ in the exact definition of the FLP problem. Authors in [5] solve the UA-FLP (Unequal Area FLP), the variant of FLP where the placed facilities have unequal areas. They propose a solution using particle swarm optimization. Authors in [10] proposed a solution to UA-FLP combining harmony search and slicing tree. They represent harmony vector as coding of a slicing tree, which has two parts – the first part being a binary string determining the slicing tree node type, i.e., inner or leaf, and the second part codes the content of the slicing tree leaves. Authors in [11] use a similar genetic approach with a chromosome defined as a post-order traversal of a slicing tree. A genetic solution for the UA-FLP proposed in [2] uses a BRKGA (Biased Random Key Genetic Algorithm), where the chromosome contains facility sequence random keys, aspect ratios and position of the first facility. Next, an iterative greedy heuristic with the above chromosome as an input is used. Authors in [6] propose a unique solution to the FLP called parallel tempting based on simulated annealing.

2.2 Shelf-space planning problem

Shelf-space planning problem solves the assignment of different products to shelves to achieve maximum profit. It contains two parts – partitioning the shelf and assigning product or product variants to each partition. There can be multiple shelves that need to be considered simultaneously. [12]

Authors in [3] define the capacity and facing constraints for the shelf-space planning problem. Capacity constraints determine the maximum number of products that can be placed on each shelf. Facing constraints determine the minimum and maximum number of facings of each product, i.e., how much total area of the shelf can be taken up by the product. Also, there are availability constraints for each product, i.e., the supply limit of each product. Lastly, considering an unlimited supply of each product and multiple shelves with fixed dimensions, the authors define the shelf-space planning problem as an integer programming problem using a profit function P as

$$\max P = \sum_{i=1}^{N} \sum_{k=1}^{M} p_{ik} x_{ik} \,, \tag{2.4}$$

where N is the number of products, M is the number of shelves, p_{ik} is the profit of product i placed inside the shelf k, x_{ik} is number of products i placed inside the shelf k. According to the authors, the shelf-space planning problem mentioned above is NP-hard.

Authors in [4] add to the shelf-space problem a function that assigns different importance to parts of the shelf. They argue that the reason for using such a function is that products placed at eye level are more noticed by the customers. Thus, the proposed function has higher values for products placed at eye level and lower at the bottom of the shelf, where customer attention is the lowest. Also, authors in [13] consider the dimensions of the products, i.e., their width and height. They argue that products with a larger area are noticed more often by the customers, and thus their demand is increased.

Solution method in [13] is a genetic algorithm where each individual is represented as a container, which contains one or multiple products. Each individual is thus a power set of products that can be placed. This solution inherently decides which products to place, e.g., an individual does not contain a product, and thus the product is not placed. Decoding of an individual then takes place, which serves as an input to the BL-F (Bottom Left Fill) pack heuristic to fill up the shelves and calculate fitness.

Another solution in [4] uses a genetic algorithm and a slicing tree. They define a chromosome as a traversal of a slicing tree, which contains horizontal and vertical cuts as the internal nodes and products as leaves. Upon that traversal, they apply genetic operators – crossover as copying parts of the chromosome from parents and mutation as random swapping and inverting. When using each genetic operator, an invalid individual might be created. Thus, they fix the resultant invalid individual with a left-to-right scan.

Lastly, according to the comprehensive review [12], there exists no unified dataset for the shelf-space planning problem that can be used as a benchmark.

2.3 Sheet metal cutting

The sheet metal cutting problem evaluates two aspects – compactness of the layout, also called pattern efficiency, and distance taken by the path-cutting tool to cut all placed parts. Also, placed parts that determine the pattern efficiency can be irregular, for example, forming a polygon [14].

Authors in [15] use both of these aspects to formulate an objective function F that defines a sheet metal cutting problem as

$$\min F = w_1\, f_1 + w_2\, f_2\,, \tag{2.5}$$

where f_1 is pattern efficiency, which is calculated as $\frac{\sum_{i=1}^{N} a_i}{WH}$, with N being number of placed parts, a_i the area of the i-th placed part, W width and H height of the the smallest rectangle that can contain all placed parts, and f_2 the distance needed by the path-cutting tool to cut all the placed parts. Weights $w_1 \in \mathbb{R}^+, w_2 \in \mathbb{R}^+$ balance the bias towards pattern efficiency or path-cutting tool distance.

Some variants of the sheet metal cutting problem neglect the pattern efficiency and try only to find the shortest path of a given placed parts [16]. Then, the sheet metal cutting problem can be reformulated as the GTSP (Generalized Traveling Salesman Problem) [14]. The sheet metal cutting problem is thus considered an NP-hard problem [15].

Authors in [15] solve sheet metal cutting problem using a genetic algorithm where an individual is represented as a 3D chromosome. It contains the cluster size written as a binary number, the sequence of placed parts, and their orientation. This 3D chromosome is then input to the placing heuristic, which places the parts.

The solution proposed in [16] considers only path-cutting tool distance. However, they partition each placed part into multiple segments using micro joints, i.e., points at the edges of the part. The path-cutting tool then cuts these segments instead of the whole part at once. They propose a genetic algorithm with the 2D chromosome – containing angles that determine the placement of the micro joints and cutting sequence of these segments. The chromosome is then directly used by the path-cutting tool that starts cutting segments according to the cutting sequence defined in the second part of the chromosome.

Chapter 3

Problem statement and formulation

This chapter defines the painting placement problem, its inputs, constraints, and what a solution to the painting placement problem is. First, let us define what a painting placement instance is.

Painting placement isntance is an ordered quadruple

$$I = (P, F, L, \pi)\,, \tag{3.1}$$

where $P \subseteq (\mathbb{N}, \mathbb{N})^N$ are painting dimensions (width, height pairs), N is the number of paintings, $F \subseteq \mathbb{R}^{N,N}$ is matrix defining flow between paintings, $L \in (\mathbb{N}, \mathbb{N})$ is layout dimension (width, height pair) and $\pi : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is evaluation function.

Flow expresses the affinity of paintings to each other. Paintings that should be placed close together have flow higher compared to paintings that should not. Additionally, layout and painting dimensions are abstract, e.g., they have dimensionless units. However, they can be interpreted as any suitable measurement unit, e.g., meters.

An example of a painting placement instance is

$$I_1 = (\overbrace{\langle(5,4),(8,5)\rangle}^{paintings}, \overbrace{\begin{pmatrix} 0 & 5.8 \\ 5.8 & 0 \end{pmatrix}}^{flow}, \overbrace{(15,7)}^{layout}, \overbrace{f(x,y) = x + y}^{evaluation function})\,.$$

Instance I_1 contains two paintings, the first with dimensions 5×4, the second with 8×5. The flow between them is 5.8. The layout to which paintings are placed has a width 15 and a height 7. The evaluation function is $x + y$.

Next, each painting placement instance can have multiple solutions.

Painting placement solution is a sequence of placement points

$$S \subseteq (\mathbb{N}, \mathbb{N})^N\,. \tag{3.2}$$

For example, one solution S_1 for the instance I_1 is

$$S_1 = \langle(0,0),(6,1)\rangle\,.$$

It means that the first painting's lower left corner has coordinates $(0,0)$ and similarly $(6,1)$ for the second painting. Illustration of both I_1 and S_1 can be seen in figure 3.1.

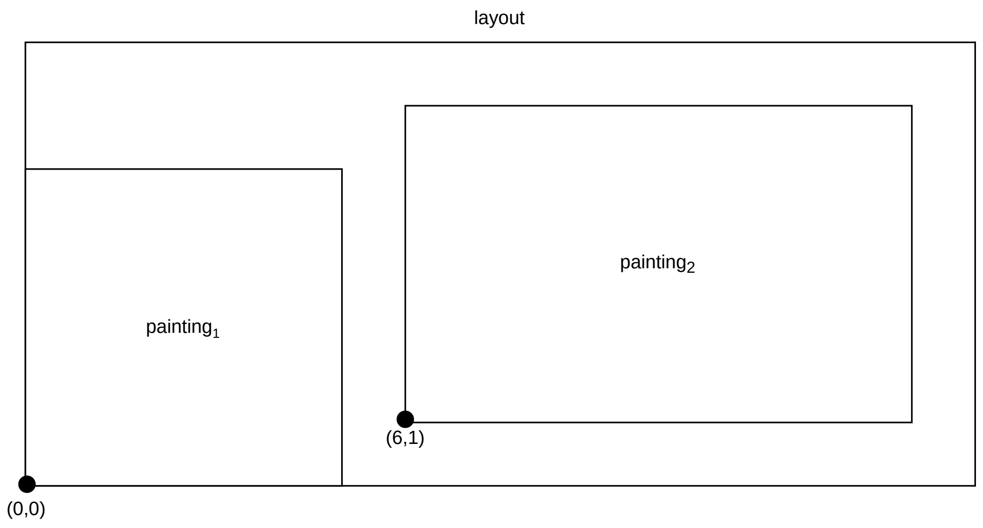


Figure 3.1 Example of a painting placement solution $S_1 = \langle(0,0),(6,1)\rangle$ for instance I_1.

Lastly, we need to evaluate the performance of the painting placement solution. Let us assume a painting placement instance I and a set of all possible painting placement solutions S to that instance. Then, the ***painting placement problem*** is to find a minimum of a cost function $c : S \to \mathbb{R}^+$, which can also be called an objective function, as

$$\operatorname*{arg\,min}_{x \in S} c(x) = \sum_{i=1}^{N} \sum_{j=i+1}^{N} f_{i,j} d(i,j) + \sum_{i=1}^{N} \pi(i) + \lambda m(x) + \gamma n(x)\,, \tag{3.3}$$

where $f_{i,j}$ is flow between painting i and j, $d(i,j)$ is their distance, $\pi(i)$ is the evaluation function applied to the bottom left corner of the painting's i placement point, m calculates number of overlapping paintings with penalization constant $\lambda \in \mathbb{R}^+$ and n calculates the number of paintings placed outside their allocated area (see 4.3.3) with penalization constant $\gamma \in \mathbb{R}^+$.

The problem defined as such is NP-hard. The reason is that by setting penalization constants λ, γ to zero and π to $f(x,y) = 0$, the objective function of a painting placement problem becomes similar to the FLP objective as defined in equation 2.1. The only difference is the price for a unit of distance in the FLP objective, but it can be added to the flow making the objective functions identical. Also, at FLP, placed facilities are defined in terms of their area and maximum aspect ratio as opposed to width and height pair in the painting placement problem. However, the facilities at the FLP problem must be assigned a dimension as the objective function application includes calculating the distance from placement points. Thus, by FLP being NP-hard [5, 2, 6], painting placement is also NP-hard.

Chapter 4

Coding and solution construction

This chapter is the central part of the thesis. It presents the novel genetic approach demonstrated on solving the painting placement problem. In section 4.1, definitions regarding genetics are laid out together with the Schema Theorem description. Section 4.2 describing individual representation and section 4.4 describing genetic operators together present the novel genetic approach. Section 4.3 describes how to decode an individual and construct a painting placement solution. Lastly, section 4.5 describes the genetic algorithm.

4.1 Genetics

This section describes important genetic terms that are used throughout the thesis. They are allele, gene, chromosome, individual, population, crossover, mutation, and reproductive plan. Also, in subsection 4.1.1, the integral part of genetics called the Schema Theorem is described.

First, it is essential to describe what the genetic approach means. The genetic approach was first introduced by Holland in 1975 to solve optimization problems where it is computationally infeasible to find an optimal solution by enumerating all possible solutions [17].

This genetic approach is inspired by nature and Darwin's Theory of Evolution – a population of individuals evolving over time. Individuals more adapted to the environment are more likely to survive and thus pass their genes to the next generation. Thus, over time, the population should converge to the state where the adaptation to the environment is the highest [18].

Holland in [17] defines several structures that reassemble this natural process. The most important ones are described in the rest of this section.

Allele represents a concrete value that a gene can have. It can be thus described as a set of alternatives to choose from.

Gene is a structure composed of alleles. It often describes one trait or characteristic.

Chromosome is a structure composed of genes. Thus it is an amalgam of characteristics described by genes.

Individual is defined by its chromosome and represents a solution to the problem or a structure from which a solution can be constructed. A numeric value called ***fitness*** can be assigned to each individual, representing how well the individual performs in an environment.

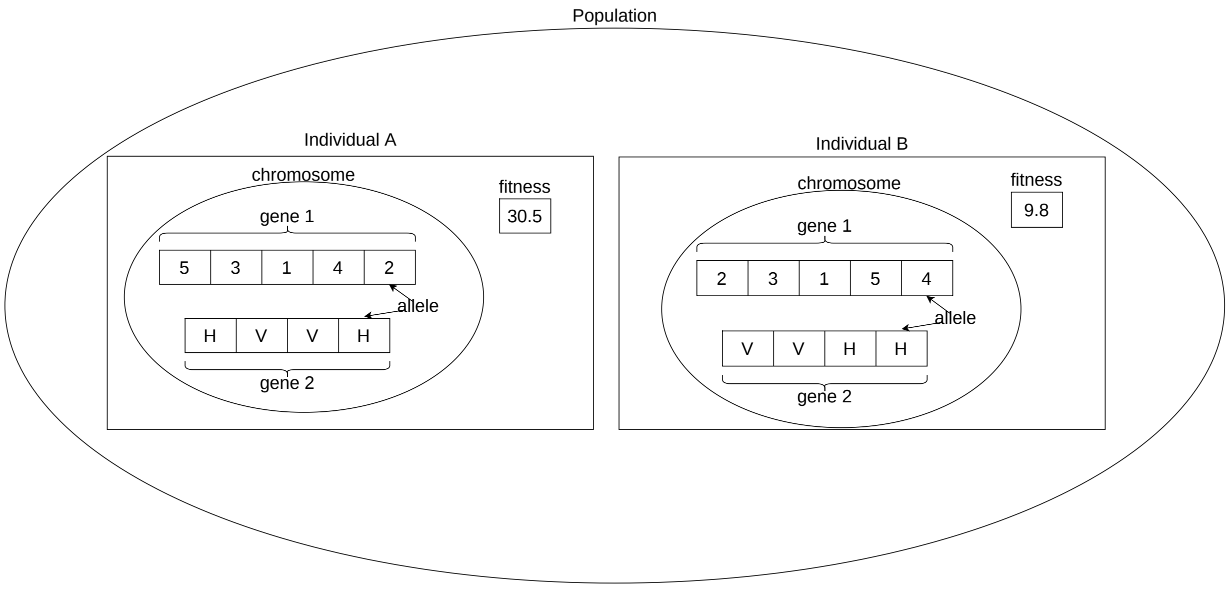


Figure 4.1 Example of a population composed of two individuals.

Population is a set of individuals. It can be interpreted as a subset of possible solutions.

One concrete example of the above-mentioned definitions is in figure 4.1. There are two individuals in the population, with their chromosomes having two genes. The first gene is a vector containing permutation with alleles of 1 to 5. The second gene is a string vector with alleles having values H or V (cut types from subsec. 4.3.1). Lastly, each individual has a fitness value. Thus, because A's fitness 30.5 is greater than B's fitness 9.8, we can say that individual A performs better than B.

For the structures defined above, multiple operations called ***genetic operators*** or simply operators are defined by Holland and used by other researchers following his work. Genetic operators aim to create new individual/s using individuals already present in a population as input. Two of them that are used in this thesis are described below.

Crossover genetic operator takes two individuals as input and, by recombination of their alleles in their genes, produces a new individual/s called ***offspring*** or ***child***.

Mutation genetic operator takes one individual as input and produces a new one which may have some of its alleles replaced by different ones at random.

Additionally, there needs to be a process that transforms a population to a new one. This process is called the reproductive plan. Also, there is a special term for the population to which the reproductive plan is applied.

Reproductive plan is a process that takes a population on input and produces a modified population on the output by using mainly genetic operators.

Initial population is the population before the first application of the reproductive plan.

Generation k is the population after applying the reproductive plan k-times to it.

Finally, a genetic algorithm uses all the processes mentioned above to find a solution to some problem.

Genetic algorithm or ***genetic approach*** applies a reproductive plan on an initial population until the stopping condition is met with the goal of finding a (sub)-optimal solution to the problem.

One example of a genetic approach is in figure 4.2. At the start, an initial population of individuals is generated. Then, the reproductive plan is applied until the stopping condition is met. Application of the reproductive plan creates the next generation by using crossover and mutation genetic operators.

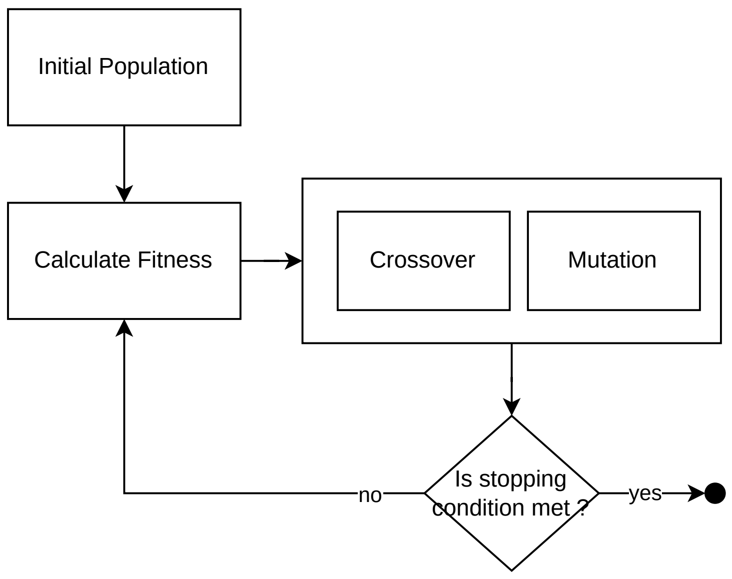


Figure 4.2 Example of a genetic approach. It uses a reproductive plan that creates the next generation by applying crossover and mutation.

4.1.1 Schema Theorem

Holland in [17] proposed the Schema Theorem arguing why the genetic approach described above works. This subsection describes the main idea behind the argument. First, an important term to describe is schema.

Schema is an extended representation of chromosome, where each gene can contain a "don't care" symbol marked as underscore _. This symbol can take up any value that an allele can in the given context. We can then say that a chromosome belongs to a schema and that a schema contains a chromosome.

Schema can be illustrated on a chromosome with one gene represented as a vector that contains a permutation of numbers 1 to 7. Then, example of a schema is $H_1 = \langle 5, \_, \_, 2, \_, 3, \_ \rangle$. It contains 24 chromosomes, with one example being $\langle 5, 4, 1, 2, 6, 3, 7 \rangle$. On the other hand, schema $H_2 = \langle 1, 2, 3, 4, 5, 6, \_ \rangle$ contains only one chromosome.

There are two other properties that a schema has. They are length and order and are defined as follows.

Length of a schema is the distance from the first "non-don't care" symbol to the last.

Order of a schema is the number of "non-don't care" symbols contained in the schema.

For example, H_1 has length 6 and order 3. It is graphically illustrated in figure 4.3. On the other hand, schema H_2 has the same length, 6, but higher order, which is also 6.

With schema being defined, we can interpret any population of individuals as a pool of schemata. It can then be reformulated that a genetic approach, which has a reproductive plan and genetic operators, (a) creates new schemata by recombination of the one already present in the population, (b) creates schemata that are absent in the population, and (c) keeps a history of the best schemata found. The Schema Theorem can then be written as

$$\mathrm{E}[M(H,t+1)] \geq M(H,t)\frac{\mu(H)}{\overline{\mu}}\left[1 - p_c\frac{\delta(H)}{l-1} - \sigma(H)p_m\right], \tag{4.1}$$

where $M(H,t)$ is expected number of individuals whose chromosome belongs to schema H in population t, $\mu(H)$ is average fitness of individuals whose chromosome belongs to H, $\overline{\mu}$ is average population fitness, $\delta(H)$ is length of schema H with it's maximum length l, $\sigma(H)$ is order of H, p_c is crossover probability, and $p_m \ll 1$ is mutation probability.

Inequality 4.1 says, that the success of a schema H, considering only crossover and low probability mutation are purely determined by its better-than-average performance, length, and order. It can be thus said that the genetic approach favors short schemata with low order that have better-than-average performance.

The reasoning behind the argument is that schemata with high order are more likely to be damaged by mutation, i.e., an allele of a schema is replaced by a different one. Also, longer schemata are more likely to be split using a crossover, whereas Holland considers a one-point-crossover that produces an offspring's chromosome by copying of the first parent's chromosome up to the crossover point, followed by the second parent chromosome after the crossover point.

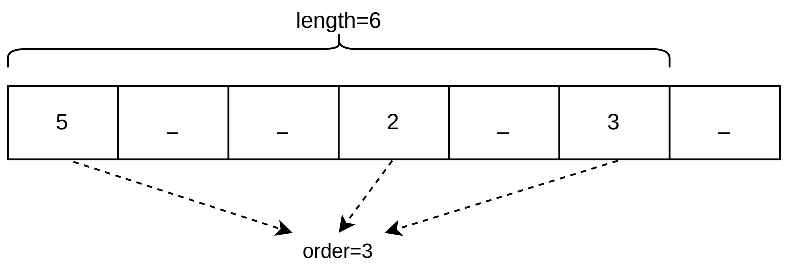


Figure 4.3 Example of a schema, where "don't care" symbol marked as underscore _.

4.2 Coding

The central part of the novel genetic approach proposed in this thesis is how an individual is represented. It is crucial for constructing the genetic operators, e.g., crossover and mutation, and decoding an individual from its representation to the solution.

An individual is represented as a 3D chromosome—which means having three genes—that is composed of painting sequence random key, slicing order random key, and orientation probabilities. An example of a chromosome is in figure 4.4.

Let us use the notation for painting sequence random key as PS_{rk}, slicing order random key as SO_{rk}, orientation probabilities as OR_{prob}, and number of paintings as N, which is also called an ***instance size***. Lower index rk means random key, and lower index $prob$ means probabilities. First two are vectors, where $PS_{rk} \in \mathbb{R}^N$ and $SO_{rk} \in \mathbb{R}^{N-1}$. Orientation probabilities is a matrix where $OR_{prob} \in \mathbb{R}^{N-1,3}$. Constraints 4.2 apply to each of these parts with a ***stochastic vector*** defined as a vector that contains non-negative elements that add up to one.

1. PS_{rk} is a stochastic vector.
2. SO_{rk} is a stochastic vector. (4.2)
3. Each row in OR_{prob} is a stochastic vector.

The representation mentioned above is based on a genetic solution to the FLP from [6, 19], where the authors represent an individual as a 3D chromosome with concrete identifiers for facilities, slicing order, and orientations. The novel approach to coding proposed in this thesis is (1) the use of stochastic vectors instead of concrete identifiers, (2) interpreting the stochastic vectors as random keys [20], (3) novel mutation and crossover operator, and (4) decoding an individual from the stochastic vector representation. Thus, the search in the proposed genetic approach takes place in a different space, which is a space of stochastic vectors.

