Optimization Algorithms

Thomas Weise

December 30, 2023

Abstract

Optimization is the art of finding good solutions for problems that are hard to solve. From Logistics, the scheduling of work in a factory or the lectures of students, the packing of objects into bins, over the design of rotor wings to the synthesis of dynamic controllers for engines – all involve, implicitly or explicitly, optimization. In this book we explore this very wide field. We look at it from the perspective of a programmer. We try to find define a pattern common to most optimization problems and then we explore how algorithms can be designed on basis of this pattern. We step-by-step investigate more and more complicated optimization approaches and learn lessons along the way. This book follows a very practical approach. We focus less on theory and more on learning-by-doing and trial-and-error . . . which, coincidentally, is very similar to how many of the algorithms that we will study work as well.

Contents

Preface

After writing "*Global Optimization Algorithms – Theory and Application*" [73] during my time as PhD student a long time ago, I now want to write a more *practical* guide to optimization and metaheuristics. Currently, this book is in an early stage of development and work-in-progress, so expect many changes.

This book tries to introduce optimization in an accessible way for an audience of undergraduate and graduate students without background in the field. It tries to provide an intuition about how optimization algorithms work in practice, how to recognize optimization problems and the basic structure behind them, what things to look for when solving an optimization problem, and how to get from a simple, working, "proof-of-concept" approach to an efficient algorithm for a given problem.

We follow a "learning-by-doing" approach by trying to solve one practical optimization problem as example theme throughout the book. All algorithms are directly implemented and applied to that problem after we introduce them. This allows us to discuss their strengths and weaknesses based on actual results. We learn how to compare the performance of different algorithms. We try to improve the algorithms step-by-step, moving from very simple approaches, which do not work well, to efficient metaheuristics.

We use concrete examples and algorithm implementations written in Python. The source code is freely available at `https://thomasweise.github.io/moptipy` under the GNU GENERAL PUBLIC LICENSE Version 3, 29 June 2007. The code listings in the book will usually be abridged excerpts. This means that we will omit a lot of details that are unnecessary for understanding the algorithms in play, such as type hints, sanity checks, or even complete methods. The full versions of the code can be reached by clicking on the "(src)"-links at the end of the listings captions away. The full versions therefore can look different from the illustrated abridged code in the book.

In order to fully understand the code examples, we recommend the reader to familiarize themselves with Python, numpy, and matplotlib. Of course, if you just read this book to learn about algorithms, you can ignore the source code examples.

The text of the book itself is actively written and available in the repository `https://github.com/thomasWeise/oa`. There, you can also submit issues, such as change requests, suggestions, errors, typos, or you can inform me that something is unclear, so that I can improve the book.

This book is released under the *Attribution-NonCommercial-ShareAlike 4.0 International license* (CC BY-NC-SA 4.0). You can freely share it. Please do not print it, though, to preserve the environment.

The results of the experiments that we run in this book are available in the repository `https://github.com/thomasWeise/oa_data`.

Chapter 1

Introduction

Today, algorithms influence a bigger and bigger part of both of our daily private and work life. They suggest interesting movies for us to watch or products to purchase. They help us find efficient routes when driving by car or match us to the next available and sufficiently nearby taxi. They control advertisement campaigns and suggest product pricing policies [53]. They support us by suggesting good decisions in a variety of fields, ranging from engineering, timetabling and scheduling, product design, to logistic planning. They will be the most important element of the transition of our industry to smarter manufacturing and intelligent production, where they can automate a variety of tasks, as illustrated in Figure 1.1.

Optimization and Operations Research provide us with algorithms that propose good solutions to a very wide range of questions. These solutions achieve a predefined goal while minimizing (at least) one resource requirement, be it costs, energy consumption, space, the time requirement, and so on. Besides saving direct costs, the reduction of resource consumption often is also good for the environment. Therefore, optimization can help us to become more efficient both economically and ecologically.

We can thus already list three obvious reasons why optimization will be a key technology for the next century:

1. The automation of production can improve the work life by reducing manual work while increasing productivity and product quality. However, any form of intelligent production or smart manufacturing needs automated decisions. Since these decisions should be *intelligent,* they can only come from a process which involves optimization in one way or another.

2. All branches of industry, all service sectors, as well as cities and regions face both global and

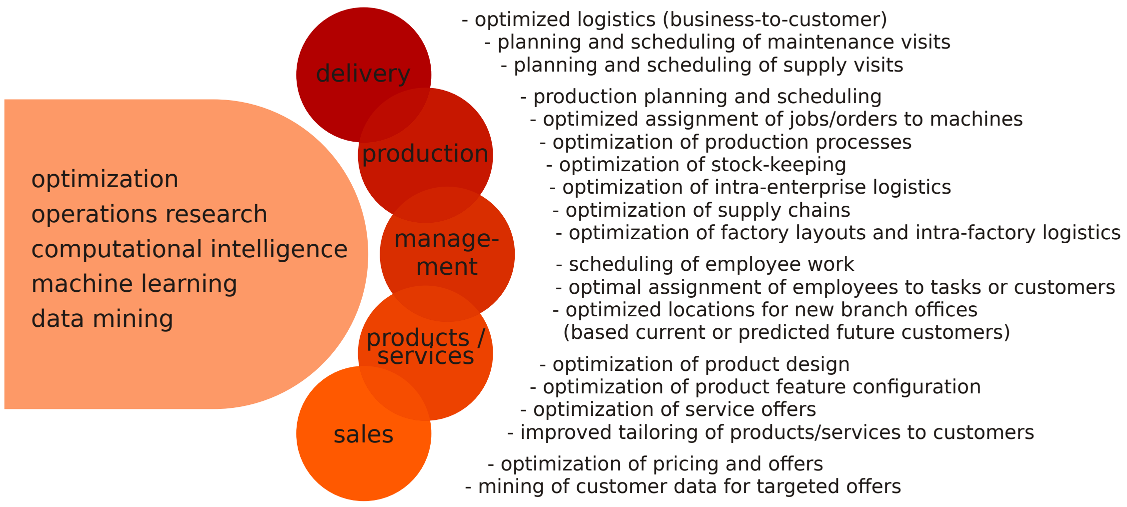

optimization
operations research
computational intelligence
machine learning
data mining
delivery
production
manage-ment
products / services
sales
- optimized logistics (business-to-customer)
- planning and scheduling of maintenance visits
- planning and scheduling of supply visits
- production planning and scheduling
- optimized assignment of jobs/orders to machines
- optimization of production processes
- optimization of stock-keeping
- optimization of intra-enterprise logistics
- optimization of supply chains
- optimization of factory layouts and intra-factory logistics
- scheduling of employee work
- optimal assignment of employees to tasks or customers
- optimized locations for new branch offices (based current or predicted future customers)
- optimization of product design
- optimization of product feature configuration
- optimization of service offers
- improved tailoring of products/services to customers
- optimization of pricing and offers
- mining of customer data for targeted offers

Figure 1.1: Examples for applications of optimization, computational intelligence, machine learning techniques in five fields of smart manufacturing: the production itself, the delivery of the products, the management of the production, the products and services, and the sales level.