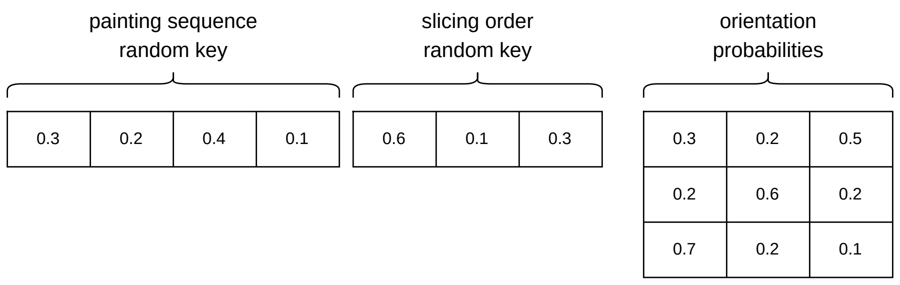


Figure 4.4 Example of an individual representation – two vectors and one matrix. Each vector and matrix row form a stochastic vector (vector that contains non-negative elements that add up to one).

There are multiple ideas behind representing an individual as a set of stochastic vectors that stem from extending RKGA [20], in which chromosome is represented as a vector with elements from interval $\langle 0, 1\rangle$. First of them is the ability to perform mutation at an arbitrary element of these vectors using a simple replacement, i.e., substituting an element for a random one from interval $\langle 0, 1\rangle$, followed by normalization back to the stochastic vector. For example, using representation described in [6, 19], there must be a different mutation method for each gene of a chromosome. By using the representation proposed in this thesis, there has to be only one mutation operator that can be used universally for all genes of the chromosome.

Additionally, when using representations similar to [6, 19], after applying the genetic operators, usually crossover and mutation, an invalid individual might be created. That is an individual that does not represent any solution. The presence of invalid individuals might lead to performance loss in FLP [21]. Moreover, unique solutions for dealing with invalid individuals must be introduced. For example, left-to-right scan used by [4, 16] or leaving invalid individuals inside the population but penalizing them [4]. The coding proposed in this thesis produces only valid individuals.

Finally, the reasoning behind using a stochastic vector instead of a vector, where each element is from interval $\langle 0, 1\rangle$ as in RKGA [20], is the novel implementation of crossover used in this thesis. It is described in subsection 4.4.1.

4.3 Solution construction

This section describes the process of how to transform or decode an individual. This transformation aims to create a solution to the painting placement problem, which is a sequence of placement points for the paintings. There are multiple steps to this process.

1. Individual decoding (4.3.1).
2. Slicing tree construction (4.3.2).
3. Slicing layout construction (4.3.2).
4. Using placement heuristic to create a painting placement solution (4.3.3).

Steps in the transformation of an individual to the painting placement solution are in figure 4.5. All of these steps are explained in the following text.

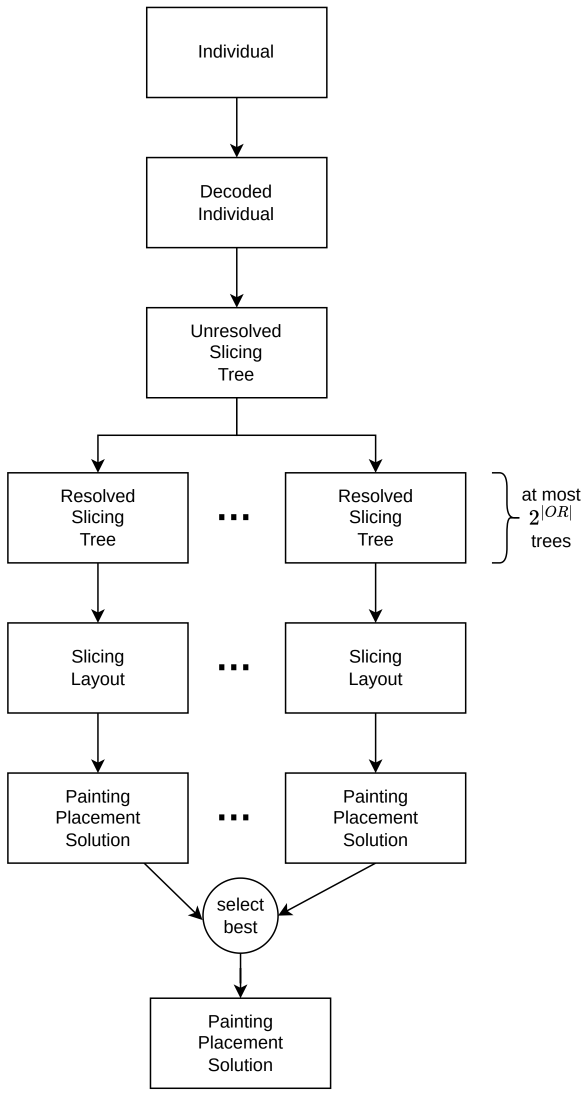


Figure 4.5 Steps in the transformation of an individual to the painting placement solution.

4.3.1 Individual decoding

First, we must decode an individual to the representation from which a slicing layout can be constructed. A decoded individual is composed of painting sequence, slicing order, and orientations. An example of individual decoding is in figure 4.6.

Let us use the notation for painting sequence as PS, slicing order as SO, and orientations as OR. PS contains painting identifiers, SO contains information used to construct slicing layout, and OR contains type of the cuts.

Cut type in OR can take up three values.

- H for horizontal.
- V for vertical.
- $*$ for wildcard, that can take up any value H or V.

The introduction of the wildcard cut type $*$ is a novel idea proposed in this thesis. In literature, only H and V cut types are used [6, 10, 21, 4].

Decoding random keys

Decoding PS_{rk} to PS and SO_{rk} to SO is the same as the RKGA in [20]. The graphical illustration is in figure 4.6 marked as *random key decoder*. Decoding random keys in PS_{rk} and SO_{rk} can be explained in the following steps on a sequence of four numbers $S = \langle 0.3, 0.2, 0.4, 0.1 \rangle$.

1. Create S' by adding a lower index to each element from S, which marks its ordinal position starting from one. $S' = \langle 0.3_1, 0.2_2, 0.4_3, 0.1_4 \rangle$.
2. Sort S' in descending order. $S' = \langle 0.1_4, 0.2_2, 0.3_1, 0.4_3 \rangle$.
3. Take lower indexes of S'. It is the result – $\langle 4, 2, 1, 3 \rangle$.

Orientation probabilities decoding

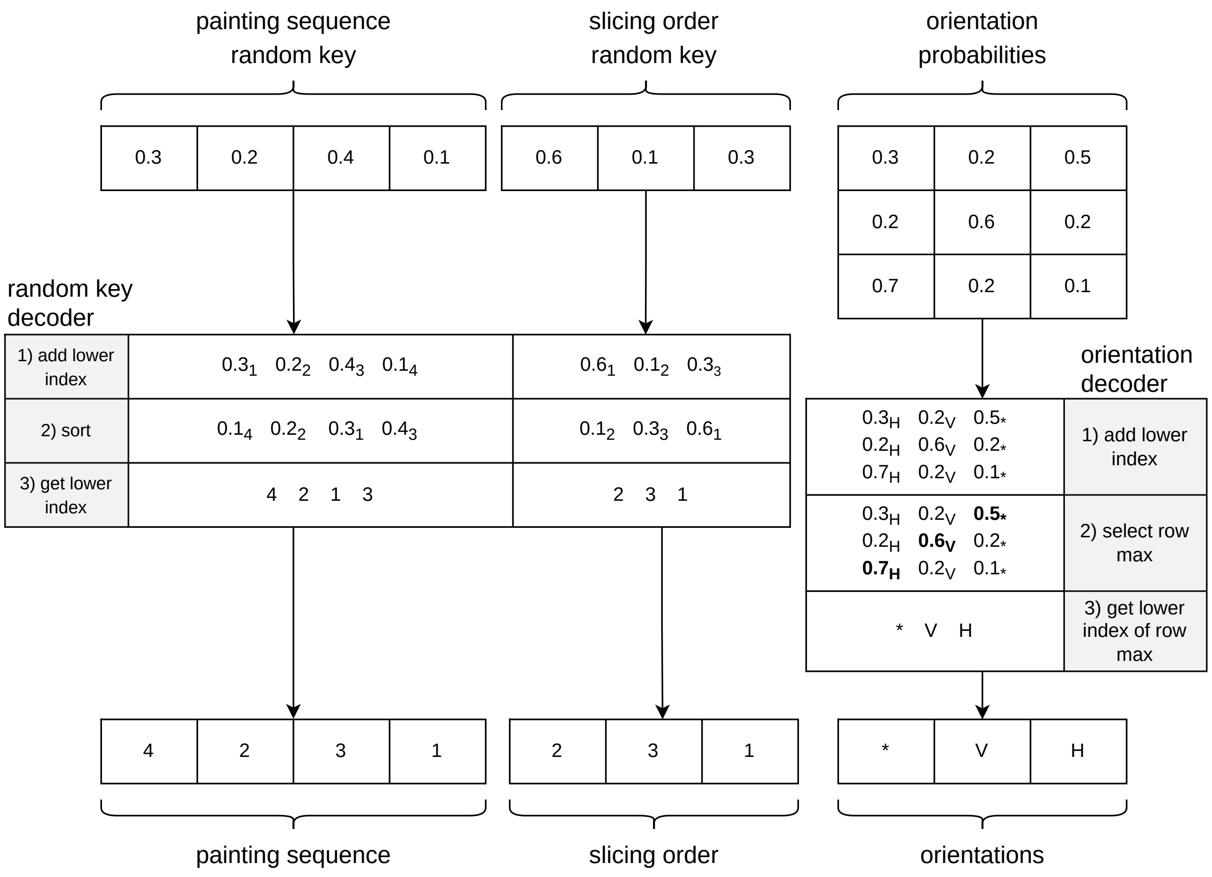
Last part of the individual, matrix $OR_{prob} \in \mathbb{R}^{N-1,3}$, decodes to OR, which is a sequence of cut types. The graphical illustration is in figure 4.6 marked as *orientation decoder*.

Decoding OR_{prob} to OR translates each row to one cut type. Thus, decoding orientation probabilities can be explained for one row, say $R = \langle 0.7, 0.2, 0.1 \rangle$ in the following steps.

1. Create R' by adding lower index H to the first, V to the second, and $*$ to the last R's elements. $R' = \langle 0.7_H, 0.2_V, 0.1_* \rangle$
2. Select element from R' with the maximum value. $\max R' = 0.7_H$.
3. Take lower index of $\max R'$. It is the result – H.

There is one exception to the steps described above. It is the limit on the maximum number of $*$ cut types produced by OR_{prob} decoding. Let us call this limit k. If the limit is not applicable, i.e., $k \geq N - 1$, there is no change to decoding steps 1–3 described above. However, only the first k wildcard cut types $*$ with the highest value are considered if applicable. It is achieved by setting the value to 0 (only for the duration of the decoding) to the bottom $N - 1 - k$ wildcard cut types $*$ with the lowest values. Then the same 1–3 decoding steps are applied as described above.

One example where the limit on wildcard cut types $*$ applies is for the $k = 1$ and OR_{prob} that has two rows, $R_1 = \langle 0.2, 0.3, 0.5 \rangle$ and $R_2 = \langle 0.1, 0.2, 0.7 \rangle$. Without exception, the result is $*, *$. However, considering the exception on the maximum limit $k = 1$, the result is $V, *$. Reason is that in R_2, wildcard $*$ has value 0.7, which is higher than value of $*$ in R_1, which is 0.5.



■ **Figure 4.6** Individual decoding example. Both the painting sequence random key and slicing order random key are decoded using the same procedure. The decoded individual is used to construct an unresolved slicing tree.

4.3.2 Slicing layout construction

In the previous subsection, decoding an individual is described. The decoded individual consists of three parts – painting sequence, slicing order, and orientations. From this representation, a slicing layout can be constructed.

Slicing layout is the recursive partitioning of space to rectangles using horizontal and vertical cuts.

Construction of the slicing layout from painting sequence, slicing order, and orientations has three steps, which are as follows.

1. Construct an unresolved slicing tree from a decoded individual.
2. Resolve an unresolved slicing tree.
3. Create a slicing layout using a resolved slicing tree.

Slicing tree construction

First, let us describe what a slicing tree is. The slicing tree was first introduced in 1982 by Otten [22] to solve automatic floorplan design. In the most general sense, it is a tree that codes the recursive division of space into rectangles using horizontal and vertical cuts. A slicing tree can thus be used to construct a slicing layout. According to [23], a slicing tree is a complete representation of a slicing layout. That means every possible slicing layout has at least one slicing tree representing it. This thesis defines and uses the slicing tree in two variants.

Resolved slicing tree is a binary tree with internal nodes having values from $\{H, V\}$ and leafs having values from the painting sequence.

Unresolved slicing tree is an extension of a resolved slicing tree where internal nodes have values from $\{H, V, *\}$.

Cut types H for horizontal and V for vertical are common for both types of the slicing tree. An unresolved slicing tree can also contain the wildcard cut type $*$.

Next, we can use a decoded individual to construct an unresolved slicing tree. This construction is graphically illustrated in the left part of a figure 4.7. During this process, the painting sequence results in leaf nodes, slicing order determines the shape of a tree, and orientations are the values assigned to internal nodes. Thus, each decoded individual represents one unresolved slicing tree.

Finally, an unresolved slicing tree is resolved. Resolving is graphically illustrated in the right part of a figure 4.7, where the unresolved tree contains one wildcard symbol $*$ as a root. By resolving this tree, $*$ is first replaced by H and then by V. In this case, resolving the unresolved slicing tree produces two resolved slicing trees, which differ in root node value. In the general case, an unresolved slicing tree can at most resolve to 2^p resolved slicing trees, where p is the number of internal nodes, i.e., the nodes that can contain wildcard $*$. Reformulation for a decoded individual is that decoded individual can, at most, represent $2^{|OR|}$ resolved slicing trees, where OR are orientations from section 4.3.1.

Slicing layout

Next, we can construct a slicing layout using a resolved slicing tree. Input to the construction has three parts that are as follows.

1. Layout to partition.
2. Areas of paintings to place.
3. Resolved slicing tree

The layout is the wall on which the paintings are placed. Areas of the paintings are retrieved using the resolved slicing tree, which contains painting identifiers as leaf nodes.

Construction can be described using figure 4.8. On the left is an input to the construction – three paintings $1, 2, 3$ with areas a_1, a_2, a_3 and resolved slicing tree. Then, we recursively traverse the resolved slicing tree. Depending on the node value, there are three possible actions.

- H – cut layout horizontally.
- V – cut layout vertically.
- Otherwise, assign node value to the layout.

After performing the cut, the process mentioned above is recursively repeated for the left and right child. If the cut is horizontal, the left child is given the upper part of the cut as its layout, and the right child is given the lower part. If the cut is vertical, the left child is given the left part of the cut as its layout, and the right child is given the right part. It can be seen in the middle part of the figure, where the cut is vertical. The left child is an orphan, i.e., it has no children. Thus, the left part of the cut is assigned value 1. The right child is not an orphan, meaning the process is applied recursively to the right part of the cut and the right child. It is depicted on the right part of the figure.

The last part of creating a slicing layout is the position of the cut. As mentioned above, the resolved slicing tree has horizontal and vertical cut types. Each cut type's position is determined proportionally to the area of rectangles assigned to the cut result. Again, it can be described using an example in figure 4.8. The first cut is vertical, where the left part of the cut is assigned rectangles 1 and the right part is assigned rectangles $2, 3$. The vertical cut thus splits the layout into two parts – the left part having 1/3 of the total layout area and the right part having the rest.

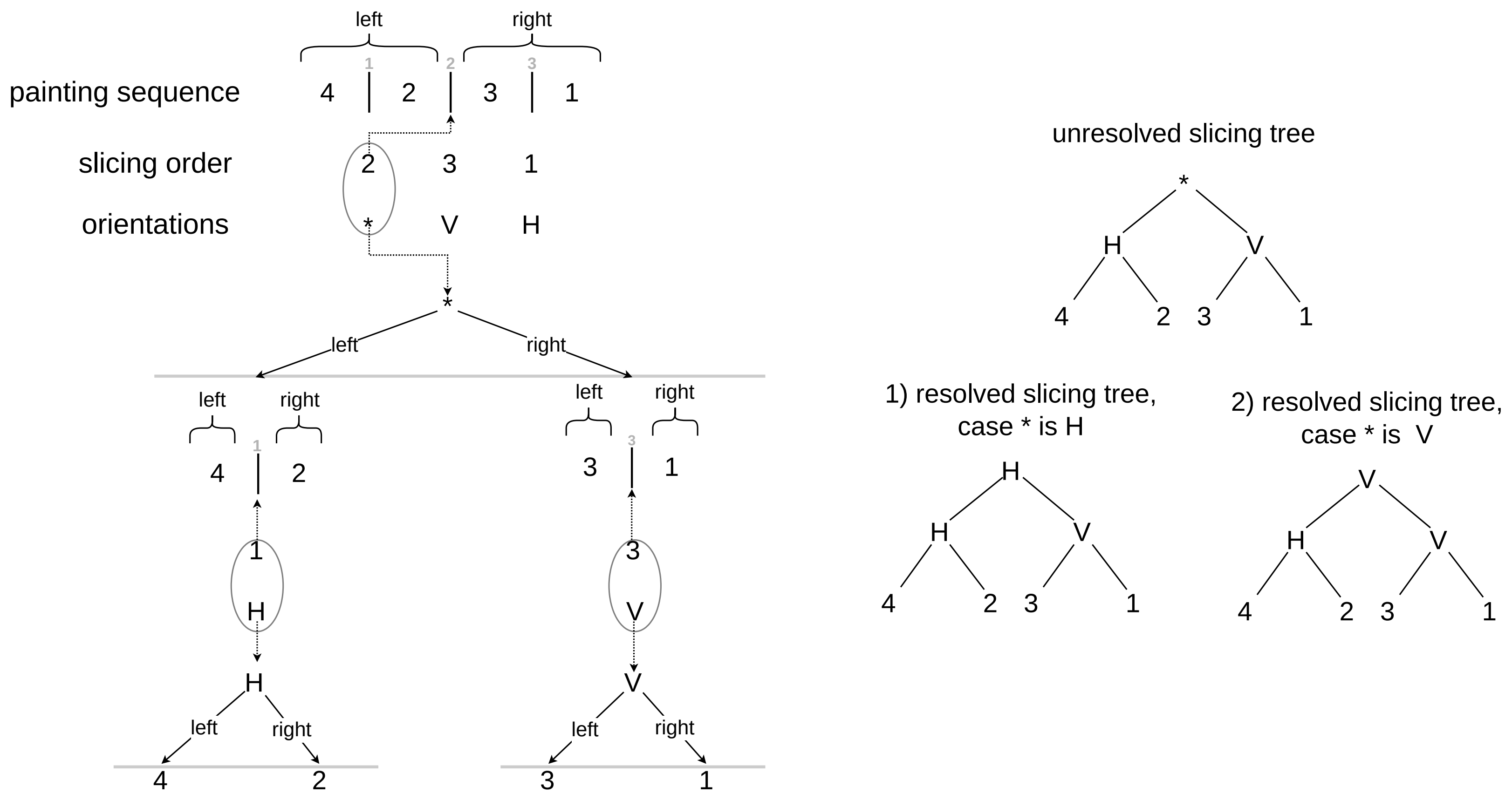


Figure 4.7 On the left is an example of unresolved slicing tree construction from a decoded individual. On the right is an example of resolving an unresolved slicing tree.

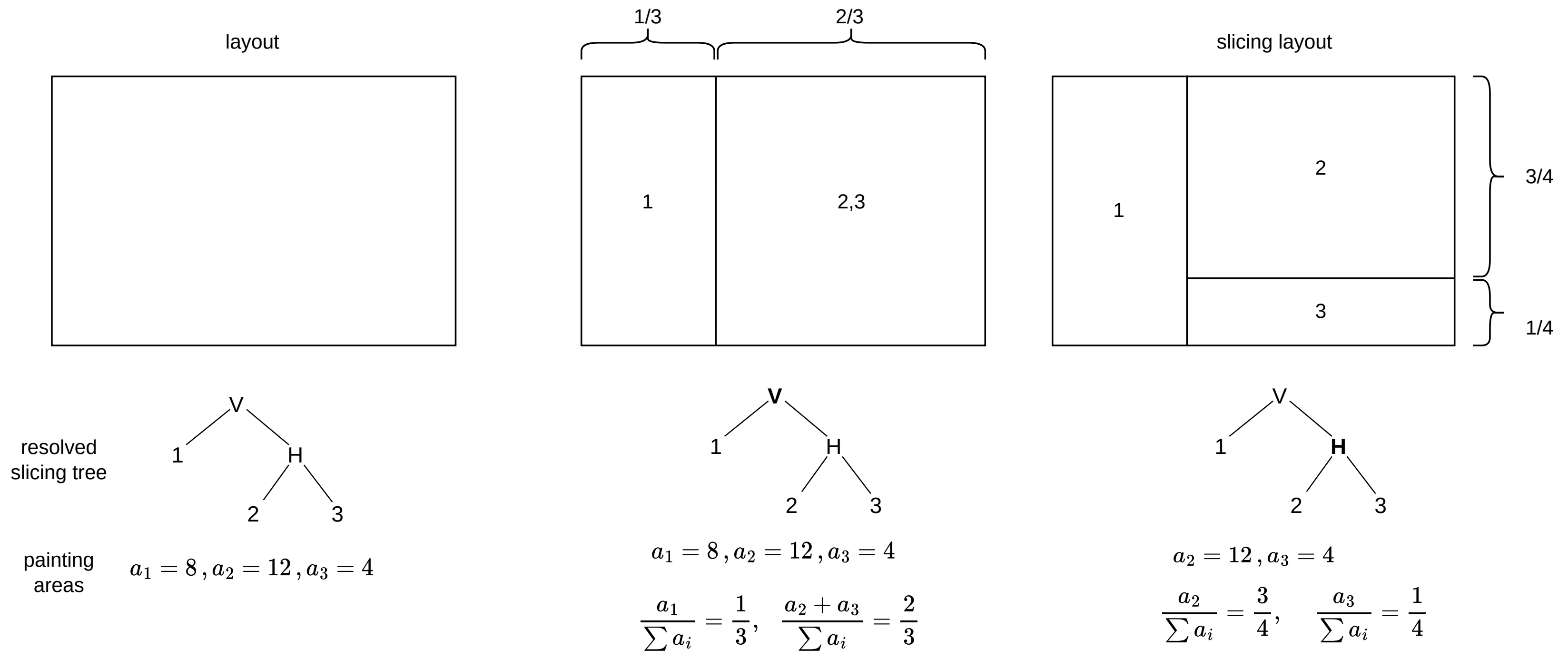

layout
1/3
2/3
1
2,3
slicing layout
2
1
3
3/4
1/4
resolved slicing tree
V
1
H
2
3
painting areas
$a_1 = 8\,, a_2 = 12\,, a_3 = 4$
$\frac{a_1}{\sum a_i} = \frac{1}{3}, \quad \frac{a_2 + a_3}{\sum a_i} = \frac{2}{3}$
$a_2 = 12\,, a_3 = 4$
$\frac{a_2}{\sum a_i} = \frac{3}{4}, \quad \frac{a_3}{\sum a_i} = \frac{1}{4}$

Figure 4.8 Example of a slicing layout construction from a resolved slicing tree, painting areas, and layout. There are three paintings, $1, 2, 3$ together with their areas a_1, a_2, a_3. The position of a cut is determined proportionally to the area of the paintings.

4.3.3 Placement heuristic

The placement heuristic is the last part of transforming an individual into a painting placement solution. Input to the heuristic is the slicing layout together with paintings, and output is the painting placement solution. The pseudocode of the proposed placing heuristic is in algorithm 1.