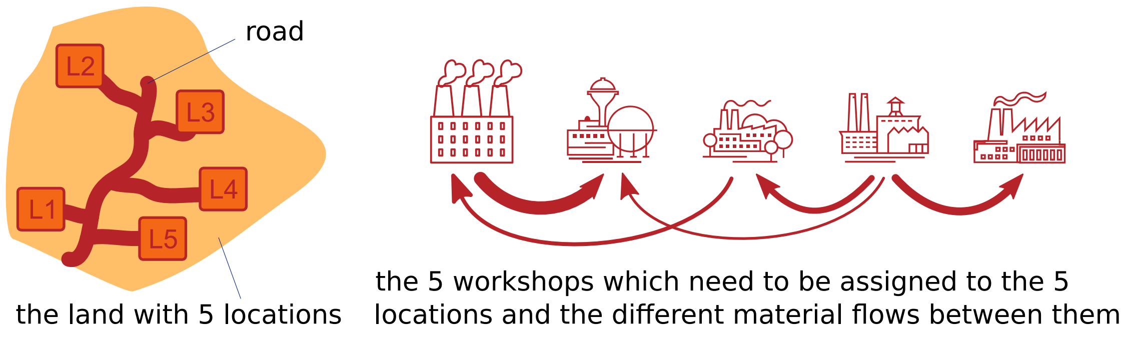

road
L1
L2
L3
L4
L5
the land with 5 locations
the 5 workshops which need to be assigned to the 5 locations and the different material flows between them

Figure 1.2: Illustrative sketch of a quadratic assignment scenario, where different buildings of a factory need to be laid out on a plot of land.

local competition. Those who can reduce their resource consumption and costs while improving product, production, or service quality and efficiency will have the edge. One key technology for achieving this is better planning via optimization.

3. Our world suffers from both depleting resources and too much pollution. Optimization can "give us more while needing less." This often leads to more environmentally friendly processes.

But how can algorithms help us to find solutions for hard problems in a variety of different fields? How general are these algorithms? How can they help us to make good decisions? How can they help us to save resources?

In this book, we will try to answer all of these questions. We will explore quite a lot of different optimization algorithms. We will look at their actual implementations and we will apply them to example problems to see what their strengths and weaknesses are.

1.1 Examples

Let us first look at some typical use cases of optimization.

1.1.1 Example: Layout of Factories

There are both dynamic and static aspects of intelligent production, as well as all sorts of nuances in between. The question of where to put which facility in a factory is a rather static, but quite important aspect. Let us assume we own a company and bought a plot of land to construct a new factory. Of course we know which products we will produce in this factory. We also know which facilities we need to construct, i.e., the workshops, storage depots, and maybe an administrative building. What we need to decide is where to place them on our land, as illustrated in Figure 1.2.

Let us assume we have n locations on our plot of land that we can use for our n facilities. In some locations, there might be already buildings, in others, we may need to construct them anew. For each facility and location pair, a different construction cost may arise. A location with an existing shed might be a good solution to put a warehouse. However, if we want to put the administration building there, we would first need to demolish the shed. These are the static costs.

But placing the facilities also has a very big impact on the running costs.

For every possible plan, costs also arise from the relative distances between the facilities that we wish to place. Maybe there is a lot of material flow between two of the workshops. Finished products and raw material may need to be transported between a workshop and the storage depot. Between the administration building and the workshops, on the other hand, there will usually be no material flow. Of course, the distance between two facilities will depend on the locations we pick for them. For each pair of facilities that we place on the map, flow costs will arise as a function of the amount of material to be transported between them and the distance of their locations.

The total cost of an assignment of facilities to locations is therefore the sum of the resulting base costs and flow costs. Our goal would be to find the assignment (i.e., the plan) with the smallest possible total cost.

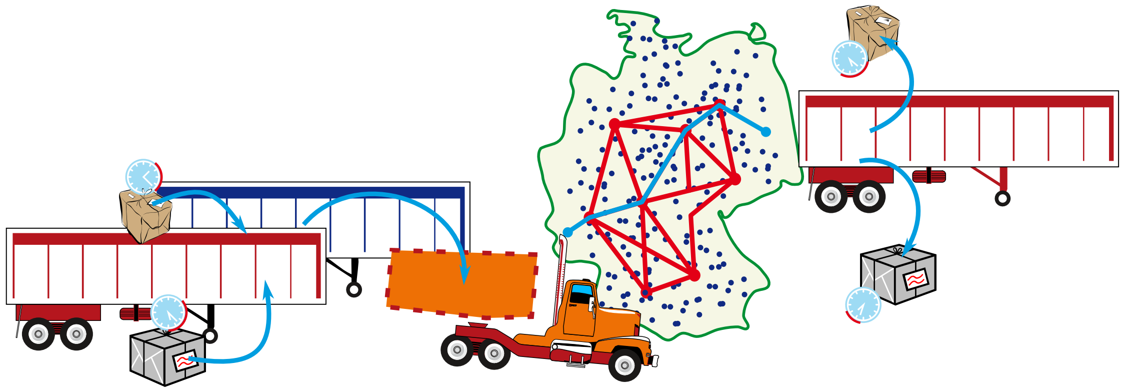


Figure 1.3: Illustrative sketch of logistics problems: Orders require us to pick up some items at source locations within certain time windows and deliver them to their destination locations, again within certain time windows. We need to decide which containers and vehicles to use and over which routes we should channel the vehicles.

This scenario is called Quadratic Assignment Problem (QAP) [14]. It has been subject to research since the 1950s [8]. QAPs appear in wide variety of scenarios such as the location of facilities on a plot of land or the placement of work stations on the factory floor. Even if we need to place components on a circuit board in a way that minimizes the total wire length, we basically have a QAP, too [64]! Despite being relatively simple to understand, the QAP is hard to solve [60].

1.1.2 Example: Route Planning for a Logistics Company

Another, more dynamic application area for optimization is logistics. Let us look at a typical real-world scenario from this field [75, 76]: the situation of a logistics company that fulfills delivery tasks for its clients. A client can order one or multiple containers to be delivered to her location within a certain time window. She will fill the containers with goods, which are then to be transported to a destination location, again within a certain time window. The logistics company may receive many such customer orders per day, maybe several hundreds or even thousands. The company may have multiple depots, where containers and trucks are stored. For each order, it needs to decide which container(s) to use and how to get them to the customer, as sketched in Figure 1.3. The trucks it owns may have different capacities and could, e.g., carry either one or two containers. Besides using trucks, which can travel freely on the map, it may also be possible to utilize trains. Trains have higher capacities and can carry many containers. Different from trucks, they must follow specific schedules. They arrive and depart at fixed times to/from fixed locations. For each possible vehicle, different costs could occur. Containers can be exchanged between different vehicles at locations such as parking lots, depots, or train stations.

The company could have the goals to fulfill all transportation requests *at the lowest cost*. Actually, it might seek to maximize its profit, which could even mean to outsource some tasks to other companies. The goal of optimization then would be to find the assignment of containers to delivery orders and vehicles and of vehicles to routes, which maximizes the profit. And it should do so within a limited, feasible time.

Of course, there is a wide variety of possible logistics planning tasks. Besides our real-world example above, a classical task is the Traveling Salesperson Problem (TSP) [5, 33, 48], where the goal is to find the shortest round-trip tour through n cities, as sketched in Figure 1.4. Many other scenarios can be modeled as such logistics questions, too: If a robot arm needs to several drill holes into a circuit board, finding the shortest tour means solving a TSP and will speed up the production process, for instance [32].