Algorithm 1: Placement heuristic

```
Data: slicing layout, paintings
Result: painting placement solution
1  placedPaintings ← EMPTY LIST
2  for painting in paintings do
3      best ← NIL
4      for point in possiblePlacementPoints(painting, placedPaintings, slicing layout) do
5          candidate ← place(painting, point)
6          if best == NIL
7          or objective(placedPaintings + candidate, slicing layout)
8          < objective(placedPaintings + best, slicing layout) then
9              best ← candidate
10         end
11     end
12     placedPaintings += best
13 end
14 return placedPaintings
```

The proposed heuristic is greedy and iterative. Iterative means that it creates the painting placement solution gradually, as it tries to place one painting after another using the slicing layout. It is greedy because, at each iterative step, it places a painting in a way that minimizes the objective value.

As mentioned earlier, the slicing layout recursively divides space into rectangles. Additionally, each rectangle in the slicing layout has been assigned a painting identifier. The function in algorithm 1 `possiblePlacementPoints` uses this assigned space as an allocated area.

Allocated area for a painting is a rectangle from a slicing layout to which its identifier is assigned.

It means that every slicing layout has one allocated area for each painting. Also, the allocated area does not necessarily need larger dimensions than the painting. E.g., the width of a painting might be greater than the width of its allocated area. An example of a painting and its allocated area is in figure 4.9.

An important part of the algorithm 1 is function `possiblePlacementPoints` on line 4. This function returns points that the heuristic tries for placing a painting. Implementation in this thesis is called the corner-placing heuristic.

Corner-placing heuristic is a placing heuristic that considers four painting placement points for each painting – the allocated area's bottom-left, bottom-right, top-left, and top-right corners.

Examples of the painting placement points created by the corner-placing heuristic are in figure 4.9. On the left, we can see that the allocated area's dimensions are sufficient to try all four points. In the middle, there are only two points where the placing of a painting does not result in it being outside the allocated area. On the right, all points result in being outside the allocated area. Additionally, figure 4.10 shows iteration of the corner-placing heuristic considering one placement point.

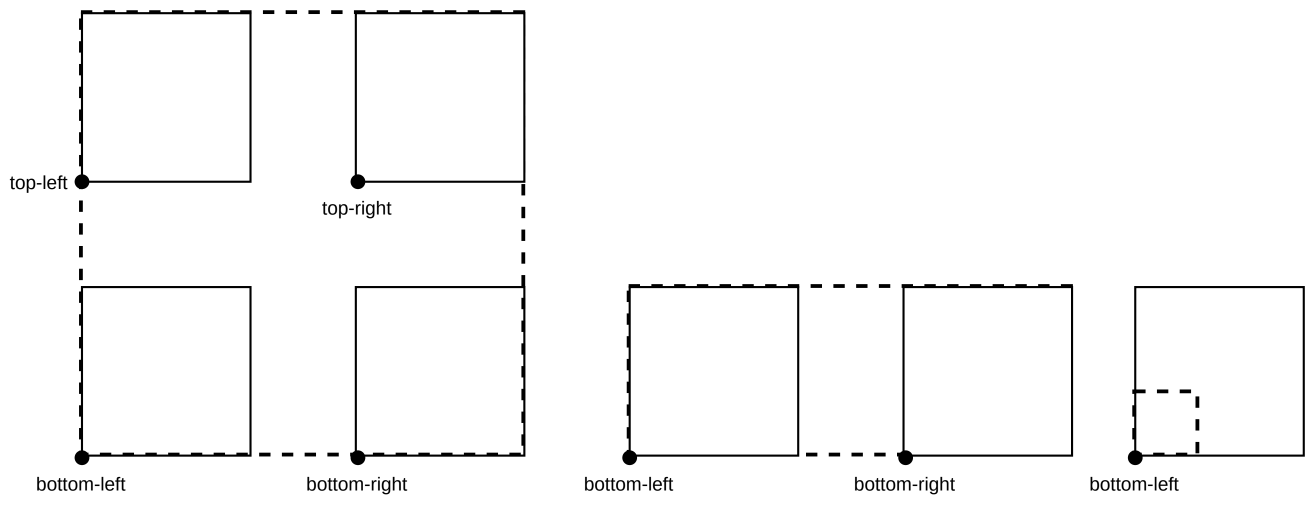


Figure 4.9 Examples of the painting placement points created by the corner-placing heuristic for three different allocated areas that are plotted using a dashed line.

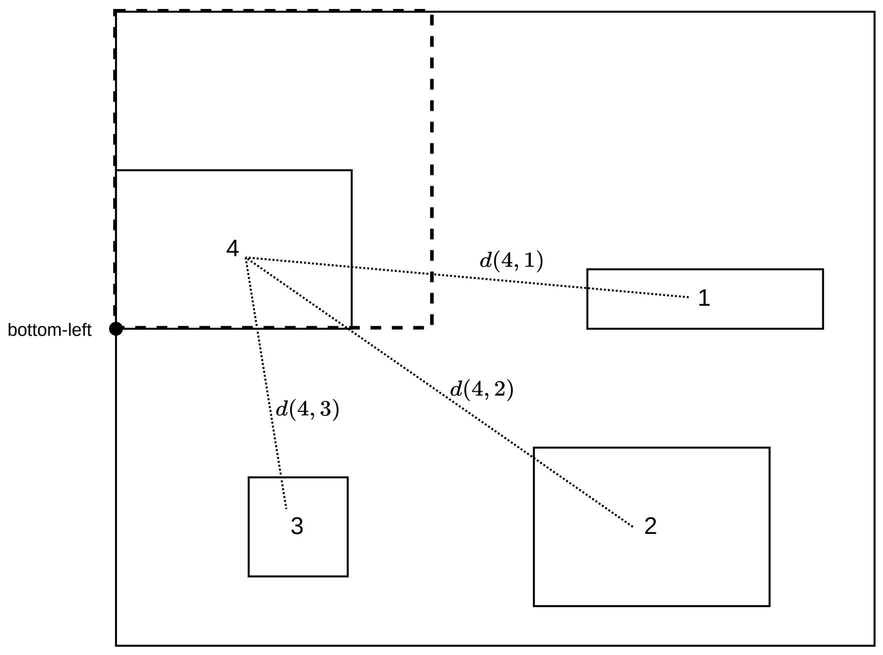


Figure 4.10 Example of a painting 4 placed inside its allocated area using the corner-placing heuristic. The allocated area is plotted using a dashed line. Also, the distances between the placed painting and all other paintings are displayed using a dotted line.

4.4 Operators

As described in section 4.1, genetic operators are used to create new individuals. This section presents the novel crossover and mutation genetic operators that are used for the individual representation from section 4.2.

4.4.1 Crossover

Novel crossover approach proposed in this thesis creates a new individual by weighted vector addition of the parent's stochastic vectors followed by a normalization back to the stochastic vector. Using notation from section 4.2, that is PS_{rk} for painting sequence random key vector, SO_{rk} for slicing order random key vector and OR_{prob} for orientation probabilities matrix, we can define crossover for two parents A, B and offspring C as

$$\|w_A A_{PS_{rk}} + w_B B_{PS_{rk}}\| = C_{PS_{rk}} \ , \tag{4.3}$$

$$\|w_A A_{SO_{rk}} + w_B B_{SO_{rk}}\| = C_{SO_{rk}} \ , \tag{4.4}$$

$$\|P^T(w_A A_{OR_{prob}i:} + w_B B_{OR_{prob}i:})\| = C_{OR_{prob}i:} \ , \tag{4.5}$$

where $\| \cdot \|$ is normalization to the stochastic vector[1], $+$ is vector addition, $w_A, w_B \in \mathbb{R}$ are weights, $P \in \mathbb{R}^{N-1}$ is orientation penalization vector with N being instance size, notation $X_{i:}$ means i-th row of a matrix X and multiplying a vector by a scalar multiplies each element of the vector by that scalar.

Example of crossover for painting sequence random key and slicing order random key (eq. 4.3 and 4.4), is in figure 4.11. An example of crossover for orientation probabilities (eq. 4.5) is in figure 4.12.

The crossover implementation described above has multiple parts – vector addition, weights, orientation penalization, and normalization. Following are arguments for incorporating each of those parts into a crossover.

Vector addition and normalization

Adding and then normalizing vectors to stochastic vectors differs from other crossover implementations. For example, in one-point-crossover [17] and uniform crossover [24], parts of the parent chromosomes are copied directly without any modification to form an offspring.

By using a stochastic vector for painting sequence random keys, slicing order random keys and rows of an orientation probability matrix, we can interpret each of them as a probability mass function. Next, by implementing the crossover as vector addition followed by normalization, we can say the crossover approximates a probability mass function of some distribution from two samples, i.e., two parents. Throughout the multiple generations, more samples are added to this approximation. Thus, each part of the chromosome tries to approximate the probability mass function that produces (sub)-optimal painting placement solution.

Going back to the schema theorem described subsection 4.1.1, we can say that the crossover proposed in this thesis does not prefer schemata with any particular order and that length of a schema does not matter.

Preference for schemata with a short length in one-point-crossover [17] stems from the fact that a chromosome is split at a particular position. This idea of splitting a chromosome is absent in the proposed approach. For example, consider slicing order random key

[1]Normalization $\|\langle x_1, x_2, \ldots, x_n\rangle\| = \langle y_1, y_2, \ldots, y_n\rangle$, where $y_i = \dfrac{x_i}{\sum\limits_{k=1}^{n} x_k}$.

vector $A_{rk} = \langle 0, 0.2, 0.7, 0.1\rangle$. Using the decoding procedure described in 4.3.1, A_{rk} decodes to $A = \langle 1, 4, 2, 3\rangle$. Then, A belongs to the schema $S_1 = \langle 1, \_, \_, \_\rangle$ and also to schema $S_2 = \langle 1, \_, \_, 3\rangle$. S_1 has order 1 and S_2 has order 4. Let us apply crossover to A_{rk} and $B_{rk} = \langle b_1, b_2, b_3, b_4\rangle$. Result is $\|\langle b_1, 0.2 + b_2, 0.7 + b_3, 0.1 + b_4\rangle\|$. We cannot make any assumptions about whether the result belongs to S_1 or S_2, as it purely depends on B_{rk}. Additionally, we cannot predict the probability of whether S_1 or S_2 survives, i.e., they will still be present in the crossover result. However, when using a one-point-crossover, it would depend on the position of the split.

Lastly, the proposed crossover does not prefer any particular order of schemata for the same reasons mentioned above. It is a common feature with one-point crossover, which only prefers shorter schemata.

Weights

Adding weights w_A and w_B determines the preference for transferring information from one parent to another. The weights are calculated using a cost function c from eq. 3.3 as

$$w_A = \frac{c(B)}{c(A) + c(B)}, w_B = 1 - w_A \,. \tag{4.6}$$

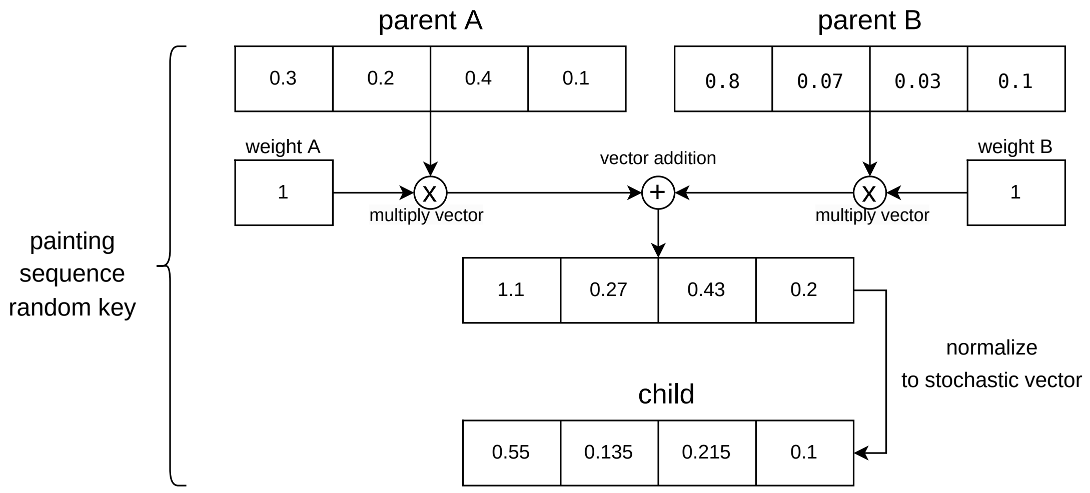
Thus, the parent with better performance in the population, i.e., having lower cost function, has more influence on what genes are being transferred to the offspring.

Adding w_A and w_B lowers the chance of creating offspring that do not share the advantageous schemata. For example, considering only one stochastic vector of length three as a chromosome, it might be advantageous to have a high value for the first value in the chromosome, e.g., $A = \langle 0.7, 0.1, 0.2\rangle$. On the other hand, a poorly performing individual might be $B = \langle 0.1, 0.3, 0.6\rangle$. Without weights, $\|A + B\| = \langle 0.4, 0.2, 0.4\rangle$. Adding weights according to the eq. 4.6 penalizes the transfer of poorly performing schemata.

Orientation penalization

Another part of the crossover used in the orientation probabilities matrix is the penalization vector P. As mentioned in section 4.3, each individual decodes to one unresolved slicing tree whose internal nodes contain a type of the cut – H for horizontal, V for vertical, and $*$ for a wildcard, that can take up any value H or V. Vector P controls the preference for each type of cut. For example, setting $P = \langle 1, 1, 0.5\rangle$ penalizes only the wildcard cut $*$. On the other hand, setting $P = \langle 1, 1, 1\rangle$ removes any penalization.

The main reason for introducing P is to limit the spread of wildcard cut $*$ in population, making its appearance only at the most advantageous parts of the genes.



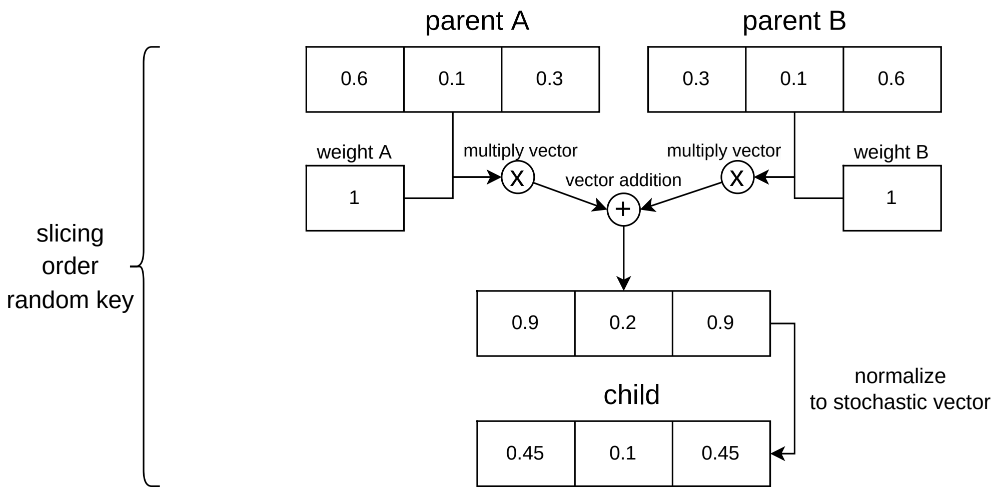
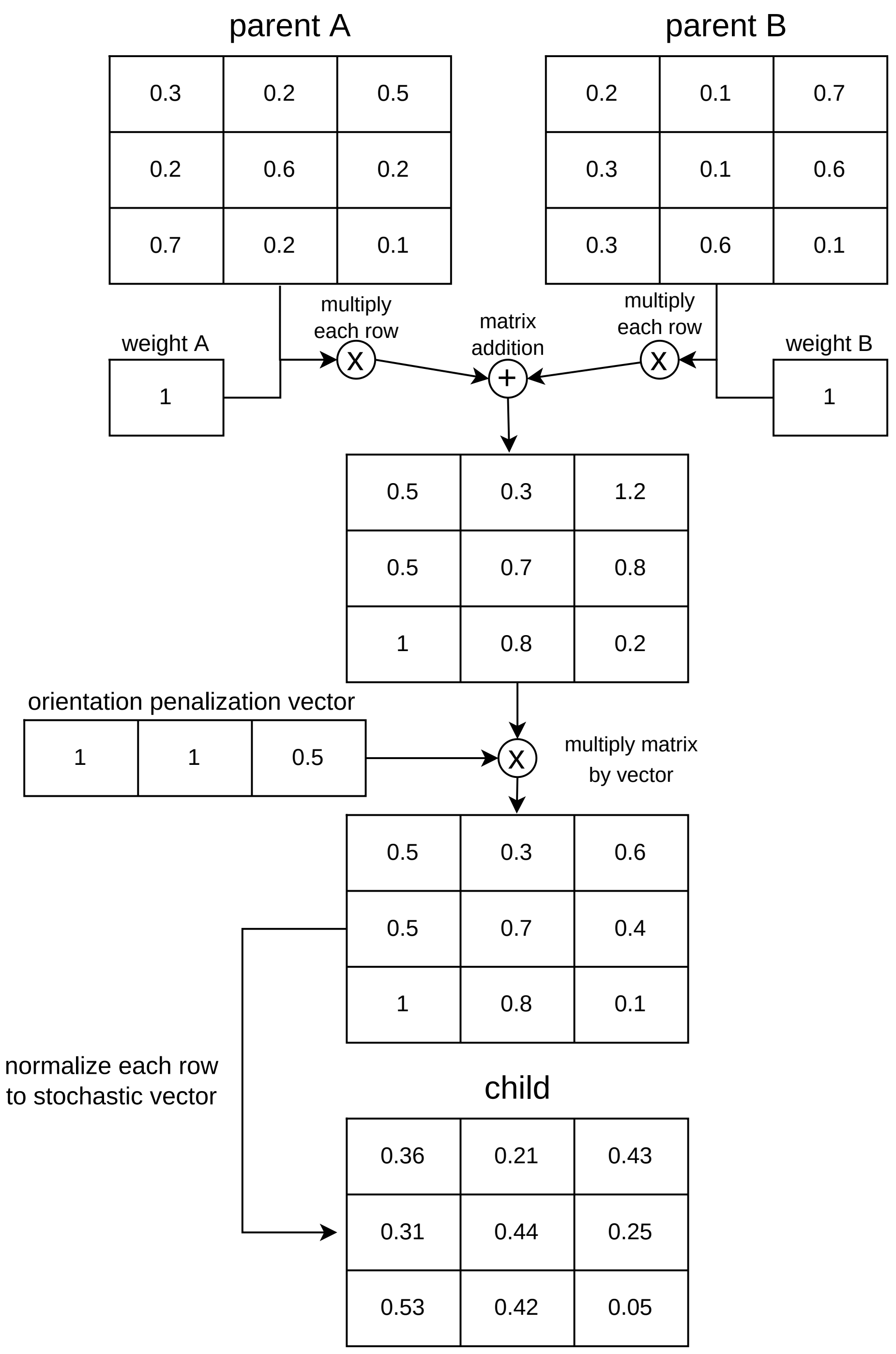


Figure 4.11 Crossover example for painting sequence and slicing order random keys. The procedure is the same for both – sum weighted parent vectors and then normalize to stochastic vector.



■ **Figure 4.12** Crossover example for orientation probabilities. The procedure is first to sum weighted parent matrices, then multiply the matrix with orientation penalization vector and normalize each row to a stochastic vector.

4.4.2 Mutation

Mutation can happen on all three genes of a chromosome – painting sequence random key PS_{rk}, slicing order random key SO_{rk} and orientation probabilities OR_{prob}. Due to the coding described in section 4.2, PS_{rk}, SO_{rk} and rows of OR_{prob} are stochastic vectors. We use this common trait and define the mutation operator for a stochastic vector as follows.

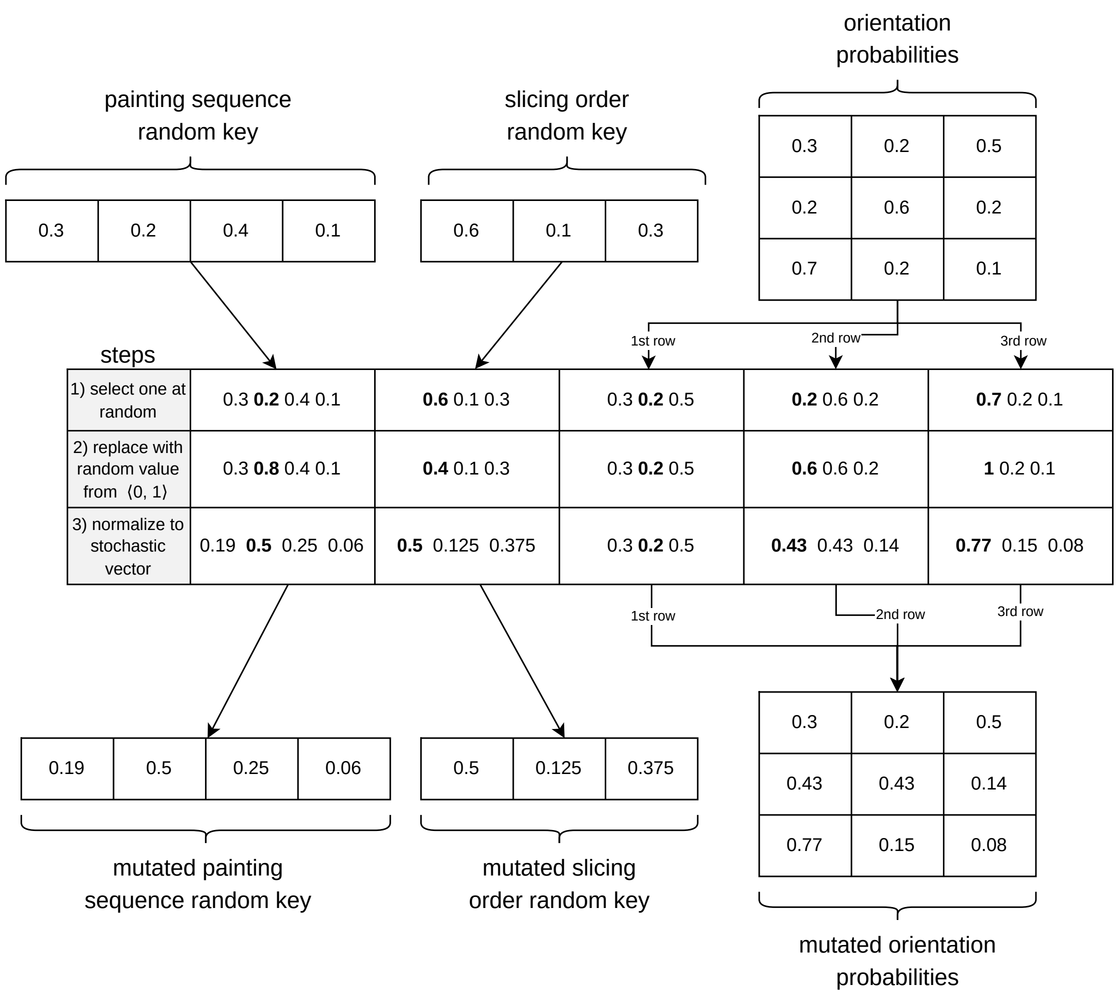
1. Replace one randomly chosen element with a uniformly generated value from $\langle 0, 1 \rangle$.
2. Normalize to stochastic vector.

With the definition of the mutation operator for a stochastic vector above, the mutation operator for an individual is defined as follows.

1. Choose one of PS_{rk}, SO_{rk}, OR_{prob} at random.
2. If PS_{rk} or SO_{rk} is chosen, apply the mutation operator for a stochastic vector.
   If OR_{prob} is chosen, select one row at random and apply a mutation operator for a stochastic vector.

It is important to mention how to interpret a mutation operator. As mentioned in section 4.3, an individual decodes to one unresolved slicing tree. Mutation modifies this tree. First, if applied to OR_{prob}, the value of an inner node of the tree might change. That means a change in a type of cut – H, V, or $*$. Second, if applied to PS_{rk}, the value of leaves in the tree might change. Lastly, if applied to SO_{rk}, the whole structure of a tree might change.

As described above, even changing one value can result in a completely different tree and after further decoding the slicing layout and painting placement solution. It is the reason for defining mutation as such – to damage the chromosome as little as possible. An example of a mutation that happens on all tree parts at once is in figure 4.13.



steps					
1) select one at random	0.3 **0.2** 0.4 0.1	**0.6** 0.1 0.3	0.3 **0.2** 0.5	**0.2** 0.6 0.2	**0.7** 0.2 0.1
2) replace with random value from ⟨0, 1⟩	0.3 **0.8** 0.4 0.1	**0.4** 0.1 0.3	0.3 **0.2** 0.5	**0.6** 0.6 0.2	**1** 0.2 0.1
3) normalize to stochastic vector	0.19 **0.5** 0.25 0.06	**0.5** 0.125 0.375	0.3 **0.2** 0.5	**0.43** 0.43 0.14	**0.77** 0.15 0.08

Figure 4.13 Example of a mutation on all three parts of a chromosome. Since all parts can be treated as a stochastic vector, the same procedure is used for all of them – replace one value randomly and then normalize it to a stochastic vector.

4.5 Genetic algorithm

This section concludes the whole chapter by presenting the genetic algorithm which is used to find an individual representing (sub)-optimal solution to the painting placement problem. Pseudocode is presented in algorithm 2, initial population generation strategy is described in subsection 4.5.1, and reproductive plan is described in subsection 4.5.2.

Genetic algorithm has two main properties that have to be well-balanced with respect to each other – intensification and diversification [25].

Intensification is the ability to identify parts of the search space with a high-quality solutions.

Diversification is the ability to prevent premature convergence to the suboptimal solutions.

We can classify genetic operators in terms of their intensification and diversification effects. The mutation operator is considered the most straightforward diversification strategy, as it creates a small change in an individual's chromosome that can lead the search out of the suboptimal solution [25]. On the other hand, crossover creates a new solution by recombination of already present individuals, which can be considered a diversification strategy [25]. However, researchers in [26] argue that crossover also has an intensification effect. They argue that if the population were primarily composed of the same individuals, the crossover would not be able to improve the solution.

Algorithm 2: Genetic algorithm

```
population ← generateInitialPopulation
for $i$ ← 1 to maxNumberOfIter do
    population ← applyReproductivePlan(population)
end
return selectBest(population)
```

4.5.1 Initial population

The initial population is the population that is used as a starting point for the genetic algorithm. It consists of two parts – RANDOM and GREEDY. Visualization can be seen on the left of the figure 4.14.

RANDOM part consists of randomly generated individuals. The process of generating is (1) fill vectors PS_{rk}, SO_{rk} and matrix OR_{prob} with random values from $\langle 0, 1\rangle$, and then (2) normalize PS_{rk}, SO_{rk} and rows of OR_{prob} to stochastic vectors to meet constraints in 4.2.

GREEDY part consists of individuals who are, at worst, as good as RANDOM. The process of generating k GREEDY individuals is (1) to create $100k$ RANDOM individuals and (2) to select k best ones in terms of their objective value.

Incorporating GREEDY individuals into the initial population may decreases the time the genetic algorithm needs to find a space with high-quality solutions. The reason for adding RANDOM individuals is that greedy solutions might increase the chance of an algorithm getting caught in local optima.

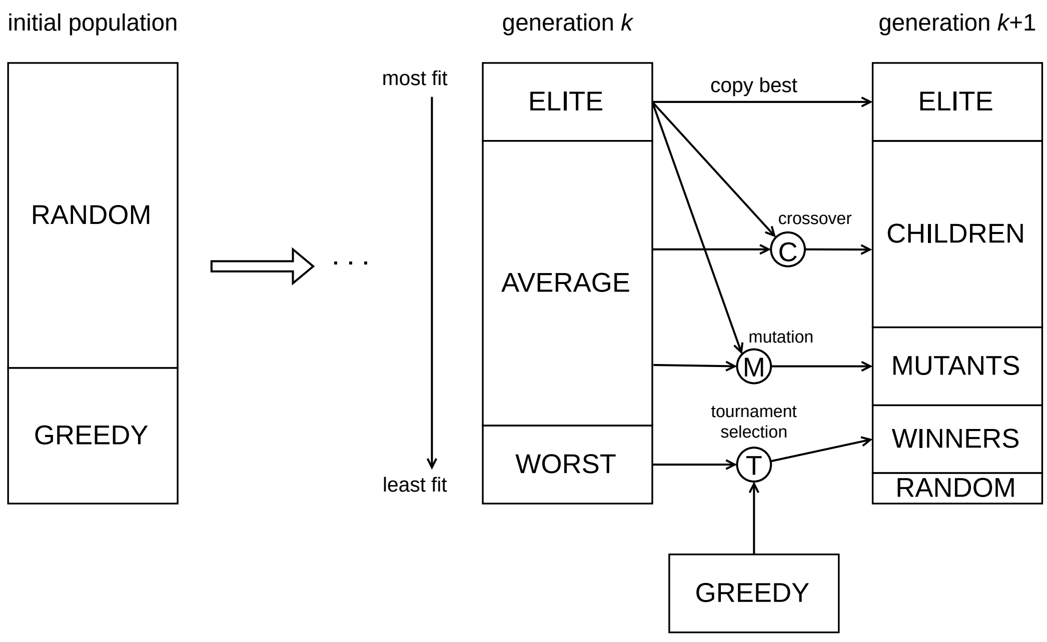


Figure 4.14 Initial population generation strategy and transition from generation k to $k + 1$ using a reproductive plan.

4.5.2 Reproductive plan

This section describes a reproductive plan used in this thesis. It is visualized on the right of the figure 4.14. The reproductive plan starts with partitioning the individuals in the current population into three groups according to their performance.

ELITE individuals are the ones that decode to the solutions with the lowest objective value.

WORST are the ones that decode to the solutions the highest objective value.

AVERAGE individuals are in between the ELITE and WORST.

Then, the elitism strategy is used. It copies all ELITE individuals to the next generation. Elitism enforces intensification, keeping the best individuals inside the population without any modification or recombination with others. It increases the representation of current (sub)-optimal solutions in the population, and thus it is more likely that operators increasing intensification will use ELITE as an input. In addition, without elitism, the best individuals might be lost after crossing over or mutation.

The next step is to use crossover and mutation genetic operators, with the crossover being the most significant in terms of the individuals it produces for the next generation.

CHILDREN are individuals that are created using a crossover, with the first parent being selected at random from ELITE and the second parent selected at random from AVERAGE.

MUTANTS are individuals that are created using a mutation, with an input selected at random from ELITE and AVERAGE.

Another step is the tournament selection between the least performant and greedily generated individuals. Reason for it is that the least performant individuals often receive high penalization values in objective 3.3. It means that they produce solutions with mostly overlapping paintings or paintings outside their allocated area.

WINNER individuals result from tournament selection between WORST and GREEDY. Selection picks the best individuals from WORST and GREEDY until all available spots for WINNER are filled.

Finally, RANDOM, a small group of randomly generated individuals, is injected into the next population. The reason is to decrease the chance of the genetic algorithm getting stuck in a local optimum by randomly adding samples from the search space.

Chapter 5
Computational results

This chapter presents the computational results of the proposed solution, generated dataset and testing scenarios, hyperparameters of the genetic algorithm, and implementation of the computation server to which a user can submit a painting placement instance and receive a solution to that instance.

Four testing scenarios used for evaluation are described in section 5.1. They are random, clustering, packing, and London National Gallery. Next, in section 5.2, the dataset created for each testing scenario is described. Then, section 5.3 describes and discusses the hyperparameters of the proposed genetic algorithm 2. Also, reasonable hyperparameter values are determined. Section 5.4 presents a painting placement solutions to the painting placement instances and their visualizations. Lastly, section 5.5 describes the implementation of the computation server.

5.1 Scenarios

Four testing scenarios evaluate different aspects of the proposed solution. They are random, clustering, and packing. Additionally, one scenario describes the painting placement at the London National Gallery in figure 1.1.

Random scenario contains randomly generated painting placement instances. It is mainly used for performance testing.

Clustering scenario tests the ability to form clusters. It is achieved by dividing the paintings into groups. Paintings belonging to the same group have increased flow between them. Paintings from the distinct group have flow between them set to 0.

Packing scenario is the same as the random scenario, with the only difference being that the layout area is equal to the area of all paintings summed together. It tests the ability to create compact solutions.

London National Gallery scenario contains one painting placement instance created from the London National Gallery in figure 1.1. It tests the ability to work with actual painting placement used at a gallery.

5.2 Dataset

This section describes the parameters for creating datasets for random, clustering, and packing testing scenarios. Generated painting placement instances are described in subsection 5.2.2.

Additionally, as mentioned in chapter 2, no datasets in the literature would satisfy the definition 3.1 of painting placement instance. Thus, all datasets are exclusively created by the author and can be used by other researchers for benchmarking their solutions.

Generation of testing instances is performed using a Python programming language in combination with Jupyter Notebook. Both the datasets and Notebooks are in the attached medium.

5.2.1 Generation parameters

Generation parameters used to create testing instances are presented in table 5.1. A description of each of them follows in the rest of this subsection.

Table 5.1 Parameters used to generate testing scenarios

	Layout area ratio	Max paint. width	Max paint. height	Max paint. ratio	Flow min	Flow max	Eval func.
random	1.2	10	10	3	0	4	$x + y$
clustering	1.2	10	10	3	-	-	0
packing	1	10	10	3	0	4	0

Left-out values marked with - are discussed later in the text.

Layout area ratio is the ratio between the area of the layout and the painting area sum. It is computed as

$$\frac{\sum_{i=1}^{N} w_i h_i}{WH},$$

where w_i is width, h_i is height of painting i, W is width, and H is height of the layout. If the layout area ratio is set to 1, it means a preference for more compact layouts. On the other hand, increasing this value implies the presence of more free space in the resulting layout.

Max painting ratio controls the maximum aspect ratio between width w and height h of each painting. It is computed as

$$\frac{\max(w, h)}{\min(w, h)}.$$

Increasing the max painting ratio implies the possibility of the generation of paintings that are very thin, i.e., $w \ll h$ or $h \ll w$. On the other hand, setting the value to 1 implies that every generated painting is square.

Evaluation function is function π from objective function 3.3. In the random scenario, the evaluation function is set to $x + y$ because of its simplicity, linearity, and interpretability. Also, it implies that placing small paintings close to the bottom left corner is advantageous as the function value is the lowest there and big paintings to the top right corner. On the other hand, for clustering and packing scenarios, the evaluation function is set to a constant value 0. The reason is that different capabilities are tested (clustering and packing). Furthermore, using a non-constant evaluation function makes it harder to interpret the results.

Rest of the parameters, ***max painting width***, ***max painting height***, ***flow min***, ***flow max*** are self-explanatory and were set as low numeric values to increase computation speed and avoid overflow.

5.2.2 Instances

Six painting placement instances are generated using table 5.1 parameters, and one is created from the London National Gallery in figure 1.1. They are described in table 5.2. For random and packing scenarios, values for max painting width/height, flow min/max, and max painting ratio are randomly generated from the parameter range described in table 5.1. The clustering parameters are also generated randomly. The only exception is the flow, which is set in a way to form clusters.

Visualization of the flow for two painting placement instances is in the appendix in figure A.1. On the left is the randomly generated flow for random_10 instance, and on the right is the flow for cluster_3_6 instance that forms clusters.

Table 5.2 Painting placement instances

Instance name	Paint. count	Layout width, height	Scenario	Description
random_10	10	24 x 19	random	
random_20	20	31 x 25	random	
packing_10	10	19 x 15	packing	
packing_20	20	33 x 26	packing	
cluster_3_6	18	30 x 25	clustering	3 clusters, 6 paintings each
cluster_4_5	20	34 x 27	clustering	4 clusters, 5 paintings each
london_gallery_wall	7	180 x 90	London National Gallery	created from figure 1.1

5.3 Hyper-parameters

Proposed genetic algorithm 2 has eight hyperparameters. They are described in table 5.3 and used in listing 4. The rest of the sections discuss these hyperparameters further and tries to find their reasonable values.

Two instances are chosen for hyperparameter testing – random_10 and random_20. The reason is that they are not biased towards any preferred solution, e.g., forming clusters. Thus, insights into the proposed solution can be gained. However, fine-tuning the hyperparameters to the specific instance or scenario is recommended but only sometimes computationally feasible.

Hyperparameter testing is performed by changing only the hyperparameter under test. Hyperparameters not under test are identical to the values in listing 4. The exceptions are penalization constants λ, γ (eq. 3.3), and `populationSize`. Penalization constants are set to the length of the layout diagonal (see 5.3.7). Population size is set to $50N$, where N is the size of the instance. The reason for choosing such parameters as base parameters for testing is preliminary results (not presented in this thesis), which proved correct in many cases.

Lastly, to achieve the statistical significance of the results presented in this section, each computation[1] is submitted five times with a different random seed. Presented values are thus an average from five samples.

Table 5.3 Hyperparameters of the genetic algorithm 2

Hyperparameter	**Description**
`maxNumberOfIter`	maximum number of iterations
`populationSize`	population size
`maximumWildCardCount`	limit on the maximum number of $*$ cut types produced by OR_{prob} decoding
`orientationWeights`	penalization vector P from eq. 4.5
`populationDivisionCounts`	reproductive plan ratios (right part of fig. 4.14)
`initialPopulationDivisionCounts`	initial population ratios (left part of fig. 4.14)
`overlappingPenalizationConstant`	overlapping paintings penalization constant λ from eq. 3.3
`outsideOfAllocatedAreaPenalizationConstant`	outside of allocated area penalization constant γ from eq. 3.3

[1]Copmutations were run on a notebook with Fedora 36, 16GB RAM, AMD Ryzen 7 PRO 4750U 8 x 1.7 - 4.1 GHz, Renoir PRO (Zen 2).

5.3.1 Max number of iter

Hyperparameter `maxNumberOfIter` determines the number of iterations in the genetic algorithm 2.

Results for two random instances are in figure 5.1. We can see the initial decrease of the average population objective for both random instances. Above iteration 300, the initial decreasing trend stops for the random_10 instance, and for the random_20 instance, the decrease becomes very slow.

The conclusion is that at least 300 iterations are needed before the average population objective stops decreasing rapidly.

5.3.2 Population size

Hyperparameter `populationSize` is calculated as κN, where κ is ***population scaling factor*** and N is instance size. It determines the population size that is linear to the instance size.

Results for two random instances are in figure 5.2. We can see that scaling factor 10 does not allow the population objective average to decrease to the levels comparable to scaling factors 50 and 100. It might imply that the scaling factor 10 cannot represent knowledge gathered over time in the genetic algorithm or that more iterations are needed.

The conclusion is that using scaling factor between 50 and 100 is sufficient, with bias towards 100 for obtaining better average objective performance. However, increasing the scaling factor leads to slower computation speed as every population contains more individuals for which reproductive plan must be computed.

5.3.3 Maximum wildcard count

Hyperparameter `maximumWildCardCount` limits the maximum number of $*$ cut types produced by OR_{prob} decoding (subsec. 4.3.1). Keeping this hyperparameter low or even setting it to zero is recommended. The reason is that if it is high, computation time increases as $*$ spreads in the population. For example, consider a decoded individual whose orientations OR are solely composed of $*$ cut types. Then, the individual decodes to $2^{|OR|}$ resolved slicing trees as seen in figure 4.5.

Results for random_10 instance are in figure 5.3. The top sub-figure shows that the average population objective does not differ significantly for any limit on the wildcard cut type. However, there is a slight advantage for the maximum wildcard count equal to one. It might be caused by using wildcard penalization 0.5 (listing 4) that does not allow the spread of the wildcard in the population.

On the bottom sub-figure, we can see the computation speed as the limit on the wildcard cut type increases. It grows linearly up to the maximum wildcard count of eight, and then the increase stops. It might be because the wildcard penalization 0.5 does not allow the wildcard cut type to spread over the maximum wildcard count of eight. Another reason might be computational anomalies caused by high-memory consumption as the maximum wildcard count increases.

The conclusion tested on random_10 instance is that (a) the maximum wildcard count for wildcard penalization 0.5 performs similarly for all values, with a slight performance gain if using a maximum wildcard count equal to one, and (b) computation time is linear with increasing maximum wildcard count and wildcard penalization 0.5.

5.3.4 Orientation weights

Hyperparameter `orientationWeights` is the orientation penalization vector P from crossover eq. 4.5. It determines the bias towards the type of cut (H, V, $*$, see 4.3.1).

Only penalization for the wildcard cut type $*$ is tested, as there is no need to penalize or have a preference for H or V cut types. Also, recall that the hyperparameter `maximumWildCardCount` is set to one during testing, as described at the beginning of the section and showed in listing 4.

Performance results for two random instances are in figure 5.4. We can see that for random_10 instance, weight does not significantly influence the average population objective, and after iteration 250, differences become negligible. However, for random_20 instance, weight one (no penalization) has a faster-decreasing trend and produces a population with a better average population objective.

The reason for better average performance at larger instance with weight one (no penalization) might be that search space increases exponentially (there exists at least 2^{N-1} different unresolved slicing trees, for instance of size N), and the introduction of wildcard cut type $*$ starts to manifest itself at larger instances.

Results for the average number of wildcard cut types $*$ at the best individual before decoding at each iteration are in figure 5.5. We can see that for the random_10 instance, weights below one have less than one wildcard. However, there are between three and four wildcards for a weight equal to one (no penalization). Similar can be seen for random_20 instance.

The reason why there are wildcards present in the figure 5.5 even for weight equal to zero (maximal penalization) is that wildcard can be introduced to a chromosome by mutation or injection of random individuals (see reproductive plan 4.5.2). Described penalization only applies to the crossover.

The conclusion from figure 5.4 is that (a) smaller instances, such as random_10, do not benefit from the introduction of wildcard cut type $*$, and (b) bigger instances, such as random_20, benefit from no wildcard penalization by having a faster-decreasing trend and producing a better average population.

The conclusion from figure 5.5 is that if we want to be certain that wildcard cut type $*$ is contained in the best individual at each iteration, wildcard orientation weight must be set close to one.

5.3.5 Population division counts

Hyperparameter `populationDivisionCounts` configures ratios in the reproductive plan (subsec. 4.5.2). It influences how the next generation is created in the genetic algorithm 2 by setting (a) how many elite individuals are copied, (b) how many children are created using a crossover operator, (c) how many mutants are created using a mutation operator, (d) how many tournament winners are included, and (e) how many random individuals are injected.

Performance results for two random instances are in figure 5.6. We can see that results do not differ for the smaller or larger instance. The best reproductive plan strategy is achieved without changing population division hyperparameters from listing 4. That means elitism and randomly injected individuals are present. Additionally, the elitism strategy is the root cause of good performance, as removing randomly generated individuals does not significantly improve performance.

The reason why the use of elitism is important to obtain good results might be the implementation of the crossover operator (subsec. 4.4.1). Crossover adds weights w_A and w_B to each parent based on their objective value (lower objective value achieves bigger weight). It means that if $w_A \gg w_B$, the offspring is a sample from the search space close to the parent A or nearly identical to A. On the other hand, if the weights are similar and each parent represents a different (sub)-optimal solution, the transfer of information does not happen, and the offspring is no better than a randomly generated individual. Elitism avoids crossover by directly copying the best individuals without any modification. It leads to keeping track of multiple (sub)-optimal solutions simultaneously. In addition, each time a crossover is applied, one parent is selected from the elite pool, which supports intensification.

On the other hand, removing elitism produces the worst results. The reason might be the inability to keep track of competing (sub)-optimal and the fast spread of the first ***macho individual***, an individual that performs significantly better than everyone else. Without elitism, the most significant way to keep track of found (sub)-optimal solutions is through crossover. Crossover chooses parents randomly from the average pool, which does not give any constraint on the weights w_A and w_B. Additionally, as mentioned in the previous paragraph, similar weights for two different parents produce no better offspring than randomly generated individuals. It means that most offspring do not perform well. On the other hand, as soon as the macho individual appears as the parent in the crossover, the offspring is effectively a copy of a macho individual. Then, if the macho individual is randomly chosen more than once as a parent in a crossover, it rapidly spreads and takes over the whole population. The chance of macho-individual taking over the population with elitism is decreased as it deliberately keeps track of multiple competing performant individuals, making it harder for the macho individual to spread.

The conclusion is that using elitism is essential for obtaining good painting placement solutions as it can keep track of multiple (sub)-optimal solutions simultaneously.

5.3.6 Initial population division counts

Hyperparameter `initialPopulationDivisionCounts` configures generation ratios of the initial population (left part of fig. 4.14). It consists of randomly generated and greedily generated individuals.

Performance results for two random instances are in figure 5.7. The different ratios' results do not greatly differ.

The conclusion is that hyperparameter `initialPopulationDivisionCounts` does not significantly affect the obtained results.

5.3.7 Overlapping penalization constant

Hyperparameter `overlappingPenalizationConstant` is the penalization constant λ from eq. 3.3 used to penalize individuals representing solutions with overlapping paintings. It is calculated as ρD, where ρ is a diagonal multiple and D is the length of a diagonal in a layout. Diagonal length D for a layout with width W and height H is $\sqrt{W^2 + H^2}$.

Results of the average overlapping paintings count for the best individual at each iteration are in figure 5.8. We can see that the two lowest diagonal multiples with values 0.5 and 1.0 fail to remove most overlapping paintings from the best individuals. On the other hand, values 2 and higher can remove overlapping paintings at the smaller instance. However, in the larger instance, there are still several overlappings present.

The conclusion is that hyperparameter `overlappingPenalizationConstant` should be set at least to two times the diagonal length of the layout to start penalizing individuals that represent solutions with overlapping paintings.

5.3.8 Outside of allocated area penalization constant

Hyperparameter `outsideOfAllocatedAreaPenalizationConstant` is the penalization constant γ from eq. 3.3 used to penalize individuals representing solutions with paintings that are placed outside of their allocated area (see fig. 4.9 and subsec. 4.3.3). It is calculated identically as the overlapping penalization constant (subsec. 5.3.7). That is, as ρD, where ρ is a diagonal multiple, and D is the length of a diagonal in a layout.

Results of the percentage of paintings placed outside the allocated area for two random instances are in table 5.4 (it is presented using a table because the values do not significantly change throughout the iterations). We can see that the outside of allocated area penalization constant cannot force the creation of the allocated space that would fit the vast majority of the paintings. Interestingly, there are a few percentage points drops in favor of the larger instance. The reason might be that it has more degrees of freedom, i.e., more possibilities to create a cut. It can thus create more fitting slicing layouts.