1.1.3 Example: Packing, Cutting Stock, and Knapsack

Let's say that your family is moving to a new home in another city. This means that you need to transport all of your belongings from your old to your new place, your PC, your clothes, maybe some furniture, a washing machine, and a refrigerator, as sketched in Figure 1.5. You cannot pack everything into your car at once, so you will have to drive back and forth a couple of times. But how often will you have to drive? Packing problems [24, 61] aim to package sets of objects into containers as efficient

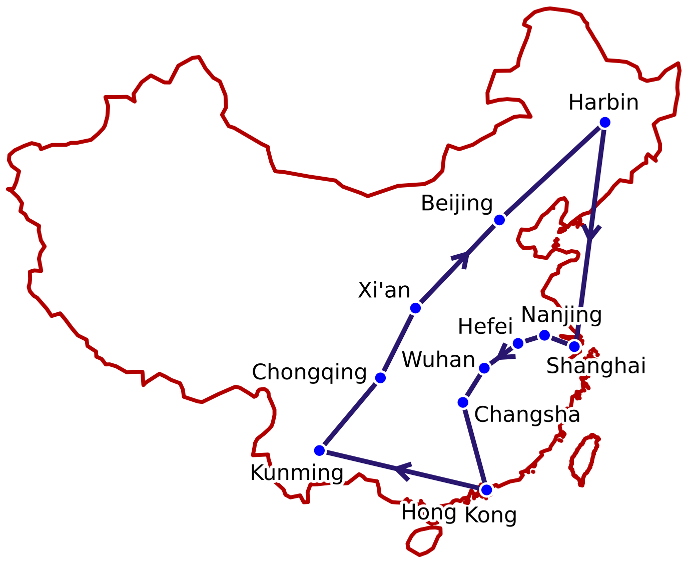


Figure 1.4: A Traveling Salesperson Problem (TSP) through eleven cities in China.

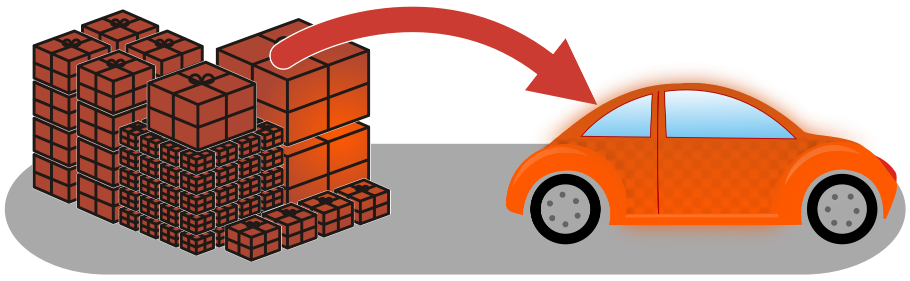


Figure 1.5: A sketch illustrating a packing problem.

as possible, i.e., in such a way that we need as few containers as possible. Your car can be thought of as a container and whenever it is filled, you drive to the new flat. If you need to fill the container four times, then you have to drive back and forth four times.

Such Bin Packing Problems (BPPs) exist in many variants and are very related to cutting stock problems [24]. They can be one-dimensional [22], for example if we want to transport dense/heavy objects with a truck where the maximum load weight is limiting factor while there is enough space capacity. This is similar to having a company which puts network cables into people's homes and therefore bulk purchases reels with 100m of cables each. Of course, each home needs a different required total length of cables and we want to cut our cables such that we need as few reels as possible. TODO A two-dimensional variant [50] could correspond to printing a set of (rectangular) images of different sizes on (rectangular) paper. Assume that more than one image fits on a sheet of paper but we have too many images for one piece of paper. We can cut the paper after printing to separate the single images. We then would like to arrange the images such that we need as few sheets of paper as possible.

The three-dimensional variant then corresponds to our moving homes scenario. Of course, there are many more different variants — the objects we want to pack could be circular, rectangular, or have an arbitrary shape. We may also have a limited number of containers and thus may not be able to pack all objects, in which case we would like to only package those that give us the most profit (arriving at a task called knapsack problem [51]).

1.1.4 Example: Job Shop Scheduling Problem

Another typical optimization task arises in manufacturing, namely the assignment ("scheduling") of tasks ("jobs") to machines in order to optimize a given performance criterion ("objective"). Scheduling [57, 58] is one of the most active areas of operational research for more than six decades.

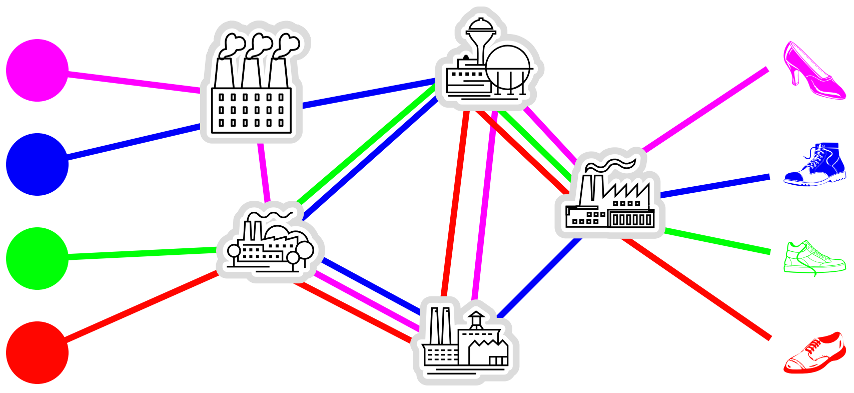


Figure 1.6: Illustrative sketch of a Job Shop Scheduling Problem (JSSP) scenario with four jobs where four different types of shoe should be produced, which require different workshops ("machines") to perform different production steps.

In the Job Shop Scheduling Problem (JSSP) [11, 17, 31, 46, 47, 66], we have a factory ("shop") with several machines. We receive a set of customer orders for products which we have to produce. We know the exact sequence in which each product/order needs to pass through the machines and how long it will need at each machine. Each production job has one production step ("operation") for each machine on which it needs to be processed. These operations must be performed in the right sequence. Of course, no machine can process more than one operation at a time. While we must obey these constraints, we can decide about the time at which each of the operations should begin. Often, we are looking for the starting times that lead to the earliest completion of all jobs, i.e., the shortest makespan.

Such a scenario is sketched in Figure 1.6, where four orders for different types of shoe should be produced. The resulting jobs pass through different workshops (or machines, if you want) in different order. Some, like the green sneakers, only need to be processed by a subset of the workshops.

This general scenario encompasses many simpler problems. For example, if we only produce one single product, then all jobs would pass through the same machines in the same order. Customers may be able to order different quantities of the product, so the operations of the different jobs for the same machine may need different amounts of time. This is the so-called Flow Shop Scheduling Problem (FSSP) — and it has been defined back in 1954 [40]!

Clearly, since the JSSP allows for an *arbitrary* machine order per job, being able to solve the JSSP would also enable us to solve the FSSP, where the machine order is fixed. We will introduce the JSSP in detail in TODO and use it as the main example in this book on which we will step-by-step exercise different optimization methods.

1.1.5 Summary

The examples we have discussed so far are, actually, related to each other. They all fit into the broad areas of operational research and smart manufacturing [21, 36]. The goal of smart manufacturing is to optimize development, production, and logistics in the industry. Therefore, computer control is applied to achieve high levels of adaptability in the multi-phase process of creating a product from raw material. The manufacturing processes and maybe even whole supply chains are networked. Product lot sizes are small and a high flexibility is required to adapt production processes to customer wishes. This creates the requirement for a large degree of automation and automatic intelligent decisions. The key technology necessary to propose such decisions are optimization algorithms. In a perfect world, the whole production process as well as the warehousing, packaging, and logistics of final and intermediate products would take place in an *optimized* manner. No time or resources would be wasted as production gets cleaner, faster, and cheaper while the quality increases.

1.2 Metaheuristics: Why do we need them?

The main topic of this book will be metaheuristic optimization (although I will eventually also discuss some other methods (remember: work in progress). So why do we need metaheuristic algorithms?