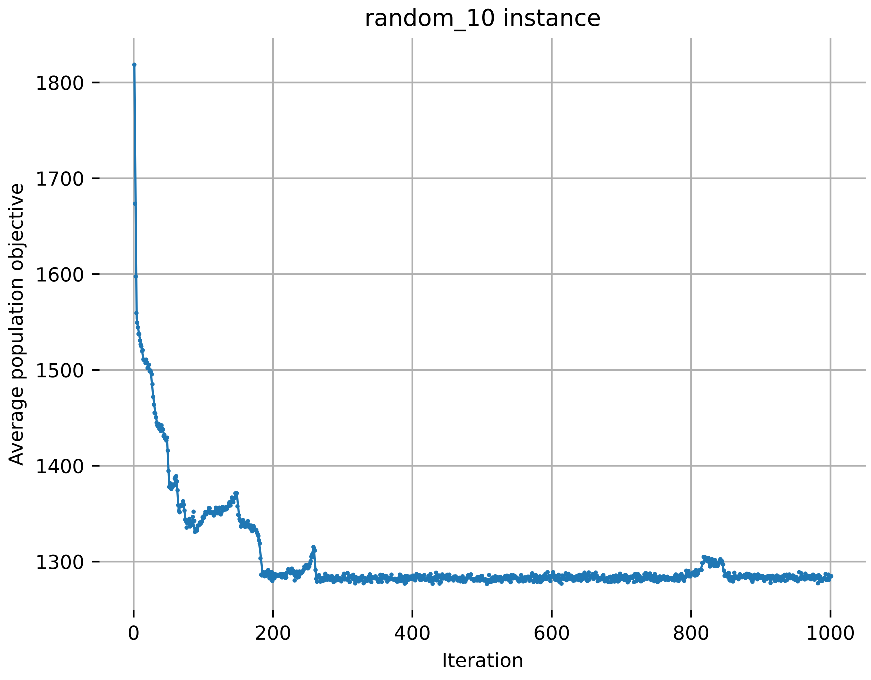
However, the failure of the outside of allocated area penalization constant to force the creation of sufficient allocated space in most cases is not that important. The reason is that the placing heuristic (subsec. 4.3.3) tries to place the painting at several placement points in the allocated area (see fig. 4.9). It can thus balance the few paintings that have been allocated sufficient area to avoid and account for other parts of the objective function, e.g., overlapping paintings.

The conclusion is that the hyperparameter `outsideOfAllocatedAreaPenalizationConstant` is the least important and might be left out by setting it to zero to save computation time.

Table 5.4 Percentage of paintings placed outside allocated area

	Instance	
Diagonal multiple	random_10	random_20
0	99.6	93.3
0.5	99.6	94.1
1	98	92.5
2	97.4	95.9
3	98.9	91.7
4	99.8	94.9
10	99.7	96.1
50	98.6	97.8

Percentage is averaged over all iterations for the best individual at each iteration.



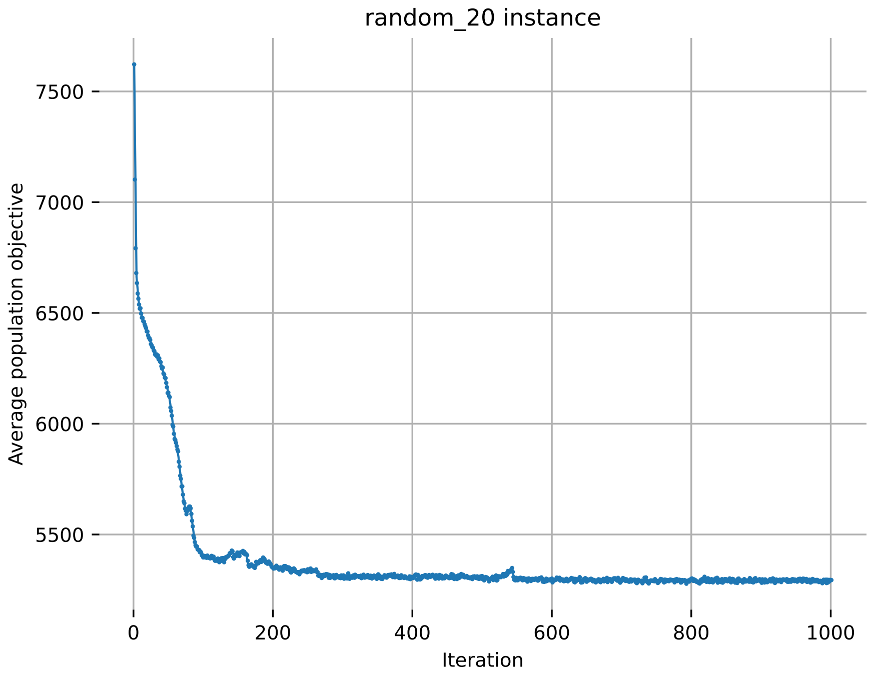
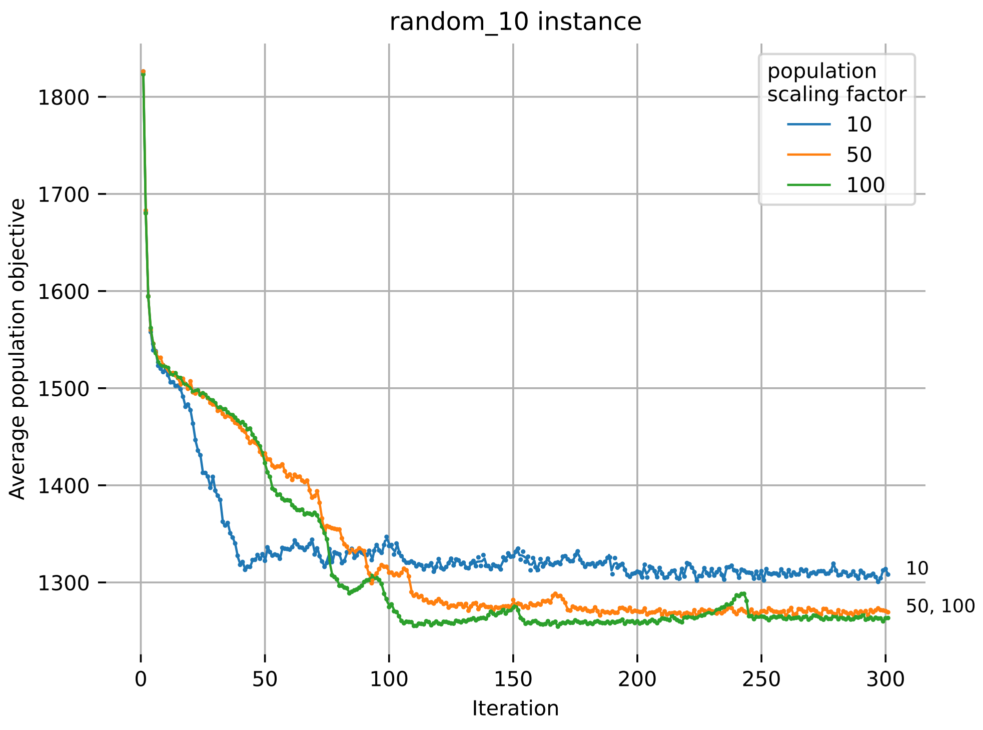


Figure 5.1 Testing maximum number of iterations at two random instances.



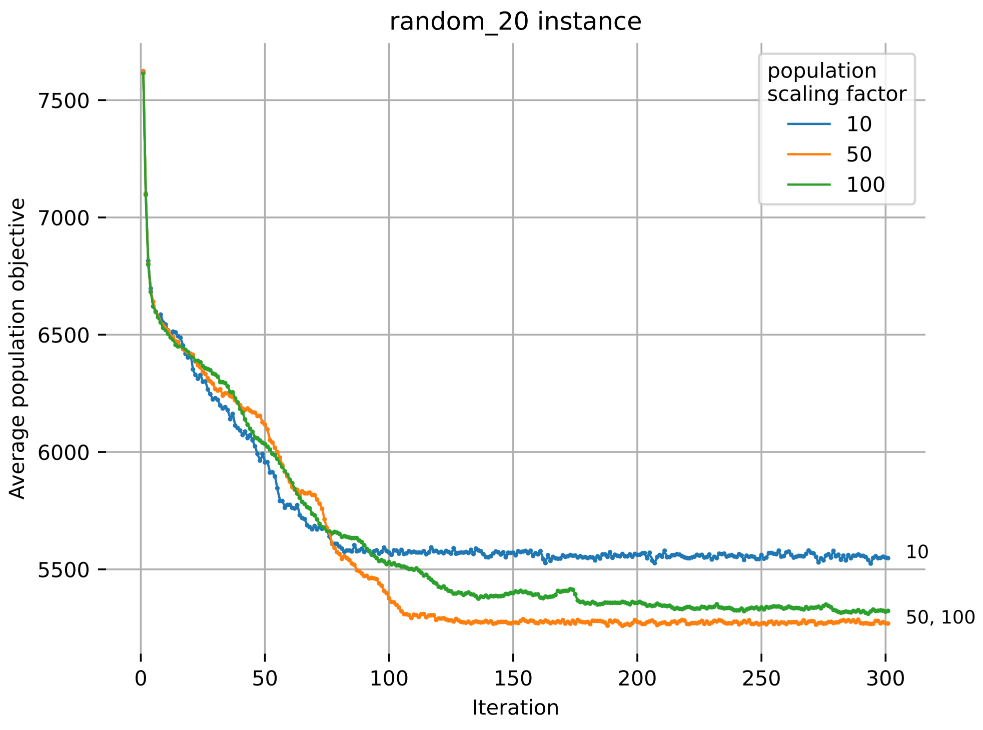
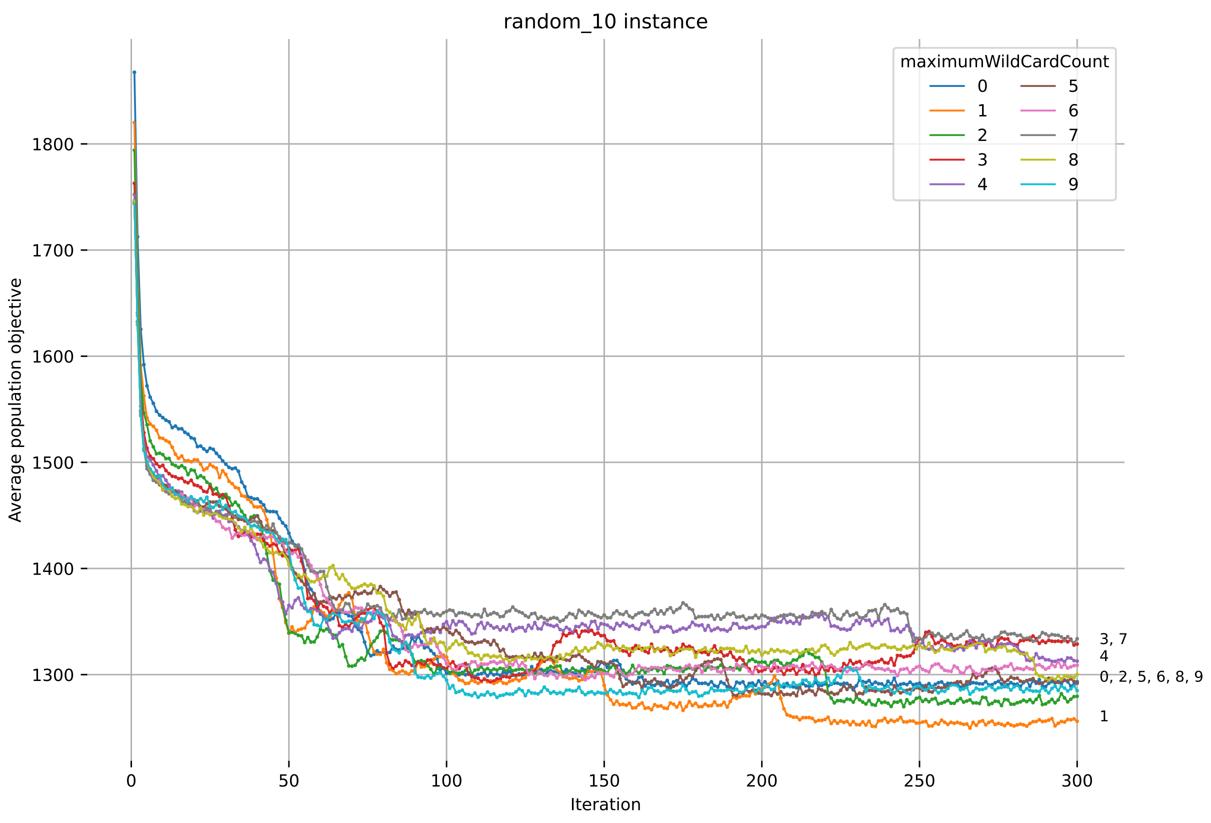


Figure 5.2 Testing population scaling factor at two random instances. The population size is κN for population scaling factor κ and instance of size N.



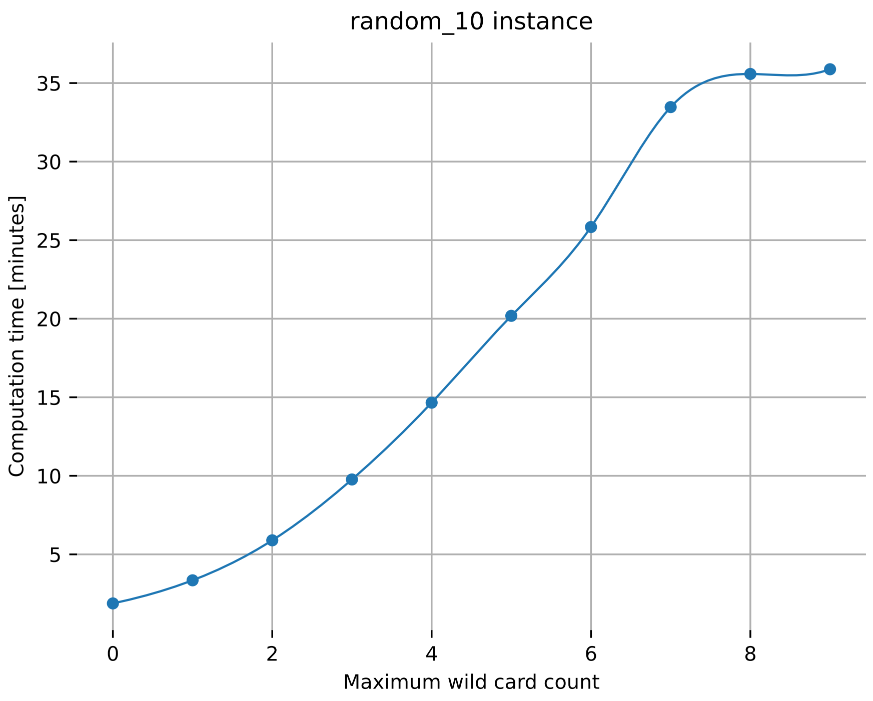
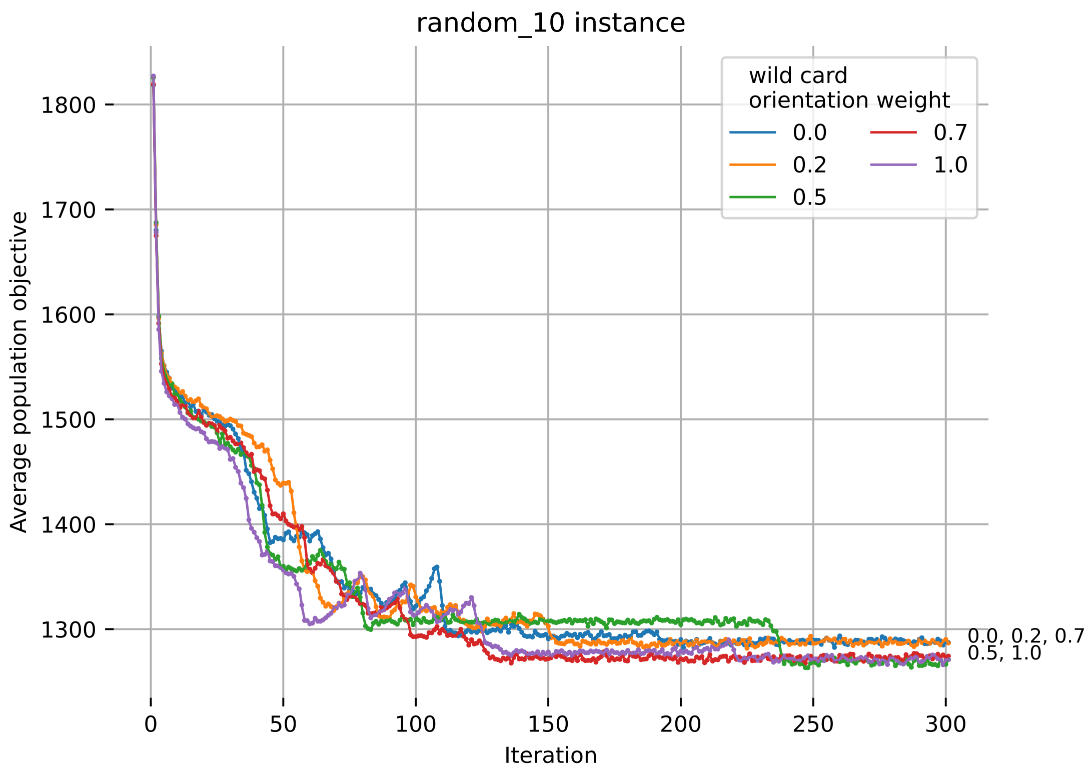


Figure 5.3 Testing increasing maximum wildcard count. Performance (top) and computation speed (bottom) are showed.



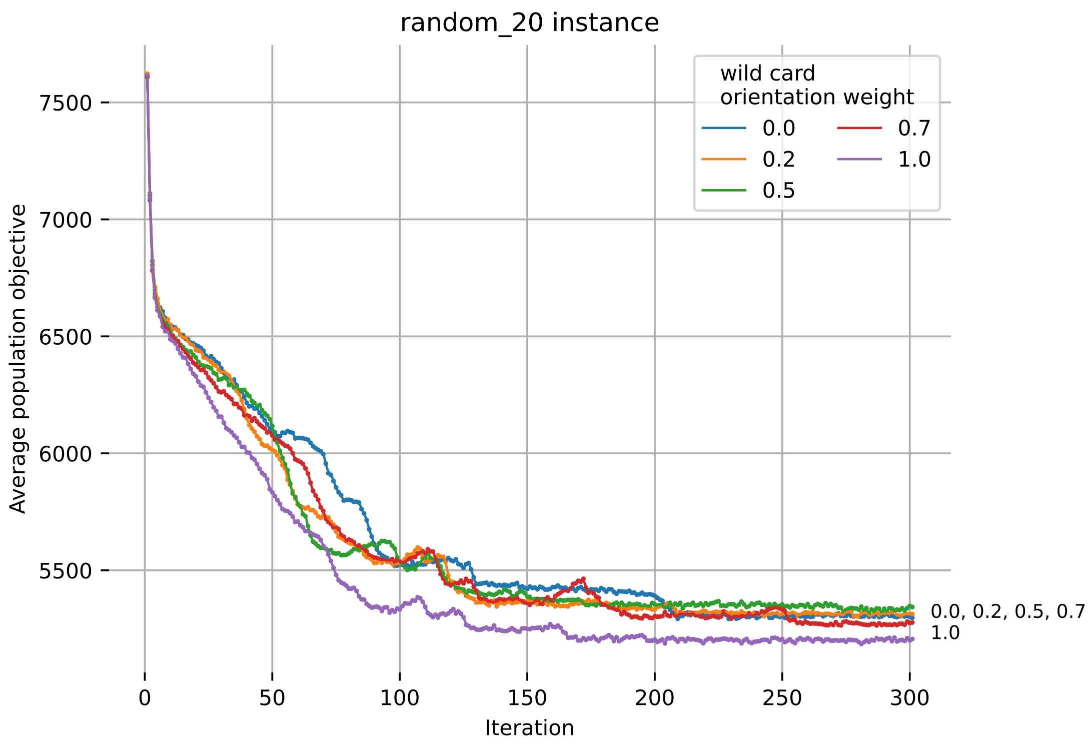
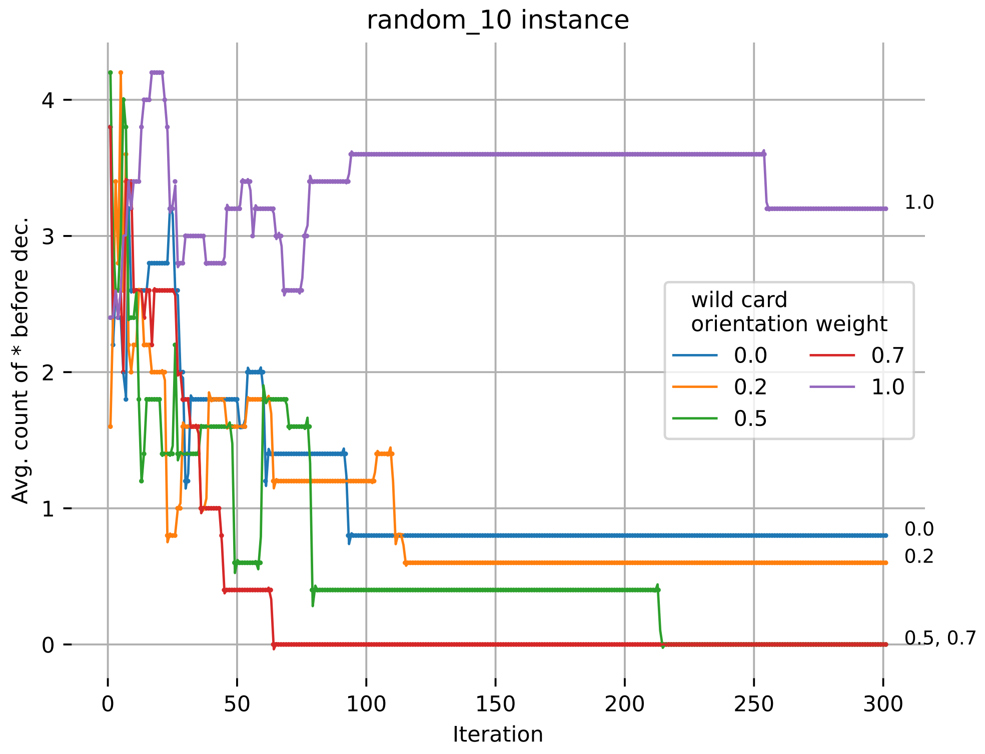


Figure 5.4 Testing performance of orientation weight for a wildcard cut type $*$ at two random instances. Hyperparameter `maximumWildCardCount` is 1.