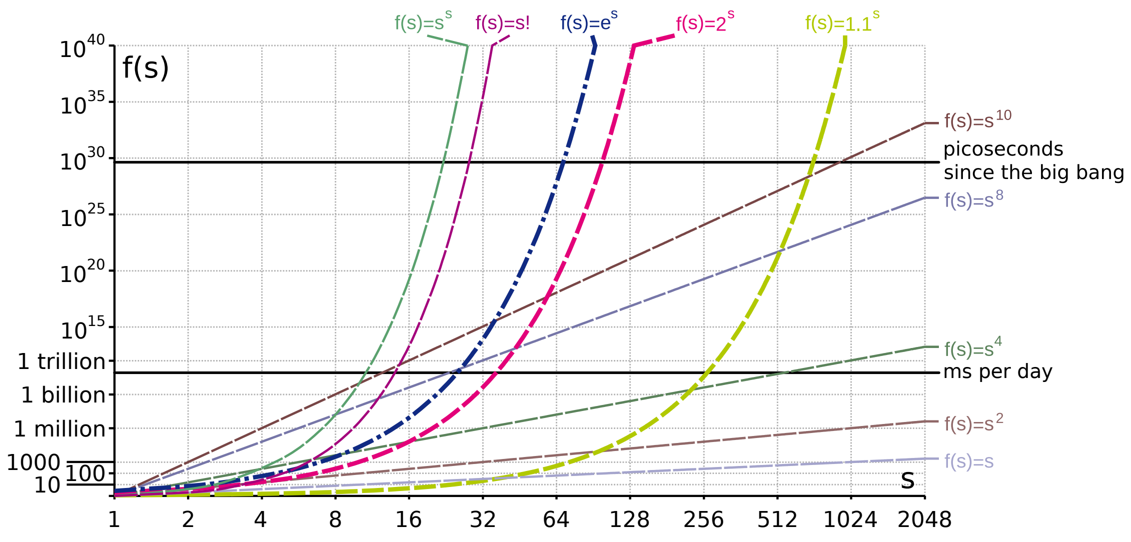


Figure 1.7: The growth of different functions in a log-log scaled plot. Exponential functions grow very fast. This means that an algorithm which needs $\sim 2^s$ steps to solve an optimization problem of size s quickly becomes infeasible if s grows.

Why should you read this book?

1.2.1 Good Solutions within Acceptable Runtime

The first and foremost reason is that they can provide us good solutions within reasonable time. It is easy to understand that there are some problems which are harder to solve than others. Every one of us already knows this from the mathematics classes in school. Of course, the example problems discussed before cannot be attacked as easily as solving a single linear equation. They require algorithms, they require computer science.

Ever since primary school, we have learned many problems and types of equations that we can solve. Unfortunately, theoretical computer science shows that for many problems, the time we need to find the best-possible solution can grow *exponentially* with the number of involved variables in the worst case. The number of involved variables here could be the number of cities in a TSP, the number of jobs or machines in a JSSP, or the number of objects to pack in a, well, packing problem. A big group of such complicated problems are called $\mathcal{NP}$-hard [15, 47]. Unless some unlikely breakthrough happens in terms of which problems can be solved efficiently and which not [18, 41], there will be many problems that we cannot always solve exactly within reasonable time. Each and every one of the example problems discussed belongs to this type!

As sketched in Figure 1.7, the exponential function rises very quickly. One idea would be to buy more computers for bigger problems and to simply parallelize the computation. Well, parallelization can provide a linear speed-up at best. If we have two CPU cores, we can solve the problem in half of the time and if we have three CPU cores, we can do it in one third of the time, and so on ... if the problem lends itself to parallelization (and not all of them do). Either way, we are dealing with problems where the runtime requirements may double every time we add a single new decision variable. And no: Quantum computers are not the answer. Most likely, they cannot even solve these problems qualitatively faster either [1].

So what can we do to solve such problems? The exponential time requirement occurs if we make *guarantees* about the solution quality, especially about its optimality, over all possible scenarios. What we can do, therefore, is that we can trade-in the *guarantee* of finding the best possible solution for lower runtime requirements. We can use algorithms from which we *hope* that they find a good *approximation* of the optimum, i.e., a solution which is very good with respect to the objective function, but which do not *guarantee* that this result will be the best possible solution. We may sometimes be lucky and even find the optimum, while in other cases, we may get a solution which is close enough. And we will get this within acceptable time limits.

In Figure 1.8 we illustrate this idea on the example of the Traveling Salesperson Problem [5, 33, 48] briefly mentioned in Section 1.1.2. The goal of solving the TSP is to find the shortest round trip tour

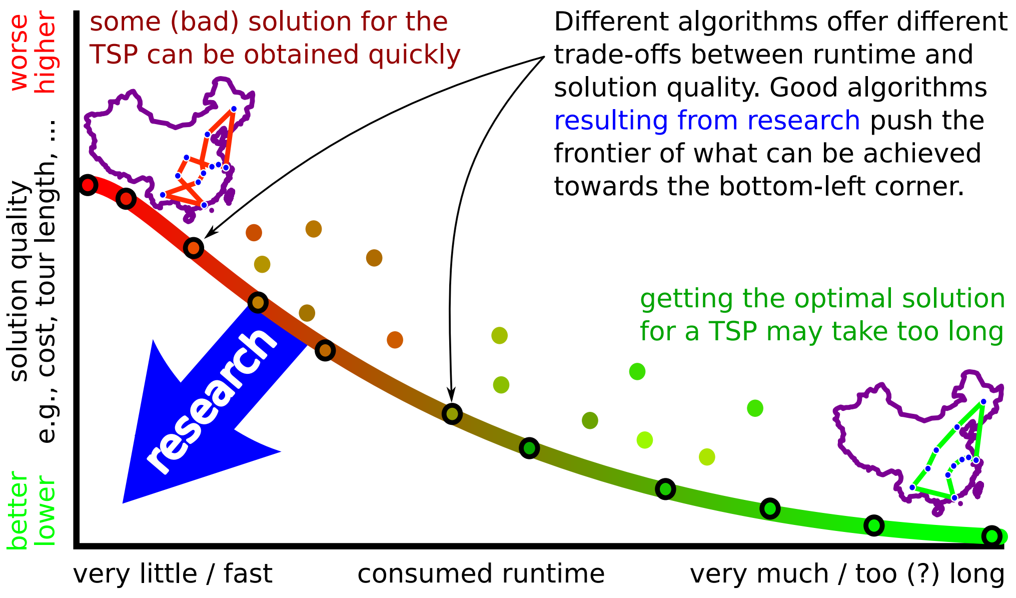


Figure 1.8: The trade-off between solution quality and runtime.

through n cities. The TSP is $\mathcal{NP}$-hard [26, 33]. Today, it is possible to solve many large instances of this problem to optimality by using sophisticated *exact* algorithms [19, 20]. Yet, finding the *shortest possible tour* for a particular TSP might still take many years if you are unlucky. Finding just *one tour* is, however, very very easy: I can write down the cities in any particular order. Of course, I can visit the cities in an arbitrary order. That is an entirely valid solution, and I can obtain it basically in 0 time. This "tour" would probably be very bad, very long, and generally not a good idea.

In the real world, we need something in between. We need a solution which is as good as possible as fast as possible. Heuristic and metaheuristic algorithms offer different trade-offs of solution quality and runtime. Different from exact algorithms, they do not guarantee to find the optimal solution and often make no guarantee about the solution quality at all. Still, they often allow us to get very good solutions for computationally hard problems in short time. They may often still discover them (just not always, not guaranteed).