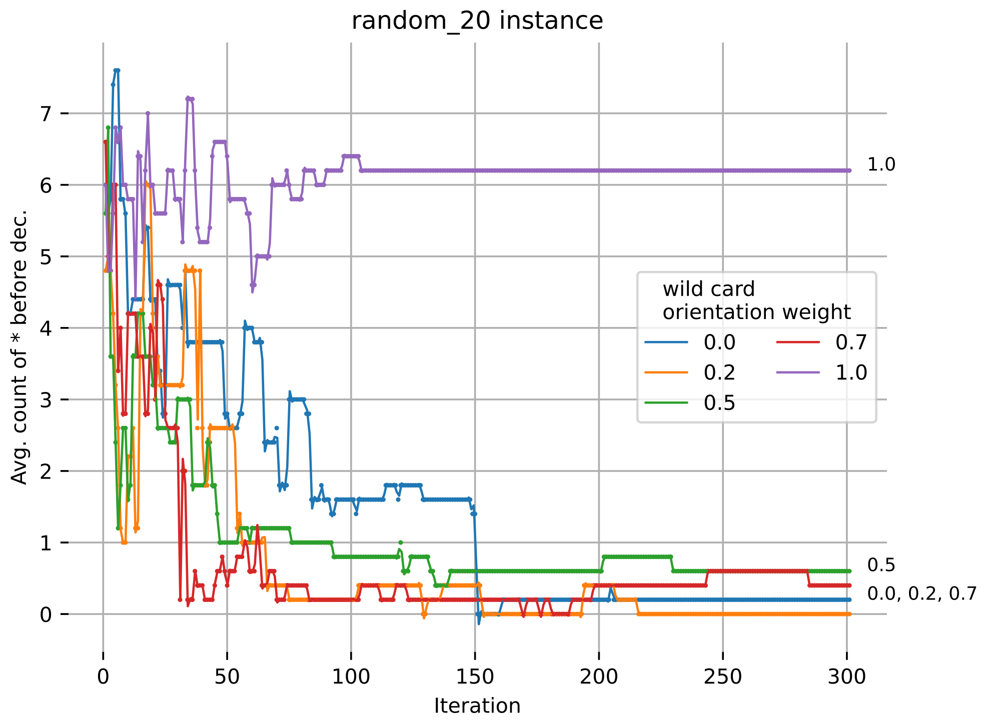
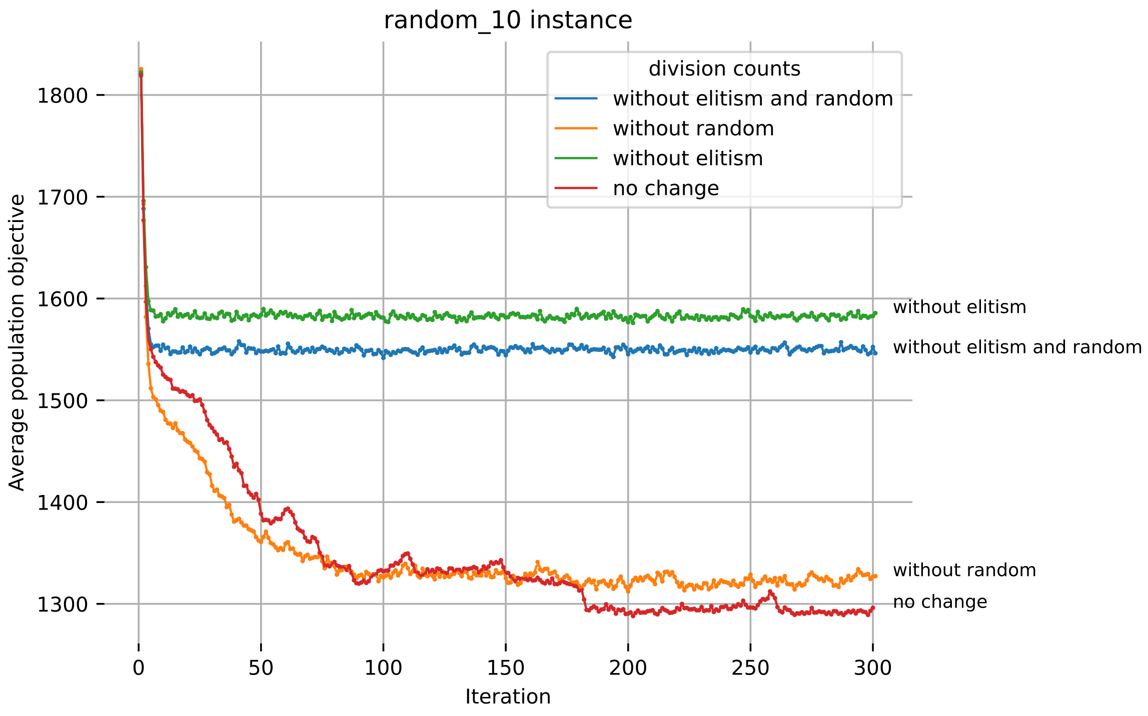


Figure 5.5 Testing wildcard cut type $*$ spread at two random instances. Each iteration shows an average count of wildcard cut type $*$ at best individual before decoding for different values of wildcard orientation weights. Hyperparameter `maximumWildCardCount` is 1.



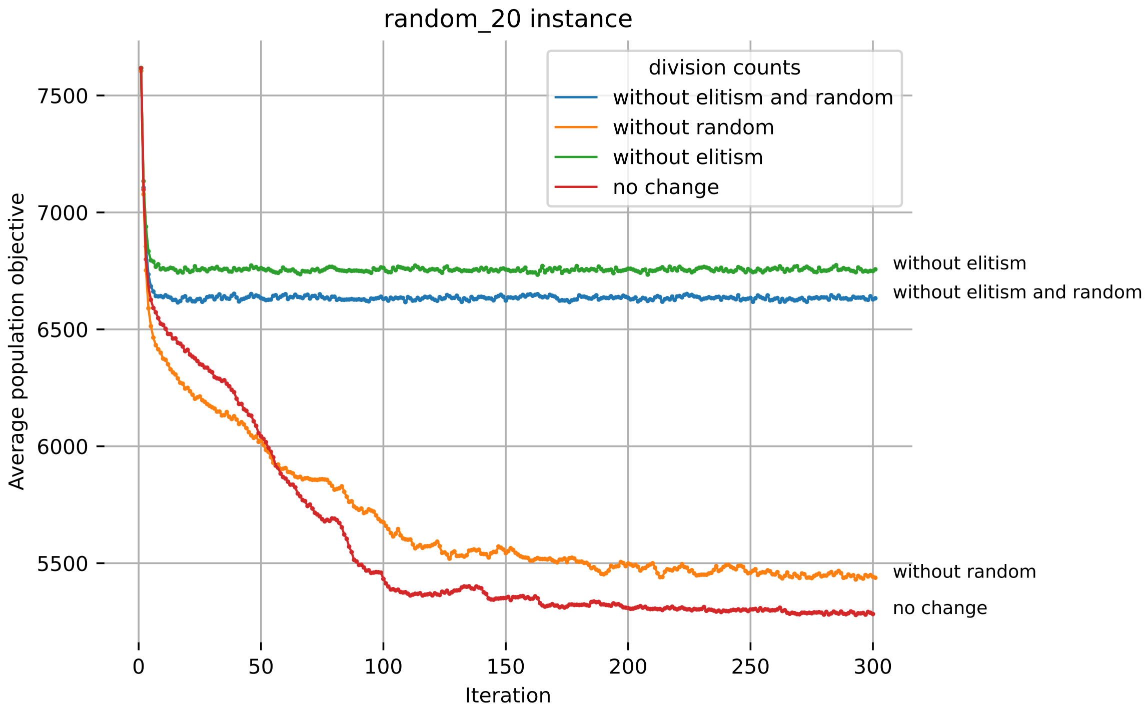
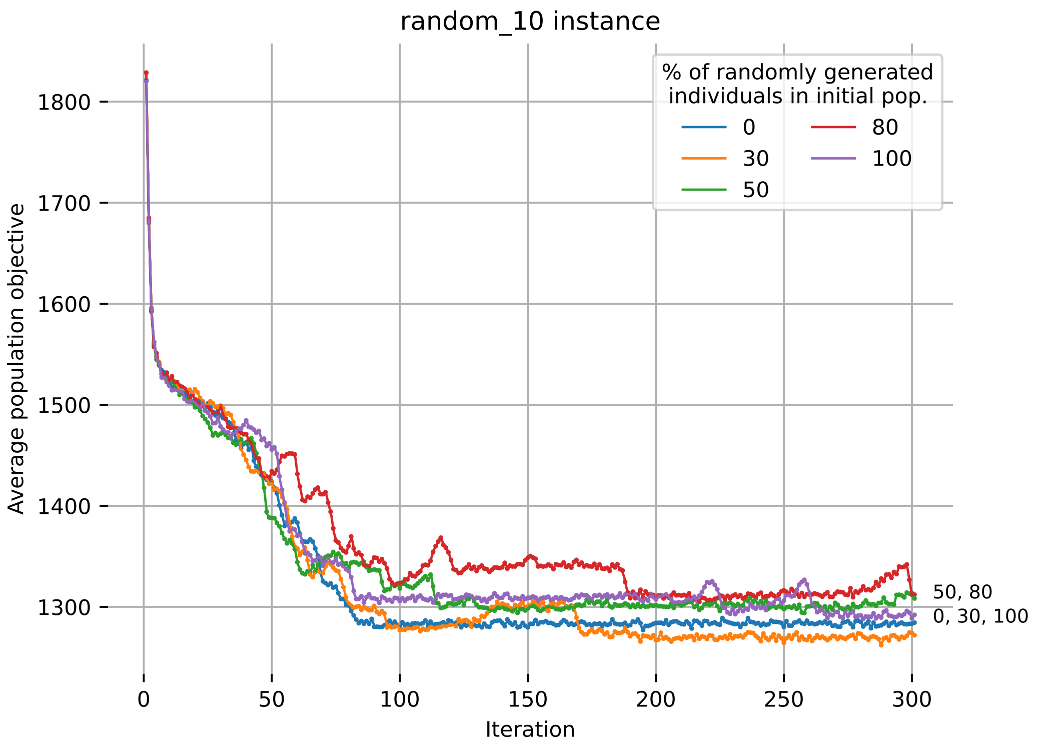


Figure 5.6 Testing population division counts at two random instances. Four variants are displayed. The first does not use elitism. The second does not inject random individuals. The third combines the first and second, and the last does not change the population division counts as described in listing 4.



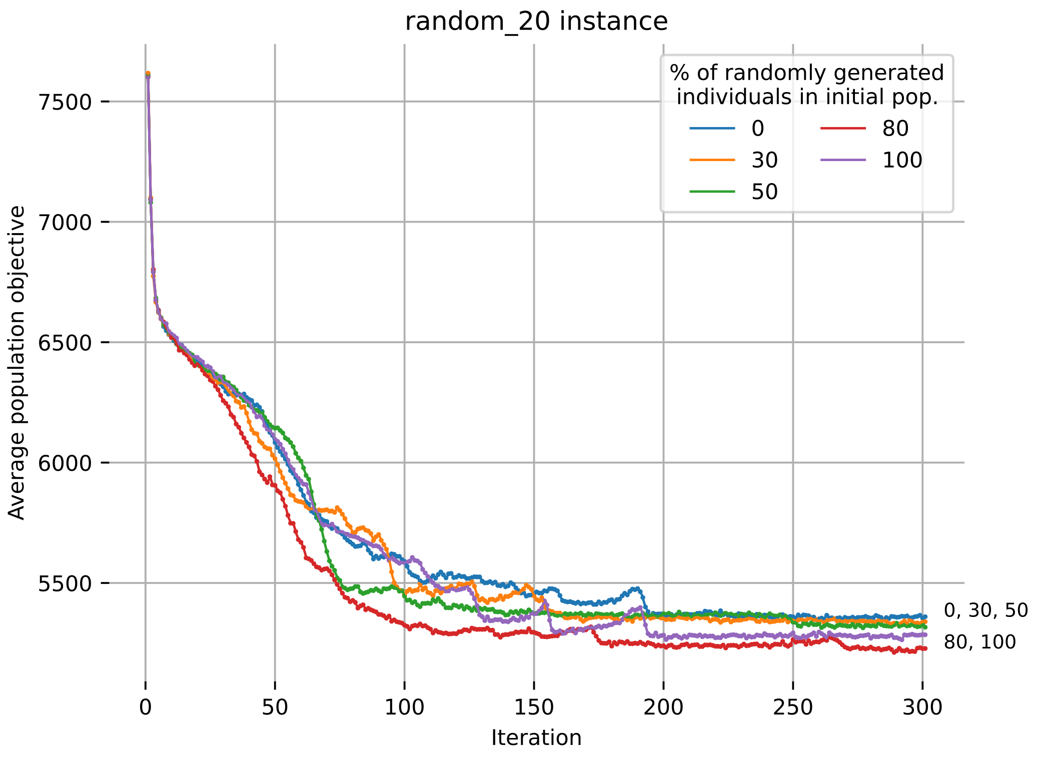
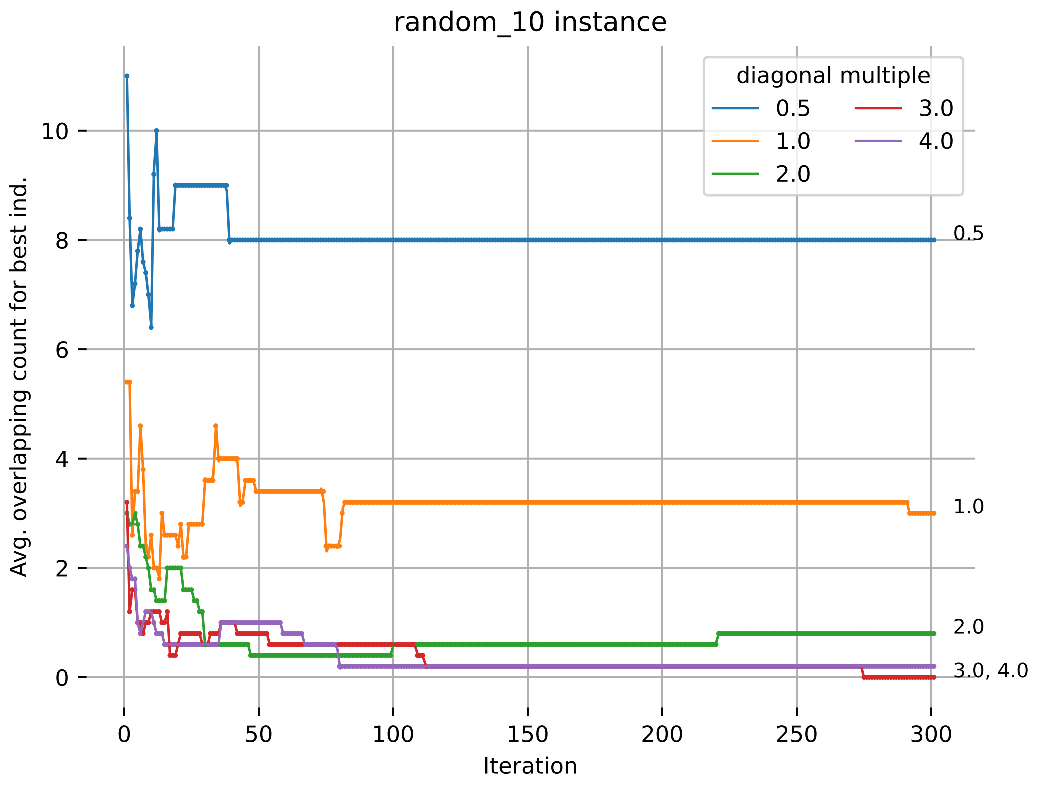


Figure 5.7 Testing initial population division counts at two random instances. The initial population consists of randomly and greedily generated individuals (left part of fig. 4.14).



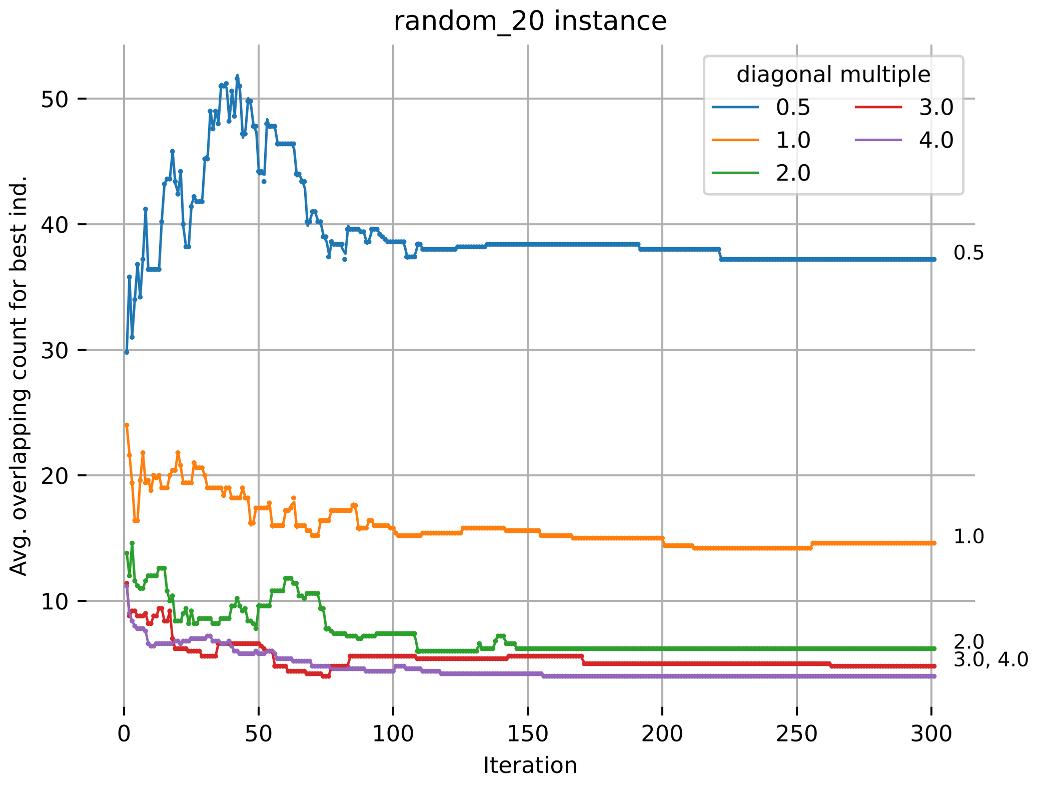


Figure 5.8 Testing overlapping penalization constant λ (eq. 3.3) at two random instances. It is calculated as ρD, where ρ is a diagonal multiple, and D is the length of a diagonal in a layout. Graphs show the average overlapping paintings count for the best individual at each iteration.

5.4 Results

This section presents painting placement solutions to the painting placement instances. Obtained solutions are discussed in the following subsections – random scenario instances in 5.4.1, packing scenario instances in 5.4.2, clustering scenario instances in 5.4.3, and London National Gallery instance in 5.4.4. Hyperparameter values used to obtain results in this section are in table 5.5. Statistics for the last iteration of the obtained results are in table 5.6.

Hyperparameter values in table 5.5 are set to their recommended values from hyperparameter testing in section 5.3. The hyperparameter not set to the recommended value is `maxNumberOfIter`. It is set to 500 instead of the recommended value of 300. The reason is to possibly find a better painting placement solution in exchange for more computation time.

The recommendation to remove orientation penalization by setting `orientationWeights` to $\langle 1, 1, 1 \rangle$ is followed. As described in hyperparameter testing, it should produce a population with a better on-average objective value and a faster-decreasing trend in objective value. Also, the population size is set to 75 times the instance size. It is the midpoint of the recommended interval 50–100. Lastly, population division is set to the same values as in listing 4. It means keeping an elitism strategy and injecting random individuals.

Table 5.5 Hyperparameter values used to obtain results

Hyperparameter	**Value**
`maxNumberOfIter`	500
`populationSize`	75 times the instance size
`maximumWildCardCount`	1
`orientationWeights`	$\langle 1, 1, 1 \rangle$
`populationDivisionCounts`	elitism, random
`initialPopulationDivisionCounts`	0.7 random, 0.3 greedy
`overlappingPenalizationConstant`	4 times the diagonal length of the layout
`outsideOfAllocatedAreaPenalizationConstant`	0

Hyperparameter description is in table 5.3.

Table 5.6 Statistics of the last iteration

Instance name	**Best obj. value**	**Worst obj. value**	**Obj. mean**	**Standard deviation**
random_10	1136.11	3438.01	1640.91	460.76
random_20	5417.8	11629.59	7209.22	1400.13
packing_10	669.48	2369.11	1142.96	383.22
packing_20	6219.13	12283.41	8140.29	1450.09
cluster_3_6	3921.68	11121.96	6317.36	1844.99
cluster_4_5	4887.04	12088.99	7177.9	1767.67
london_gallery_wall	2759.65	15764.31	5440.9	2633.81

Instance description is in table 5.2.

5.4.1 Random scenario

Visualization of the painting placement solution to the random_10 instance is in figure 5.9 and for random_20 instance in figure 5.10.

For the random_10 instance, we can see that there are no overlappings, and only painting 6 is partially outside the layout. Also, due to the evaluation function set to $x+y$, smaller paintings are placed towards the bottom left corner. This painting placement solution can be considered successful, because it adheres to all penalizations that are applied.

For the random_20 instance, we can see that there are four overlappings. Similarly to the random_10 instance, the evaluation function forced the placement of the smaller paintings to the bottom left corner, and one painting is placed partially outside the layout. Due to the four overlappings, it is not a fully successful painting placement solution. On the other hand, the solution adheres to all other penalizations. It might suggest further increasing the overlapping penalization constant λ.

5.4.2 Packing scenario

Visualization of the painting placement solution to the packing_10 instance is in figure 5.11 and for packing_20 instance in figure 5.12.

For the packing_10 instance, we can see no overlappings, and paintings 2,3,4 are partially outside the layout. It is considered a success, as the packing scenario tests the ability to form compact solutions by setting the instance generation parameter layout area ratio to one, which means that the area of the layout equals the area of all paintings summed together. However, an improvement in compactness can still be gained by moving paintings 1 and 2 downwards. It could be implemented using a post-optimization, as suggested in subsection 6.2.5. The evaluation function is absent in the packing scenario, so there is no preference for any part of the layout.

For the packing_20 instance, we can see seven overlapping pairs. It is not considered a success as there is no evaluation function present and overlapping penalization constant λ should be the main source of improvement in the objective function 3.3. However, the packing complexity for 20 paintings is high, and it might be impossible to fit all paintings to the layout without overlapping or being outside the allocated area. Also, randomly generated flow between paintings might play a role. Nevertheless, the suggestion is to increase the overlapping penalization constant λ.

5.4.3 Clustering scenario

Visualization of the painting placement solution to the cluster_3_6 instance is in figure 5.13 and for cluster_4_5 instance in figure 5.14.

For cluster_3_6 instance, we can see the formation of clusters. It is a success as the proposed solution can recognize the information encoded in the flow between paintings. However, we can see that there are seven overlappings. An interesting fact is that these overlappings are only inside the same clusters. It suggests that the flow between paintings is set too high, so it is advantageous to create overlaps and thus decrease the flow.

For the cluster_4_5 instance, we can see a partially successful formation of clusters. It still suffers from the problems of overlapping paintings in the same cluster. However, there are also several overlappings between paintings from different clusters. It might suggest that the overlapping penalization and flow are more balanced.

5.4.4 London Gallery Wall

Visualization of the painting placement solution to the london_gallery_wall instance is in figure 5.15. The instance is created from the painting placement at the London National Gallery from figure 1.1. Also, the flow between paintings is set to reassemble the relative positions from that figure (concrete flow values are in the dataset in the attached medium).

We can see that the obtained solution contains no overlappings, and two paintings are partially placed outside the layout. Also, most of the relative positions of the paintings are successfully reconstructed using the flow. The difference is that the proposed solution placed the largest middle painting to the right of the layout instead of in the middle, as seen in the original figure. However, the flow can be changed to obtain different results.