1.2.2 Good Solutions within Acceptable Development Time

Saying that we need a good algorithm to solve a given problem is very easy. Developing a good algorithm to solve a given problem is not, as any graduate student in the field can probably confirm. Before, I stated that great exact algorithms for the TSP exist [19, 20], that can solve many TSPs quickly (although not all). There are years and years of research in these algorithms. Even the top heuristic and metaheuristic algorithm for the TSP today result from many years of targeted research [34, 55, 79] and their implementation from the algorithm specification alone can take months [77]. Unfortunately, if you do not have plain TSP, but one with some additional constraints — say, time windows to visit certain cities — the optimized, state-of-the-art TSP solvers are no longer applicable. And in a real-world application scenario, you do not have years to develop an algorithm. What you need are simple, versatile, general algorithm concepts that you can easily adapt to your problem at hand. Something that can be turned into a working prototype within a few weeks.

Metaheuristics are the answer [54, 73]. They are general algorithm concepts into which we can plug problem-specific modules. General metaheuristics are usually fairly easy to implement and deliver acceptable results. Once a sufficiently well-performing prototype has been obtained, we could go and integrate it into the software ecosystem of the customer. We also can try to improve its performance using different ideas ... and years and years of blissful research, if we are lucky enough to find someone paying for it.

Part I

The Structure of Optimization Problems

Chapter 2

Introduction

From the examples that we have seen, we know that optimization problems come in all kinds of different shapes and forms. Without practice, it is not directly clear how to identify, define, understand, or solve them. Moreover, it is not really clear how we can solve such a wide range of problems using the same kind of methods.

The goal of this part of the book is to bring some order into this mess. We will approach an optimization task step-by-step by formalizing its components, which will then allow us to apply efficient algorithms to it. This *structure of optimization* is a blueprint that can be used in many different scenarios as basis to apply different optimization algorithms.

We will approach this domain from the perspective of a programmer. Imagine you are a programmer and your job would be to, well, make a program that solves optimization problems. Now, an "optimization problem" seems to be a very general and amorphous thing. The first thing you would try to do is to discover some components that commonly occur in all of the optimization problems you can think of. If you can manage to specify how such components look like, what information and functionality they provide, then you are a step closer to fulfilling your task. Then, you can develop algorithms that use these information and functionality to find solutions. This is what we will do.

First, let us clarify what *optimization problems* actually are.

Definition 2.1 (Optimization Problem I) An *optimization problem* is a situation $\mathcal{I}$ which requires deciding for one choice from a set of possible alternatives in order to reach a predefined/required goal at minimal costs.

Definition 2.1 presents an economical point of view on optimization in a rather informal manner. But the points come across: We want to reach a certain goal, e.g., visit all cities in a TSP, process all jobs in a JSSP, pack all the items into bins in a BPP, or assign all factories to locations in the QAP. We have several possible ways to reach that goal and we need to choose one among them. Each of these possible choices has an associated cost. Since all of them lead to reaching the goal, we want to pick the one choice with the minimal costs. We can refine this situation into the more mathematical formulation given in Definition 2.2.

Definition 2.2 (Optimization Problem II) The goal of solving an *optimization problem* is finding an input value $\star y \in \mathbb{Y}$ from a set $\mathbb{Y}$ of allowed values for which a function $f : \mathbb{Y} \mapsto \mathbb{R}$ takes on the smallest value.

From these definitions, we can already deduce a set of necessary components that make up such an optimization problem. We will look at them from the perspective of a programmer:

1. The first obvious component is a data structure $\mathbb{Y}$ representing possible solutions to the problem. This one half of the output of the optimization software and is discussed in TODO.

2. Then, there is the so-called objective function $f : \mathbb{Y} \mapsto \mathbb{R}$, which rates the quality of the candidate solutions $y \in \mathbb{Y}$. It basically returns the cost of a solution and we usually want to minimize it.

3. The third component follows from the definitions rather implicitly: The problem instance data $\mathcal{I}$, i.e., the concrete situation which defines the framework conditions for the solutions to be found. This input data of the optimization algorithm is discussed in TODO.

If we want to solve a TSP as sketched in Section 1.1.2), for instance, then the instance data $\mathcal{I}$ could include the locations of the cities that we want to visit from which we then can compute the travel distances. The candidate solution data structure $\mathbb{Y}$ could simply be a "city list" containing each city exactly once and prescribing the visiting order. The objective function f would take such a city list $y \in \mathbb{Y}$ as input and compute the overall tour length. It would be subject to minimization.

Often, in order to actually practically implement an optimization approach, there will also be

5. a search space $\mathbb{X}$, i.e., a simpler data structure for internal use, which can more be efficiently processed by an optimization algorithm than $\mathbb{Y}$ (see TODO),
6. a mapping $\gamma : \mathbb{X} \mapsto \mathbb{Y}$, which decodes the "points" $x \in \mathbb{X}$ from the search space $\mathbb{X}$ to candidate solutions $y \in \mathbb{Y}$ in the solution space $\mathbb{Y}$ (see TODO),
7. search operators $\mathrm{move} : \mathbb{X}^n \mapsto \mathbb{X}$, which allow for the iterative exploration of the search space $\mathbb{X}$ (see TODO), and
8. a termination criterion, which tells the optimization process when to stop (see TODO).

At first glance, all of this looks a bit complicated — but rest assured, it won't be. We will explore all of these structural elements that make up an optimization problem in this chapter, based on a concrete example of the Job Shop Scheduling Problem (JSSP) from Section 1.1.4 [11, 31, 46, 47, 66]. This example should give a reasonable idea about how the structural elements and formal definitions involved in optimization can be realized in practice. While any actual optimization problem can require very different data structures and operations from what we will discuss here, the general approach and ideas that we will discuss on specific examples should carry over to many scenarios.

At this point, I would like to make explicitly clear that the goal of this book is NOT to solve the JSSP particularly well. Our goal is to have an easy-to-understand yet practical introduction to optimization. This means that sometimes we will choose an easy-to-understand approach, algorithm, or data structure over a better but more complicated one. Also, our aim is to nurture the general ability to come up with a solution approach to a new optimization problem within a reasonably short time, i.e., without being able to conduct research over several years. That being said, the algorithms and approaches discussed in this book are not necessarily inefficient. While having much room for improvement, we eventually reach approaches that will find decent solutions.

Chapter 3

Problem Instance Data

3.1 Definitions

We distinguish optimization *problems* (see Definition 2.2) from *problem instances*. An optimization problem is the general blueprint of the tasks. The JSSP, for instance, has the goal of scheduling production jobs to machines under a set of constraints that we will discuss later. A problem instance of the JSSP is a concrete scenario, e.g., a concrete lists of tasks, requirements, and machines.

Definition 3.1 (Problem Instance $\mathcal{I}$) A concrete instantiation of all information that are relevant from the perspective of solving an optimization problems is called a *problem instance* $\mathcal{I}$.

3.2 A Programmer's Perspective

From the perspective of a programmer, the problem instance is the *input* of the optimization algorithms. A problem instance is related to an optimization problem in the same way an object/instance is related to its `class` in an object-oriented programming language like Python or Java, or a `struct` in C. The `class` defines which member variables exist and what their valid ranges are. An instance of the `class` is a piece of memory which holds concrete values for each member variable.

For the TSP, for example, a problem instance could be a matrix with the distances between the cities. For the QAP, we could have two matrices: One with the distances between the locations and one with the flows between the facilities. For the BPP, we would need to know the size of the containers into which we want to pack the objects and as well as the sizes of the objects that we want to pack. A programmer would then create a `class` with attributes to store this information.

Side note: A programmer would probably also define some sort of text file format to read and write such instance data. Because then, she would have already taken care of how to handle the input of her program: It will be files of this text format.

Rule of Thumb 1 One part of the *input* of optimization algorithm is the instance data $\mathcal{I}$.

3.3 Example: Job Shop Scheduling

3.3.1 JSSP Instance Structure

So how can we characterize a JSSP instance $\mathcal{I}$? In the basic and yet general scenario [31, 46, 47, 66], our factory has $m \in \mathbb{N}_0$ machines. At each point in time, a machine can either work on exactly one job or do nothing (be idle). A job may correspond to a customer order, e.g., "produce 10 red lady's sneakers." There are $n \in \mathbb{N}_1$ jobs that we need to schedule to these machines. For the sake of simplicity and for agreement between our notation here, the Python source code, and the example instances that we will use, we reference jobs and machines with 0-based indices from $0..(n-1)$ and $0..(m-1)$, respectively.

Each of the n jobs is composed of m "operations" — one for each machine. These operations correspond to the single production steps, such as "cut the cloth material," "stitch the cloth material to the sole," and so on. Each job may need to pass through the machines in a different order. The operation j of job i must be executed on machine $M_{i,j} \in 0..(m-1)$. Doing so needs $T_{i,j} \in \mathbb{N}_0$ time units for completion. Here, "time unit" is just a placeholder for a real measure of time and in a practical application, the time unit could be milliseconds, seconds, hours, days, or something else. Once a machine j begins to process an operation, it cannot stop until the operation is completed, i.e., will remain busy for $T_{i,j} \in \mathbb{N}_0$ time units.

This definition of a JSSP instance $\mathcal{I}$ is quite versatile. For example, assume that we have a factory that produces exactly one single product, but different customers may order different quantities of this product. Then, we would have JSSP instances where all jobs need to be processed by exactly the same machines in exactly the same sequence. In this case $M_{i_1,j} = M_{i_2,j}$ would hold for all jobs i_1 and i_2 and all operation indices j. The jobs would pass through all machines in the same order but may have different processing times (due to the different quantities).

We may also have scenarios where customers can order different types of products, say the same liquid soap, but either in bottles, plastic bags, or big canisters. Then, different machines may be needed for different orders. This is similar to the situation illustrated in Figure 1.6, where some job i does not need to be executed on a machine j' while another does. We then can simply set the required time $T_{i,j}$ to 0 for the operation j with $M_{i,j} = j'$. Notice that for this reason we wrote "$T_{i,j} \in \mathbb{N}_0$" above, because $\mathbb{N}_0$ includes 0.

In other words, the JSSP instance structure described here already encompasses a wide variety of real-world production situations. If we can build an algorithm which can solve this general type of JSSP well, it can automatically also solve the above-mentioned special cases.

3.3.2 JSSP Benchmark Instances

In order to practically play around with optimization algorithms for a certain problem, we need concrete instances of that problem. If we want to know whether an algorithm for the TSP works well, then we need some TSP instances, e.g., maps with the cities that we want to visit.

If we want to explore how different algorithms for the JSSP perform, then we need some concrete instances of the JSSP. Obviously, if we want to know whether an algorithm $\mathcal{A}$ is better than an algorithm $\mathcal{B}$, then we need to apply them to the same problem instance. Results obtained for different scenarios are inherently incomparable.

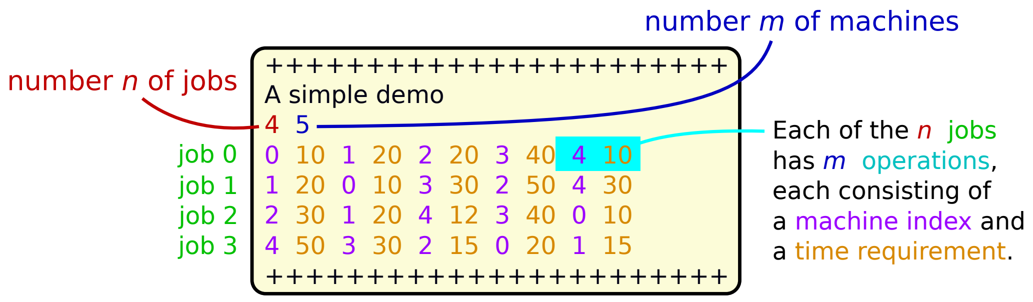
Luckily, the optimization community provides "benchmark instances" for many different optimization problems. Such common, well-known instances are important, because they allow researchers to compare their algorithms.

The eight classical and most commonly used sets of benchmark instances for the JSSP [35] are published in [3, 6, 23, 25, 49, 65, 66, 82]. Their data can be found (sometimes partially) in several repositories in the Internet, such as

- the OR-Library managed by John Edward Beasley [7],
- the comprehensive set of JSSP instances provided by Jelke Jeroen van Hoorn [70, 71], where also state-of-the-art results are listed,
- Oleg V. Shylo's Page [63], which, too, contains up-to-date experimental results,
- Éric D. Taillard's Page [67], or, finally,
- my own repository jsspInstancesAndResults [74], where I collect all the above problem instances and many results from existing works.

We will try to solve JSSP instances obtained from these collections. The goal of this book is that you can play around with the algorithms and replicate our experiments. Therefore, we cannot use all 242 instances from the above sets, because then the experiments would take too long. We have to pick a small representative subset of instances. We therefore first removed 67 instances that are relatively easy from the instance set. We then picked instances with different scales and from different sources. They will serve as illustrative example of how to approach optimization problems. In order to keep the example and analysis simple, we will focus on only eight instances, namely

1. Instance abz8 by Adams, Balas, and Zawack [3] has 20 jobs and 15 machines. The processing times of its operations were chosen from the interval 11..40.



```
+++++++++++++++++++++++++
A simple demo
4 5
0 10 1 20 2 20 3 40 4 10
1 20 0 10 3 30 2 50 4 30
2 30 1 20 4 12 3 40 0 10
4 50 3 30 2 15 0 20 1 15
+++++++++++++++++++++++++
```

number m of machines
number n of jobs
job 0
job 1
job 2
job 3
Each of the n jobs has m operations, each consisting of a machine index and a time requirement.

Figure 3.1: The meaning of the text representing our demo instance of the JSSP, as an example of the format used by the OR-Library.

2. Instance dmu67 by Demirkol, Mehta, and Uzsoy [23] has 40 jobs and 20 machines. Its processing times were chosen from the interval 1..200. This instance is structured such that the jobs first need to pass one (randomly chosen) half of the machines and then the other.

3. Instance dmu72 is from the same group and has the same structure as dmu67, but has 50 jobs and 15 machines.

4. Instance la38 by Lawrence [49] has 15 jobs and 15 machines. Its processing times are from 5..99.

5. Instance orb06 by Applegate and Cook [6] has 10 jobs and 10 machines. It was generated in 1986 as part of a set of "specially generated tougher problems" [35, 38]. Nevertheless, it will be the smallest instance in our experiments.

6. Instance swv14 by Storer, Wu, and Vaccari [65] has 50 jobs and 10 machines. Its processing times are from the interval 1..100. Like in the case of dmu72, the jobs first need to pass one (randomly chosen) half of the machines and then the other.

7. Instance ta70 by Taillard [66] has 50 jobs and 20 machines. Its processing times are from the interval 1..99.

8. Instance yn4 by Yamada and Nakano [82] has 20 jobs and 20 machines. Its processing times are from the interval 10..50.