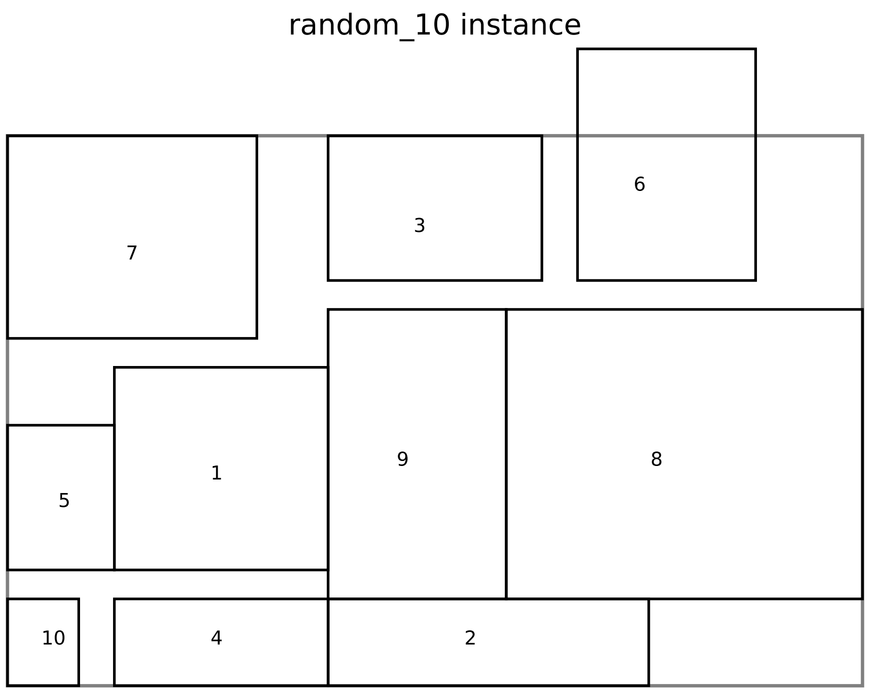


Figure 5.9 Painting placement solution for the random_10 instance. There are no overlappings.

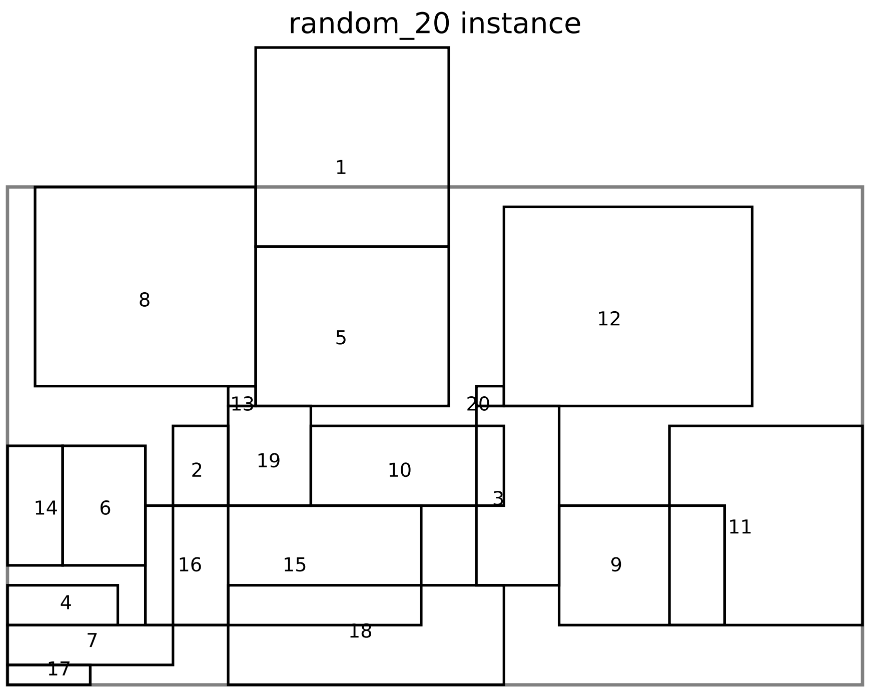
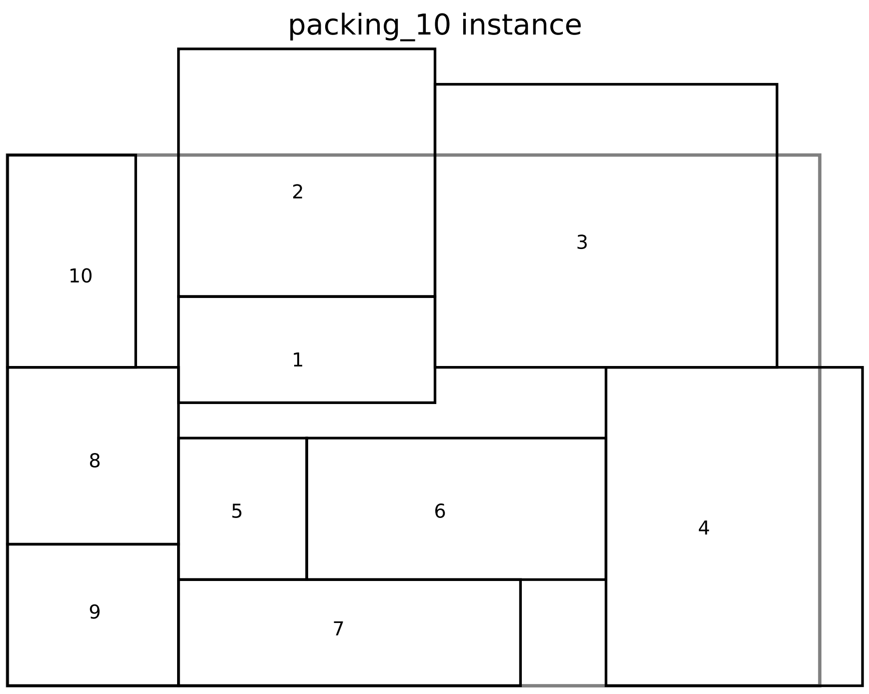
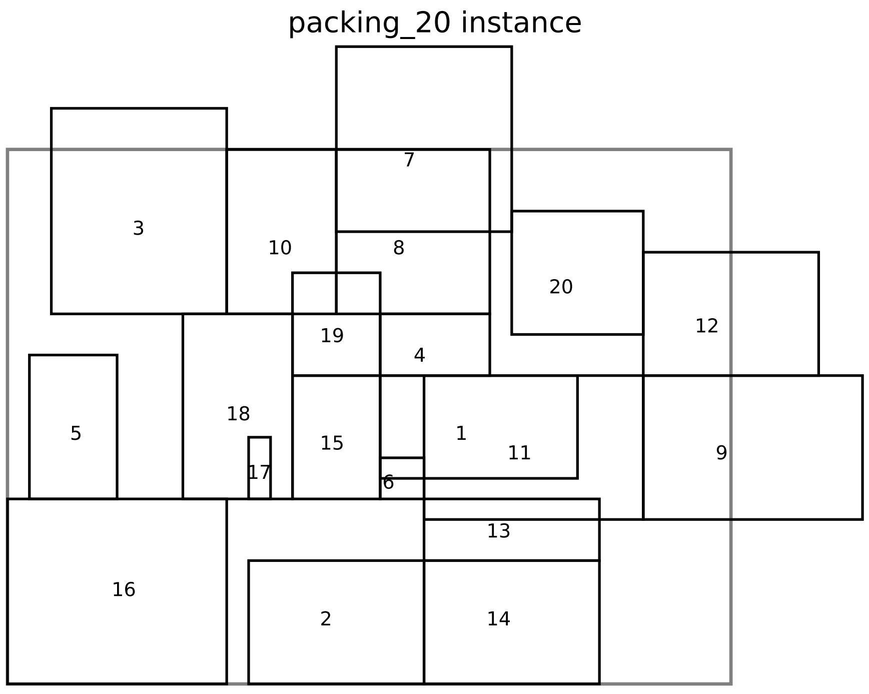


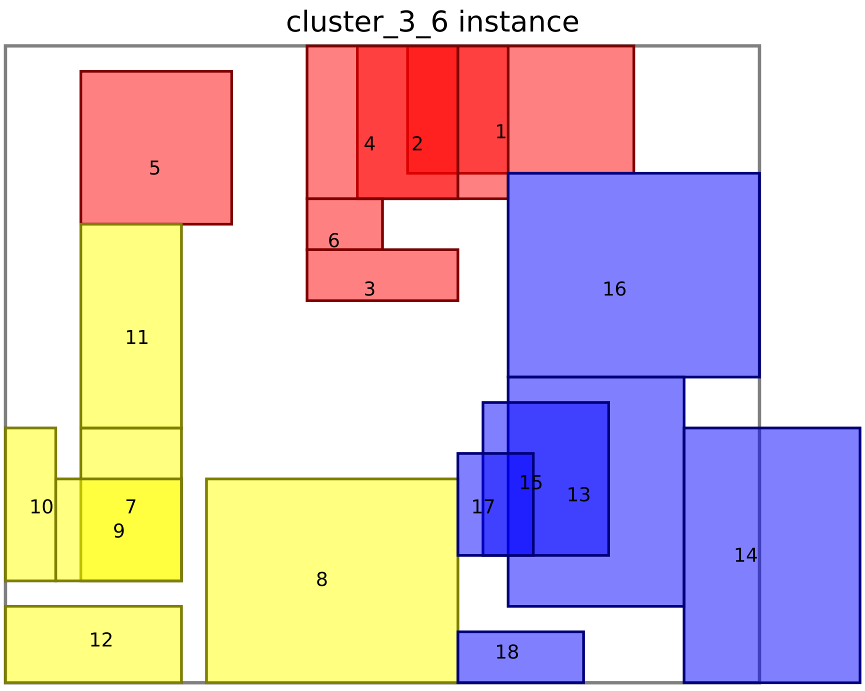
Figure 5.10 Painting placement solution for the random_20 instance. Four overlapping pairs exist (15–18, 15–16, 3–10, 9–11).



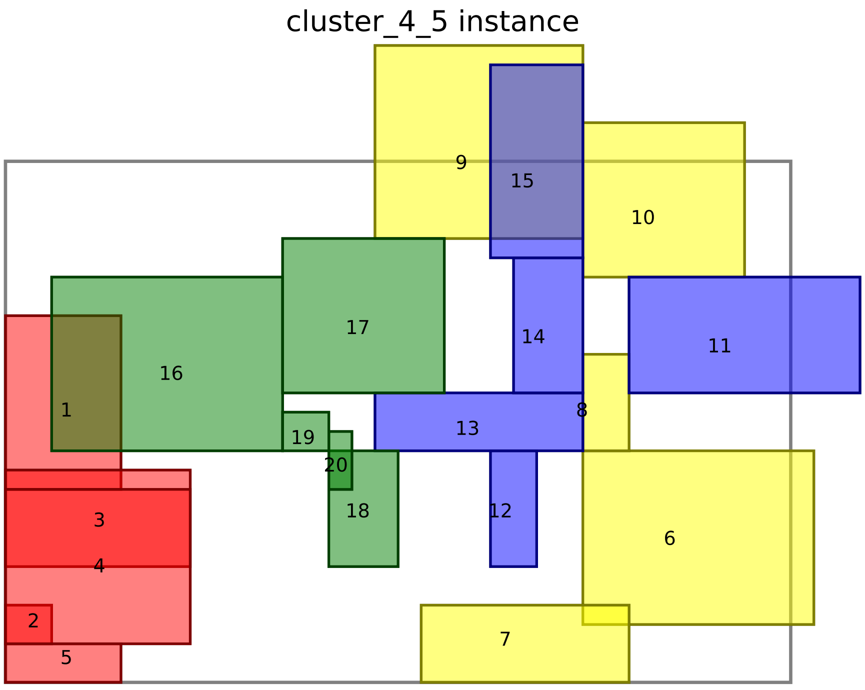
■ **Figure 5.11** Painting placement solution for the packing_10 instance. There are no overlappings.



■ **Figure 5.12** Painting placement solution for the packing_20 instance. There are seven overlapping pairs (17–18, 1–6, 1–11, 11–13, 10–19, 8–19, 7–8).



■ **Figure 5.13** Painting placement solution for the cluster__3__6 instance. Three groups of paintings, 1 to 6, 7 to 12, and 13 to 18, are marked using different colors. There are seven overlapping pairs.



■ **Figure 5.14** Painting placement solution for the cluster__4__5 instance. Four groups of paintings, 1 to 5, 6 to 10, 11 to 15, and 15 to 20, are marked using different colors. There are seven overlapping pairs.

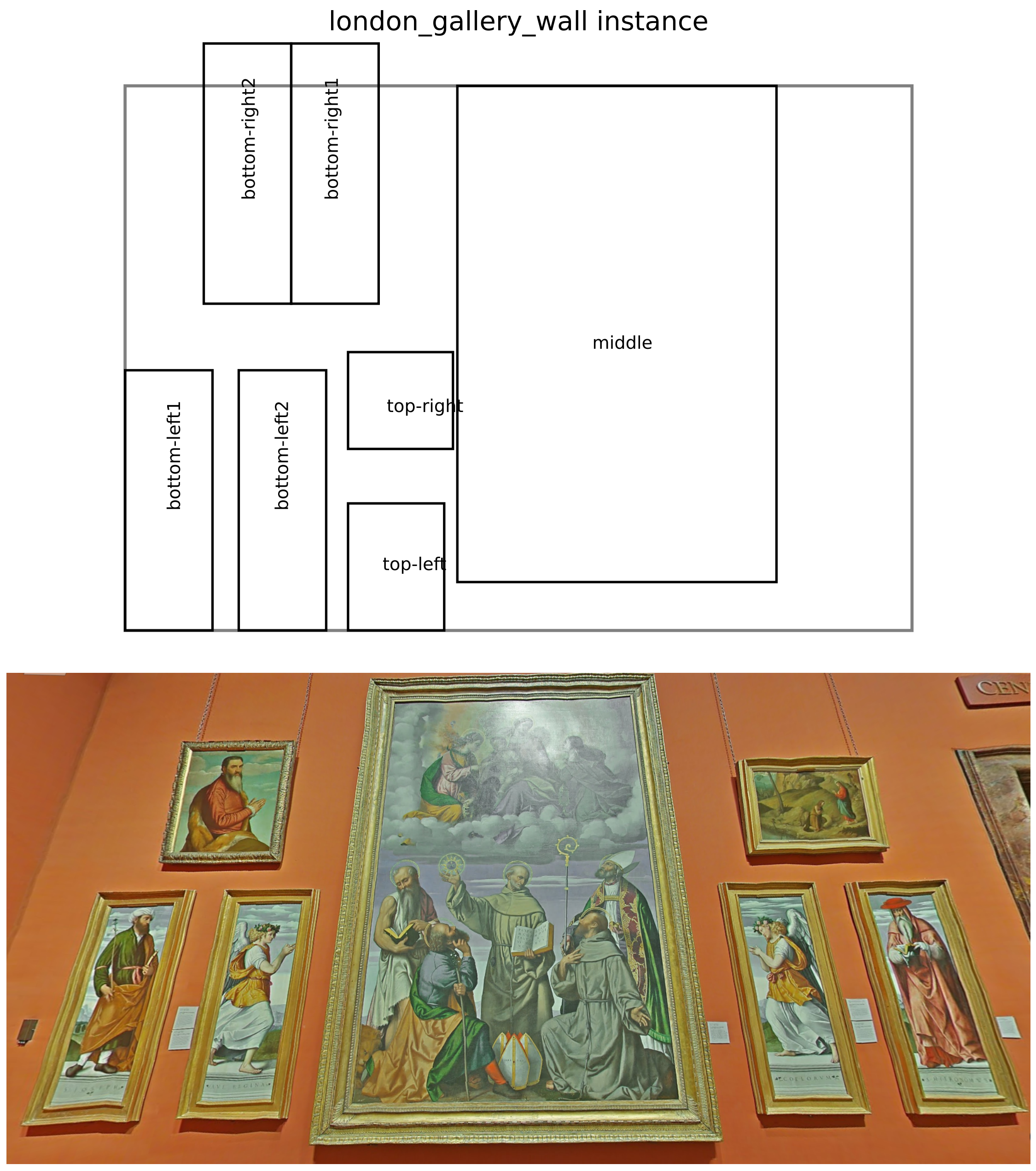


Figure 5.15 Painting placement solution for the london_gallery_wall instance (top) and the original painting placement at the London National Gallery (bottom) from figure 1.1.

5.5 Implementation

The proposed implementation of a genetic approach is written in Java 11 using a Play Framework v2.8[1], a web framework for Java and Scala.

Implementation behaves like a computation server to which a user can submit a computation. Then, the server asynchronously starts the submitted computation and returns an identifier of the computation. It means that multiple computations can be submitted without blocking the user. The user can then check the computation state using the returned identifier.

To start the computation server, locate the directory containing a file `build.sbt` in the attached medium (see appendix A). Then, run the following command in that directory (Java 11, SBT[3], and Scala must be installed).

```
$ sbt run
```

Code listing 1 Starting a computation server.

Command in listing 1 uses `sbt`[3] with `run` argument to start the computation server. By default, the computation server accepts requests on `localhost:9000`. An example of submitting a computation with a predefined instance name to the computation server is in listing 4. An example of a successful computation submission response is in listing 2.

```
{
    "id":"random_10_9B5F8",
    "outputDirectory":"./out/088_random_10_9B5F8"
}
```

Code listing 2 Successful computation submission response.

The computation server also validates input before starting the computation. For example, if misspelling the instance name, the response by the computation server can be seen in 3.

```
{
    "message":"Entity [DatasetDto] with identifier [randomm_10] was not found."
}
```

Code listing 3 Unsuccessful computation submission response.

Lastly, there is also an option not to specify the instance name in the computation submission. In that case, a user has to specify the instance manually in the request – layout width and height, paintings together with their flow and evaluation function π (eq. 3.3), in the format that is accepted by mXparser[4]. An example of submission without specifying the instance name is in the appendix in listing 5.

[1]https://www.playframework.com/documentation/2.8.x/Home
[2]https://curl.se/
[3]https://www.scala-sbt.org/
[4]https://mathparser.org/

```
$ curl --location 'localhost:9000/compute/dataset' \
--header 'Content-Type: application/json' \
--data '{
  "datasetName": "random_10",
  "gaParameters": {
    "maxNumberOfIter": 300,
    "populationSize": 500,
    "maximumWildCardCount": 1,
    "orientationWeights": [
      1,
      1,
      0.5
    ],
    "geneticAlgorithm": "simpleGa",
    "mate": "normalizedProbabilityVectorSum",
    "mutate": "flipOnePartAtRandom",
    "select": "tournament",
    "objective": "simple",
    "evaluator": "ga",
    "placingHeuristics": "corner",
    "populationDivisionCounts": {
      "elite": 0.2,
      "average": 0.6,
      "worst": 0.2,
      "children": 0.3,
      "mutant": 0.2,
      "winner": 0.2,
      "random": 0.1
    },
    "initialPopulationDivisionCounts": {
      "random": 0.7,
      "greedy": 0.3
    }
  },
  "objectiveParameters": {
    "name": "simple",
    "params": {
      "overlappingPenalizationConstant": 30.61,
      "outsideOfAllocatedAreaPenalizationConstant": 30.61
    }
  }
}'
```

Code listing 4 Example of computation submission of random_10 instance using `curl`[2] to a computation server running on `localhost:9000`.

Chapter 6

Discussion

This chapter summarizes and further discusses the computational results of the proposed solution and comments on the hyperparameter testing in section 6.1. Also, further improvements, extensions, and future work is discussed in section 6.2.

6.1 Computational result discussion

This section summarizes and discusses the computational results from chapter 5. As mentioned in the previous chapters, the process of obtaining the results is first to define four testing scenarios. They are random, clustering, packing, and London National Gallery. Then, for each scenario, painting placement instances are created. Using random scenario instances, reasonable hyperparameter values for the genetic algorithm 2 are determined. Lastly, painting placement solutions are computed for all generated instances using these hyperparameters.

The proposed solution produces good results for the smaller instances, i.e., random_10 instance in figure 5.9 and packing_10 instance in figure 5.11. ***Good result*** is a painting placement solution that respects the flow and evaluation function and has few overlapping paintings and paintings partially or fully outside the allocated area. It is all achieved in the smaller instances.

Overall, the biggest issue for larger instances, e.g., random_20 instance in figure 5.10 and packing_20 instance in figure 5.12, are overlapping paintings. It can be mitigated by setting a higher value for the overlapping penalization constant λ. However, by increasing the value, other parts of the objective function 3.3 become insignificant and thus ignored by the proposed solution. For example, by overly increasing the overlapping penalization constant, the evaluation function or the flow between paintings is ignored because the overlapping cost is too high.

Another issue is the extrapolation or generalization of hyperparameters from random scenarios to other scenarios. Indeed, fine-tuning hyperparameters for each instance would produce painting placement solutions with better on-average population objective value. However, it is computationally expensive to do so. Despite that, testing hyperparameters on random scenarios provides valuable insight into the proposed solution. For example, one insight is that elitism is integral to the reproductive plan (see 5.3.5).

Lastly, the outside of allocated area penalization constant γ is set to zero at all presented painting placement solutions. It can be thus removed from the objective function 3.3 entirely. The idea behind introducing it is to penalize partially or fully placed paintings outside the layout. However, as described in 5.3.8, this hyperparameter fails to do so. One solution is to replace γ with a different one, which penalizes solutions that place paintings partially or fully outside the layout.

6.2 Improvements, extensions, future work

This section suggests further improvements and extensions to the proposed solution and future work. Improvements and extensions are minor or moderate changes that, according to the author, can be implemented without much difficulty and the need for modifying much of the proposed solution. However, each improvement or extension might become a new idea and thus produce a topic for future work.

6.2.1 Free space

The extension of the proposed solution is to take a different approach to ***free space***, which is part of the painting placement solution where the paintings are not placed.

Free space is used to separate paintings on the wall. It is also used for other related problems. For example, free space is important in the FLP problem (sec. 2.1) as there is a need for an aisle between the facilities through which the material transportation takes place [27].

In the proposed solution, two main parts influence where the free space is created. It is (a) the placing heuristic and (b) the evaluation function π (eq. 3.3). Placing heuristic works locally, i.e., only in the allocated area for the painting, and the evaluation function, although it might be used to define arbitrary free space shape, is not a constraint but a penalization. Thus, it does not guarantee that the painting placement solution creates a solution with the desired free space.

One possible approach to guarantee free space is the introduction of ***dummy paintings***. These dummy paintings can be injected during the slicing tree construction. The resulting painting placement solution will thus, among the paintings, contain free space occupied by dummy paintings. They are then removed, which creates the free space.

An example of dummy painting injection can be seen in figure 6.1. The free space is created between paintings 1,2 and 3 by adding a vertical cut V to the tree.