The raw data of these instances is part of the moptipy Python package with the sources for our experiments as resource.

Of course, if we really want to solve a new type of problem, we will normally use as many benchmark problem instances as possible to get a good understand about the performance of our algorithm(s). Only for the sake of clarity of presentation, we will here limit ourselves to the above eight problems. We have chosen them as hopefully diverse representatives of all of the common JSSP benchmarks. They stem from instance sets contributed by different researchers and have different numbers of jobs and machines.

3.3.3 File Format and demo Instance

Our benchmark instances for the JSSP are taken from the OR-Library. We mentioned before that it often makes sense to have a simple text format to represent the input of optimization algorithms, the instance data $\mathcal{I}$. The OR-Library defines such a format for the JSSP. For the sake of simplicity, we created one additional, smaller JSSP instance to describe the format of these files, as illustrated in Figure 3.1.

Each problem instance $\mathcal{I}$ in the OR-Library starts and ends with a line of several + characters. The next line is a short description or title of the instance. In the third line, the number n of jobs is specified, followed by the number m of machines. The actual IDs or indexes of machines and jobs are 0-based, similar to array indices in Python. The JSSP instance definition is completed by n lines of text. Each such line specifies the operations of one job $i \in 0..(n-1)$. Each operation j is specified as a pair of two numbers, the ID $M_{i,j}$ of the machine that is to be used (violet), from the interval $0..(m-1)$, followed by the number of time units $T_{i,j}$ the job will take on that machine (orange). The order in which these operations appear in a line defines exactly the order in which the job needs to be passed through the machines. Of course, each machine can only process at most one job at a time.

In our demo instance illustrated in Figure 3.1, this means that we have $n = 4$ jobs and $m = 5$ machines. Job 0 first needs to be processed by machine 0 for 10 time units, it then goes to machine 1 for 20 time units, then to machine 2 for 20 time units, then to machine 3 for 40 time units, and finally to machine 4 for 10 time units. This job will thus take $10 + 20 + 20 + 40 + 10 = 100$ time units to be completed, *if* it can be scheduled without any delay or waiting period, i.e., if all of its operations can directly be processed by their corresponding machines. Job 3 first needs to be processed by machine 4 for 50 time units, then by machine 3 for 30 time units, then by machine 2 for 15 time units, then by machine 0 for 20 time units, and finally by machine 1 for 15 time units. It would not be allowed to first send Job 3 to any machine different from machine 4 and after being processed by machine 4, it must be processed by machine 3 — although it may be possible that it has to wait for some time, if machine 3 would already be busy processing another job. In the ideal case, job 3 could be completed after 130 time units.

3.3.4 A Python Class for JSSP Instances

This structure of a JSSP instance can be represented by the simple Python class.

What kind of information should the class provide? Obviously, it needs to give us the machine data M and the time data T data. To make working with these data easier, it should also tell us the number m of machines and the number n of jobs. And since we are working with well-known benchmark instances, we should also store the names of the instances.

The first design choice that we will have to make is whether we represent M and T as separate matrices, as one single "interleaved" matrix (like in the OR-Library format illustrated in Figure 3.1), or in a different way. Either choice is OK, but here I chose the latter.

I chose to put the data into a three-dimensional array: Each `Instance` is an array with one row for each job, one column for each operation, and each cell holds two values with the machine and the time spent on the machine. In other words, an $\mathcal{I}[i,j,0]$ holds the machine $M_{i,j}$ for the operation j of job i. $\mathcal{I}[i,j,1]$ holds the time $T_{i,j}$ that job i will spend on machine k. This is still fairly close to the OR-Library text format, while, at the same time, absolves me from fiddling around with the column index to figure out whether it points to a machine or a time. By using the third dimension, I can immediately see this.

Of course, I could also have two separate matrices or use the same format as the text files. This would mean that the code accessing the data would look a bit differently, but that would also be OK.

Now, if we want to represent matrices of integer values in Python, it is best to use `np.ndarrays`. These are memory efficient and accessing their elements is faster than using two-dimensional lists. So we will make our class inherit from `np.ndarray`. We will also add an attribute `name` with the instance name stored as string. Purely for the sake of simplicity, we also add the attributes `jobs` and `machines`, storing the number n of jobs and the number m, respectively.

In Listing 3.1, we give an excerpt of this class, i.e., a snippet of the original code where some methods and data verification has been omitted. We add a static method `from_resource(name)` to this class that can load any of the aforementioned benchmark JSSP instances from a resource inside our moptipy package directly based on its name (and return it as instance of `Instance`). This way, we can conveniently access all the necessary data of a job shop scheduling task. The actual code of the above, other utility methods, as well as sanity checks in the `__new__` constructor have here been omitted as they are unimportant for the understanding of the scenario.

3.4 Summary

We now have learned the first component of optimization problems. When we say "optimization problem" we do not actually mean a single problem that we can solve. We refer to a whole family of scenarios that have the same structure and can be solved by the same algorithms. There is not just *one* TSP or *one* JSSP. There are infinitely many of them. They are called problem *instances* $\mathcal{I}$.

Listing 3.1: Excerpt from a Python class for representing the data of a JSSP instance. (src)

```python
class Instance(Component, np.ndarray):
    """
    An instance of the Job Shop Scheduling Problem.

    Besides the metadata, this object is a three-dimensional np.ndarray
    where the columns stand for jobs and the rows represent the
    operations of the jobs. Each row*column contains two values (third
    dimension), namely the machine where the operation goes and the time
    it will consume at that machine: `I[job, operation, 0] = machine`,
    `I[job, operation, 1] = time` that the job spents on machine.
    """

    #: the name of the instance
    name: str
    #: the number of jobs == self.shape[0]
    jobs: int
    #: the number of machines == self.shape[1]
    machines: int
    # ... some more properties and methods ...
```

Chapter 4

The Solution Space

4.1 Definitions

As stated in Definition 2.1, an optimization problem asks us to make a choice between different possible solutions. We call them *candidate solutions*.

Definition 4.1 (Candidate Solution y) A *candidate solution* y is one potential solution of an optimization problem.

Definition 4.2 (Solution Space $\mathbb{Y}$) The *solution space* $\mathbb{Y}$ of an optimization problem is the set of all of its candidate solutions y.

Rule of Thumb 2 The best candidate solution(s) that an optimization algorithm has discovered are (one part) of its *output*.

Basically, the input of an optimization algorithm is the problem instance $\mathcal{I}$ and (one part of) the output would be (at least) one candidate solution $y \in \mathbb{Y}$. This candidate solution is the choice that the optimization process proposes to the human operator. It therefore holds all the data that the human operator needs to take action, in a form that the human operator can understand, interpret, and execute. During the optimization process, many such candidate solutions may be created and compared to find and return the best of them.

4.2 A Programmer's Perspective

From the programmer's perspective, the solution space is a data structure, e.g., a `class` in Python. An instance of this data structure is the candidate solution. On an abstract level, this data structure could be anything. It could be a `list`, a `np.ndarray`, a tree data structure, a `dict`, a graph, a construction plan for a train, anything.

Earlier, I said that we will do a lot of hands-on learning in this optimization book. We will look at things not *only* from the perspective of an algorithm scientist, but also from the perspective of a *programmer*. If a programmer is supposed to build algorithms that can deal with arbitrary data structures, she will first think about what kind of operations she will need to perform with them. In Listing 4.1, we give an excerpt example of such a "space API."