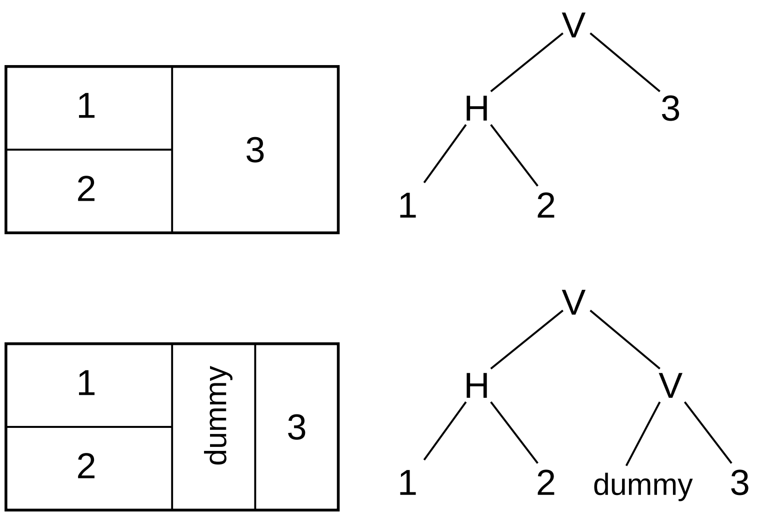
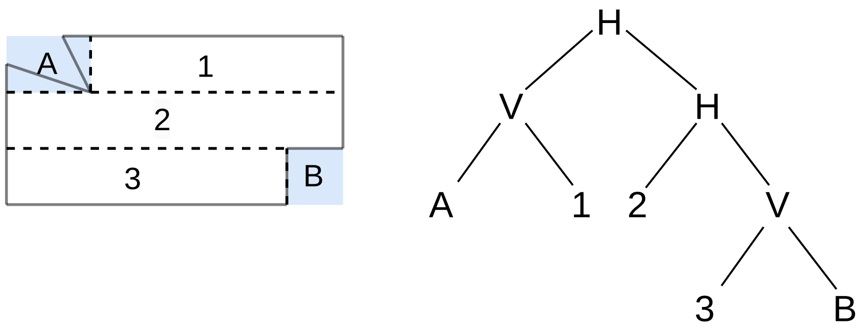


Figure 6.1 Example of dummy painting injection to guarantee the free space between paintings 1,2 and 3.

6.2.2 Non-rectangular layouts

Another extension to the proposed solution is adding the ability to work with layouts that are not rectangular. It can be solved using the dummy paintings described in subsection 6.2.1. These dummy paintings must be created as small as possible and placed over the parts of the layout that are not rectangular. By placing such created dummy facilities, the layout becomes rectangular. A similar approach is used in [27] to modify a slicing tree to solve FLP (sec. 2.1).

An example of using dummy paintings to work with a non-rectangular layout is in figure 6.2. There are two irregularities in both corners of the layout. Two dummy paintings are injected into the slicing tree to fill the irregularities.



■ **Figure 6.2** Example of working with non-rectangular layout. The allocated area is marked using a dotted line. Dummy paintings that fill the irregularities in both corners are A and B.

6.2.3 Non-rectangular paintings

Another extension is to allow painting shapes that are not rectangular. This problem can be easily solved by representing a non-rectangular painting as the smallest possible rectangle to which the painting fits. By using this approach, the proposed implementation can work with non-rectangular paintings. One possible drawback is that the painting placement solution might become more sparse, i.e., containing too much free space. However, this can be solved using a post-optimization heuristic that tries to reduce a free space of a painting placement solution.

6.2.4 Placing heuristic

Instead of using the placement heuristic described in subsection 4.3.3, a different one can be used. One candidate is a heuristic that, instead of trying to place painting in the corners of the allocated area, tries to place the painting at all possible placement points, e.g., to place the painting in the middle of the allocated area. However, using this solution might be computationally expensive. On the other hand, a heuristic that only tries the bottom-left of the allocated area as a placement point can be much faster but might not produce good results.

Another approach is calculating suboptimal or optimal placement points inside the allocated area. Then, move the painting as close as possible to that point. A similar approach is used in [2] for solving FLP (sec. 2.1). They move the facility centroid as close as possible to the calculated unconstrained optimum without leaving the boundaries where the facility can be placed.

6.2.5 Post-optimization

One interesting idea is to introduce post-optimization. It is a process that takes the result, in this case, the painting placement solution, and tries to improve it. For example, if there is insufficient free space between paintings, they can be moved by the post-optimization process. Another example will be if the goal is to create the most compact layout. Then, a solution can be to use the compaction operation proposed in [23], which tries to reduce free space as much as possible.

6.2.6 Extension to other problems

The solution proposed in this method can be applied to other problems. The central part of the thesis is the coding of an individual and crossover. These can be used to solve problems that optimize some permutation of elements.

One concrete example is 2D-KP problem with rectangular pieces [28], where the objective is to place as many rectangles in a container as possible, minimizing unused space. The solution proposed in this thesis can solve the 2D-KP problem with rectangular pieces by using unchanged individual representation (sec. 4.2), unchanged decoding procedure (subsec. 4.3.1), and unchanged slicing layout construction (subsec. 4.3.2). Then, the problem-specific placement heuristic places the rectangles, followed by BL (Bottom Left) heuristic [29] that minimized unused space.

6.2.7 Deciding which painting to place

Deciding which painting to place can happen when the wall area is insufficient to place all paintings. Thus, some subset of paintings has to be selected and placed. A similar problem is solved in the shelf-space planning problem (sec. 2.2), where the retailer has to decide which goods to place on shelves to achieve maximum profits.

6.2.8 Multiple walls

The painting placement problem can be extended to contain multiple walls instead of one. Additionally, the walls might have some interactions between them, e.g., placing a painting on the first wall limits the subset of paintings that can be placed on the second wall. A similar problem is solved in the shelf-space planning problem (sec. 2.2), where there are multiple shelves at the same time.

6.2.9 Human operator assistance

Painting placement solution, which is some placement of paintings, can be easily visualized. Thus, it can be presented to a human operator that will further modify it.

For example, FLP (sec. 2.1) creates a layout for multiple facilities. A human engineer that creates the plan for placing these facilities might use the output of an algorithm as a starting point. Then, the engineer can modify it by increasing the aisle size or changing the position of some facilities.

Chapter 7
Conclusion

The central part of the thesis was to propose a genetic approach for solving the painting placement problem. It was accomplished by creating a genetic algorithm with novel individual representation as multiple stochastic vectors and novel crossover as vector addition, followed by normalization back to the stochastic vector. Subsequent parts and goals of the thesis were defined in the introduction chapter 1. All of them were accomplished and presented in this thesis.

1. The first goal was to define a painting placement problem and what a solution is. It was defined in terms of a painting placement instance, which consists of paintings, flow between paintings, layout, and evaluation function. The solution was defined as a sequence of placement points for the paintings.

2. The second goal was to create a dataset for the painting placement problem. Four scenarios were defined – random, clustering, packing, and London National Gallery. Multiple instances of the painting placement problem were created for these scenarios.

3. The thesis's third and central goal was to propose and implement a genetic approach for solving the painting placement problem. As described above, the novel genetic approach represents an individual as multiple stochastic vectors. Then, the crossover is implemented as vector addition, followed by normalization back to the stochastic vector. These vectors decode into one slicing tree, which recursively divides the space or wall where the paintings are placed. The second novel approach was to add a wildcard symbol $*$ to the possible values contained in the internal node of a slicing tree. Wildcard symbol $*$ can take up any value – H for horizontal cut and V for vertical cut.

4. The fourth goal was to evaluate the performance of the proposed genetic approach. It was achieved by creating a computational server to which a painting placement instance can be submitted and its painting placement solution obtained. Each instance in the dataset was submitted multiple times to the computation server to account for the statistical significance of the obtained results. These results were then presented and discussed.

5. The fifth and last goal was to discuss the results and suggest further improvements, extensions, and future work. Suggestions and result discussion were presented in chapter 6. One suggestion was creating an empty space inside the resulting layout by injecting dummy paintings into the slicing tree. This injection could be further used to work with non-rectangular layouts. Additionally, human operator assistance was mentioned, where the painting placement solution is presented to a human operator that further modifies it to his/her needs.

Bibliography

1. *Screenshot of a Wall from Google Virtual Tour at the The National Gallery, London* [online]. [N.d.] [visited on 2023-04-02]. Available from: `https://www.nationalgallery.org.uk/visiting/virtual-tours/google-virtual-tour`.
2. GONÇALVES, José Fernando; RESENDE, Mauricio G.C. A Biased Random-Key Genetic Algorithm for the Unequal Area Facility Layout Problem. *European Journal of Operational Research*. 2015, vol. 246, no. 1, pp. 86–107. ISSN 03772217. Available from DOI: `10.1016/j.ejor.2015.04.029`.
3. YANG, Ming-Hsien; CHEN, Wen-Cher. A Study on Shelf Space Allocation and Management. *International Journal of Production Economics*. 1999, vol. 60–61, pp. 309–317. ISSN 09255273. Available from DOI: `10.1016/S0925-5273(98)00134-0`.
4. HWANG, Hark; CHOI, Bum; LEE, Grimi. A Genetic Algorithm Approach to an Integrated Problem of Shelf Space Design and Item Allocation. *Computers & Industrial Engineering*. 2009, vol. 56, no. 3, pp. 809–820. ISSN 03608352. Available from DOI: `10.1016/j.cie.2008.09.012`.
5. LIU, Jingfa; ZHANG, Huiyun; HE, Kun; JIANG, Shengyi. Multi-Objective Particle Swarm Optimization Algorithm Based on Objective Space Division for the Unequal-Area Facility Layout Problem. *Expert Systems with Applications*. 2018, vol. 102, pp. 179–192. ISSN 09574174. Available from DOI: `10.1016/j.eswa.2018.02.035`.
6. FRIEDRICH, Christian; KLAUSNITZER, Armin; LASCH, Rainer. Integrated Slicing Tree Approach for Solving the Facility Layout Problem with Input and Output Locations Based on Contour Distance. *European Journal of Operational Research*. 2018, vol. 270, no. 3, pp. 837–851. ISSN 03772217. Available from DOI: `10.1016/j.ejor.2018.01.001`.
7. TAM, Kar Yan; LI, Shih Gong. A Hierarchical Approach to the Facility Layout Problem. *International Journal of Production Research*. 1991, vol. 29, no. 1, pp. 165–184. ISSN 0020-7543, ISSN 1366-588X. Available from DOI: `10.1080/00207549108930055`.
8. DUNKER, T; RADONS, G; WESTKÄMPER, E. A Coevolutionary Algorithm for a Facility Layout Problem. *International Journal of Production Research*. 2003, vol. 41, no. 15, pp. 3479–3500. ISSN 0020-7543, ISSN 1366-588X. Available from DOI: `10.1080/0020754031000118125`.
9. LIU, Qi; MELLER, Russell D. A Sequence-Pair Representation and MIP-model-based Heuristic for the Facility Layout Problem with Rectangular Departments. *IIE Transactions*. 2007, vol. 39, no. 4, pp. 377–394. ISSN 0740-817X, ISSN 1545-8830. Available from DOI: `10.1080/07408170600844108`.

10. CHANG, Mei-Shiang; KU, Ting-Chen. A Slicing Tree Representation and QCP-Model-Based Heuristic Algorithm for the Unequal-Area Block Facility Layout Problem. *Mathematical Problems in Engineering*. 2013, vol. 2013, pp. 1–19. ISSN 1024-123X, ISSN 1563-5147. Available from DOI: `10.1155/2013/853586`.
11. KAR YAN TAM. Genetic Algorithms, Function Optimization, and Facility Layout Design. *European Journal of Operational Research*. 1992, vol. 63, no. 2, pp. 322–346. ISSN 03772217. Available from DOI: `10.1016/0377-2217(92)90034-7`.
12. BIANCHI-AGUIAR, Teresa; HÜBNER, Alexander; CARRAVILLA, Maria Antónia; OLIVEIRA, José Fernando. Retail Shelf Space Planning Problems: A Comprehensive Review and Classification Framework. *European Journal of Operational Research*. 2021, vol. 289, no. 1, pp. 1–16. ISSN 03772217. Available from DOI: `10.1016/j.ejor.2020.06.018`.
13. HÜBNER, Alexander; SCHÄFER, Fabian; SCHAAL, Kai N. Maximizing Profit via Assortment and Shelf-Space Optimization for Two-Dimensional Shelves. *Production and Operations Management*. 2020, vol. 29, no. 3, pp. 547–570. ISSN 1059-1478, ISSN 1937-5956. Available from DOI: `10.1111/poms.13111`.
14. HAJAD, Makbul; TANGWARODOMNUKUN, Viboon; JATURANONDA, Chorkaew; DUMKUM, Chaiya. Laser Cutting Path Optimization Using Simulated Annealing with an Adaptive Large Neighborhood Search. *The International Journal of Advanced Manufacturing Technology*. 2019, vol. 103, no. 1-4, pp. 781–792. ISSN 0268-3768, ISSN 1433-3015. Available from DOI: `10.1007/s00170-019-03569-6`.
15. VIJAY ANAND, K.; RAMESH BABU, A. Heuristic and Genetic Approach for Nesting of Two-Dimensional Rectangular Shaped Parts with Common Cutting Edge Concept for Laser Cutting and Profile Blanking Processes. *Computers & Industrial Engineering*. 2015, vol. 80, pp. 111–124. ISSN 03608352. Available from DOI: `10.1016/j.cie.2014.11.018`.
16. KANDASAMY, Vijay Anand; UDHAYAKUMAR, S. Effective Location of Micro Joints and Generation of Tool Path Using Heuristic and Genetic Approach for Cutting Sheet Metal Parts. *International Journal of Material Forming*. 2020, vol. 13, no. 2, pp. 317–329. ISSN 1960-6206, ISSN 1960-6214. Available from DOI: `10.1007/s12289-019-01488-1`.
17. HOLLAND, John H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor: University of Michigan Press, 1975. ISBN 978-0-472-08460-9.
18. DARWIN, Charles. *The Origin of Species: By Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. 6th ed. Cambridge University Press, 2009. ISBN 978-1-108-00548-7 978-0-511-69429-5. Available from DOI: `10.1017/CBO9780511694295`.
19. RIPON, Kazi Shah Nawaz; GLETTE, Kyrre; KHAN, Kashif Nizam; HOVIN, Mats; TORRESEN, Jim. Adaptive Variable Neighborhood Search for Solving Multi-Objective Facility Layout Problems with Unequal Area Facilities. *Swarm and Evolutionary Computation*. 2013, vol. 8, pp. 1–12. ISSN 22106502. Available from DOI: `10.1016/j.swevo.2012.07.003`.
20. BEAN, James C. Genetic Algorithms and Random Keys for Sequencing and Optimization. *ORSA Journal on Computing*. 1994, vol. 6, no. 2, pp. 154–160. ISSN 0899-1499, ISSN 2326-3245. Available from DOI: `10.1287/ijoc.6.2.154`.
21. LIU, Xun-bo; SUN, Xiao-ming. A Multi-Improved Genetic Algorithm for Facility Layout Optimisation Based on Slicing Tree. *International Journal of Production Research*. 2012, vol. 50, no. 18, pp. 5173–5180. ISSN 0020-7543, ISSN 1366-588X. Available from DOI: `10.1080/00207543.2011.654011`.

22. OTTEN, R.H.J.M. Automatic Floorplan Design. In: *19th Design Automation Conference*. Las Vegas, NV, USA: IEEE, 1982, pp. 261–267. ISBN 978-0-89791-020-0. Available from DOI: `10.1109/DAC.1982.1585510`.

23. LAI, M.; WONG, D.F. Slicing Tree Is a Complete Floorplan Representation. In: *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*. Munich, Germany: IEEE Comput. Soc, 2001, pp. 228–232. ISBN 978-0-7695-0993-8. Available from DOI: `10.1109/DATE.2001.915030`.

24. SYWERDA, Gilbert. Uniform Crossover in Genetic Algorithms. In: *Proceedings of the Third International Conference on Genetic Algorithms*. George Mason University, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 2–9. ISBN 1558600063.

25. BLUM, Christian; ROLI, Andrea. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*. 2003, vol. 35, no. 3, pp. 268–308. ISSN 0360-0300, ISSN 1557-7341. Available from DOI: `10.1145/937503.937505`.

26. HANSHENG, Lin; LISHAN, Kang. Balance between Exploration and Exploitation in Genetic Search. *Wuhan University Journal of Natural Sciences*. 1999, vol. 4, no. 1, pp. 28–32. ISSN 1007-1202, ISSN 1993-4998. Available from DOI: `10.1007/BF02827615`.

27. SCHOLZ, Daniel; JAEHN, Florian; JUNKER, Andreas. Extensions to STaTS for Practical Applications of the Facility Layout Problem. *European Journal of Operational Research*. 2010, vol. 204, no. 3, pp. 463–472. ISSN 03772217. Available from DOI: `10.1016/j.ejor.2009.11.012`.

28. BORTFELDT, Andreas; WINTER, Tobias. A Genetic Algorithm for the Two-Dimensional Knapsack Problem with Rectangular Pieces. *International Transactions in Operational Research*. 2009, vol. 16, no. 6, pp. 685–713. ISSN 09696016, ISSN 14753995. Available from DOI: `10.1111/j.1475-3995.2009.00701.x`.

29. CHAZELLE. The Bottomn-Left Bin-Packing Heuristic: An Efficient Implementation. *IEEE Transactions on Computers*. 1983, vol. C-32, no. 8, pp. 697–707. ISSN 0018-9340. Available from DOI: `10.1109/TC.1983.1676307`.

Appendix A

Appendix

```
$ curl --location 'localhost:9000/compute' \
--header 'Content-Type: application/json' \
--data '{
    "instanceParameters": {
        "layout": {
            "width": 30,
            "height": 20,
            "evalFunc": "f(x,y) = x+y"
        },
        "paintings": [
            { "ident": "1", "width": 5, "height": 7 },
            { "ident": "2", "width": 5, "height": 7 },
            { "ident": "3", "width": 5, "height": 7 },
            { "ident": "4", "width": 5, "height": 7 },
            { "ident": "5", "width": 5, "height": 7 },
            { "ident": "6", "width": 5, "height": 7 }
        ],
        "paintingsFlow": [
            { "from": 1, "to": 2, "flow": 3.3 },
            { "from": 1, "to": 3, "flow": 4.4 },
            { "from": 1, "to": 5, "flow": 0 }
        ]
    },
    "gaParameters": {
        "maxNumberOfIter": 300,
        "populationSize": 300,
        "maximumWildCardCount": 1,
        "orientationWeights": [ 1, 1, 0.5 ],
        "geneticAlgorithm": "simpleGa",
        "mate": "normalizedProbabilityVectorSum",
        "mutate": "flipOnePartAtRandom",
        "select": "tournament",
        "objective": "simple",
        "evaluator": "ga",
        "placingHeuristics": "corner",
        "populationDivisionCounts": {
            "elite": 0.2, "average": 0.6, "worst": 0.2,
            "children": 0.3, "mutant": 0.2, "winner": 0.2,
            "random": 0.1
        },
        "initialPopulationDivisionCounts": {
            "random": 0.7, "greedy": 0.3
        }
    },
    "objectiveParameters": {
        "name": "simple",
        "params": {
            "overlappingPenalizationConstant": 36.05,
            "outsideOfAllocatedAreaPenalizationConstant": 36.05
        }
    }
}'
```

■ **Code listing 5** Example of computation submission using `curl`[1] without specifying the instance name. Without it, everything has to be entered manually into the request – layout width and height, paintings together with their flow, and evaluation function.

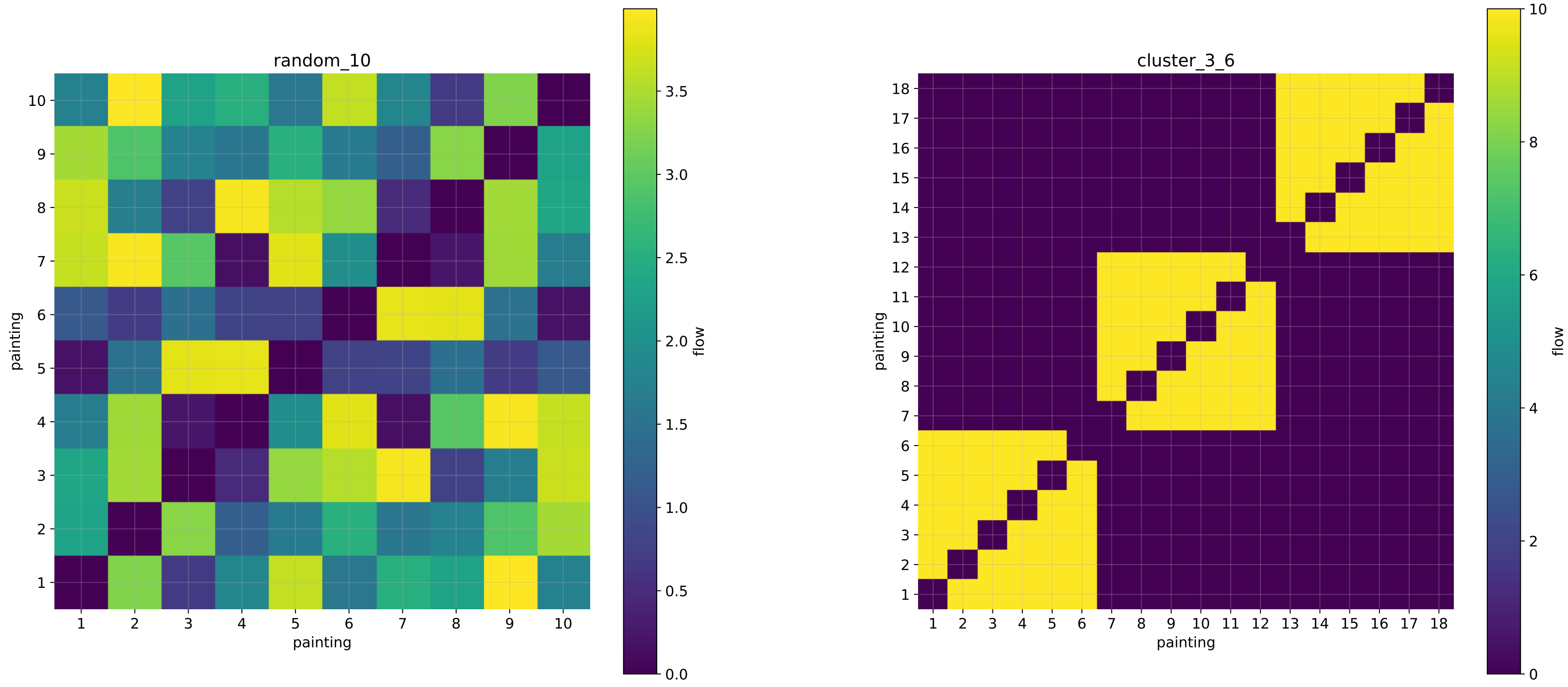


Figure A.1 Visualization of the flow for two painting placement instances. Flow expresses the affinity of paintings to each other. Paintings that should be placed close together have flow higher compared to paintings that should not. Flow matrices are symmetric,i.e., the flow between two paintings is the same from each direction. Also, the flow to/from itself is zero.

Contents of the attached medium

```
README.md ............................. the file with attached medium contents description
notebooks ...... the directory of Jupyter notebooks for visualization and dataset generation
impl ....................................... the directory with implementation source code
  build.sbt ................................... the build file for the computation server
  public/datasets .............................................. the directory of datasets
thesis ........................................................... the directory with thesis
  ctufit-thesis-en.tex ................................ the thesis source in TEX format
  ctufit-thesis-en.pdf .................................. the thesis text in PDF format
```