Now the data structures are basically containers. `np.ndarrays` are containers that can be filled with information. `lists` and `dicts` are such containers as well. If our algorithms later on should be able to work with arbitrary such container data structures, they will need a way to create them. `Space` therefore provides the method `create`. They will also need to copy the data from one container to another, for which the `copy` method is provided. A data structure instance y is always something that lives in the computer memory. But we *a*) need to show the contents of y to a user in a human-readable fashion, because they are the results of the optimization process, and *b*) need to be able to store and load them from files in order to be able to process them with other programs. For this purpose, the methods `to_str` and `from_str` are provided. They convert an instance $y \in \mathbb{Y}$ to a text string.

Listing 4.1: An excerpt of as base class for implementing space handlers. (src)

```python
class Space(Component):
    """
    A class to represent both search and solution spaces.

    The space basically defines a container data structure and basic
    operations that we can apply to them. For example, a solution
    space contains all the possible solutions to an optimization
    problem. All of them are instances of one data structure. An
    optimization as well as a black-box process needs to be able to
    create and copy such objects. In order to store the solutions we
    found in a text file, we must further be able to translate them to
    strings. We should also be able to parse such strings. It is also
    important to detect whether two objects are the same and whether
    the contents of an object are valid. All of this functionality is
    offered by the `Space` class.
    """

    def create(self) -> Any:
    def copy(self, dest, source) -> None:
    def to_str(self, x) -> str:
    def from_str(self, text: str) -> Any:
    def is_equal(self, x1, x2) -> bool:
    def validate(self, x) -> None:
```

By defining the interface `Space`, we provide a blueprint for the bookkeeping operations that we want to do with the candidate solutions y. We can *implement* this interface for all kinds of different solution spaces $\mathbb{Y}$. This will allow us to basically ignore the exact nature of $\mathbb{Y}$ in our algorithm implementations. We can create a variable to hold a y, we can copy it, and we can read and write it to a file or show its contents to the user.

4.3 Example: Job Shop Scheduling

What would be a candidate solution to a JSSP instance as defined in Section 3.3? Recall from Section 1.1.4 that our goal is to complete the jobs, i.e., the production tasks, as soon as possible. Hence, a candidate solution should tell us what to do, i.e., how to process the jobs on the machines.

4.3.1 Idea: Gantt Chart

This is basically what Gantt charts [44, 81] are for. A Gantt chart defines what each of our m machines has to do at each point in time. The operations of each job are assigned to time windows on their corresponding machines. You can find an example illustrated in Figure 4.1.

The Gantt chart contains one row for each machine. It is to be read from left to right, where the horizontal axis represents the time units that have passed since the beginning of the job processing. Each colored bar in the row of a given machine stands for a job and denotes the time window during which the job is processed. The bar representing operation j of job i is painted in the row of machine $M_{i,j}$ and its length equals the time requirement $T_{i,j}$.

Figure 4.1 illustrates one example solution of our `demo` instance as such a Gantt chart. We use a distinct color for each job. This chart defines that job 0 starts at time unit 0 on machine 0 and is processed there for ten time units. Then the machine idles until the 70th time unit, at which point it begins to process job 1 for another ten time units. After 15 more time units of idling, job 3 will arrive and be processed for 20 time units. Finally, machine 0 works on job 2 (coming from machine 3) for ten time units starting at time unit 150.

Machine 1 starts its day with an idle period until job 2 arrives from machine 2 at time unit 30 and is processed for 20 time units. It then processes jobs 1 and 0 consecutively and finishes with job 3 after another idle period. And so on.

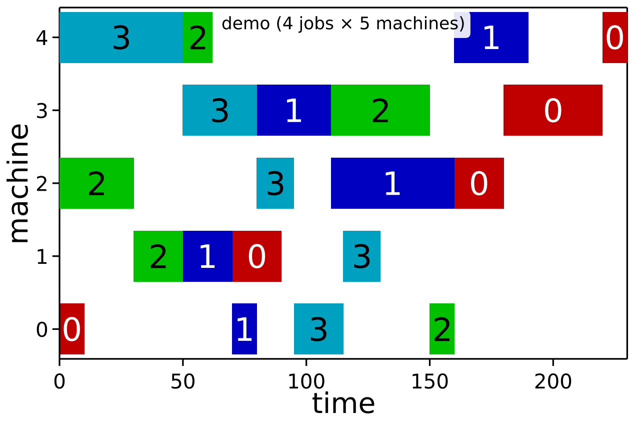


Figure 4.1: One example candidate solution for the demo instance given in Figure 3.1: A Gantt chart assigning a time window to each job on each machine.

Listing 4.2: Excerpt from a Python class for representing a Gantt chart, i.e., the data of a candidate solution to a JSSP: a subclass of `np.ndarray` to hold the data and a pointer to the JSSP instance. (src)

```python
class Gantt(np.ndarray):
    """
    A class representing Gantt charts.

    A Gantt chart is a diagram that visualizes when a job on a given
    machine begins or ends. We here represent it as a three-dimensional
    matrix. This matrix has one row for each machine and one column for
    each operation on the machine.
    In each cell, it holds three values: the job ID, the start, and the
    end time of the job on the machine. The Gantt chart has the
    additional attribute `instance` which references the JSSP instance
    for which the chart is constructed.
    Gantt charts must only be created by an instance of
    :class:`moptipy.examples.jssp.gantt_space.GanttSpace`.
    """
```

Listing 4.3: The contents of the array data of an instance of `Gantt` (see Listing 4.2) representing the solution illustrated in Figure 4.1.

```
#        job start  end job start  end job start  end  job start  end
Gantt([[[0,    0,  10], [1,  70,   80], [3,   95, 115], [2, 150,  160]],   # m0
       [[2,   30,  50], [1,  50,   70], [0,   70,   90], [3, 115,  130]],   # m1
       [[2,    0,  30], [3,  80,   95], [1, 110, 160], [0, 160,  180]],   # m2
       [[3,   50,  80], [1,  80, 110], [2, 110, 150], [0, 180,  220]],   # m3
       [[3,    0,  50], [2,  50,   62], [1, 160, 190], [0, 220,  230]]],  # m4
       dtype=int16)
```

If we wanted to create a Python `class` to represent the complete information from a Gantt diagram, it could look like Listing 4.2. Here, we again just subclass `np.ndarray` to store all data as three-dimensional array. The array has one row for each of the m machines. Each machine will process one operation of each job. Therefore, there will be with one column for each of the n operations to be executed on the machine. Each cell then holds the job ID, the start time, and the end time of the operation. Additionally, an instance of our `Gantt` class holds a reference to the JSSP instance (see Listing 3.1). This allows us to look up the instance information such as n, m, M, and T, e.g., for checking if everything in the Gantt chart is correct, as well as the instance name, e.g., for displaying it to the user.

The first row of the `Gantt` array corresponding to Figure 4.1 would look as follows: Its first element are the values `[0, 0, 10]`, since the operation of jobs 0 takes place during the first 10 time units of

Listing 4.4: Excerpt of the implementation of the `Space` API Listing 4.1 for Gantt charts. (src)

```python
class GanttSpace(Space):
    """An implementation of the `Space` API of for `Gantt` charts."""
    def __init__(self, instance):
        self.instance = instance
        self.shape = (instance.machines, instance.jobs, 3)
        self.copy = np.copyto  # type: ignore

    def create(self):
        return Gantt(self)

    def to_str(self, x):
        return CSV_SEPARATOR.join(map(str, np.nditer(x)))

    def is_equal(self, x1, x2):
        return (x1.instance is x2.instance) and np.array_equal(x1, x2)

    def from_str(self, text):
        x = self.create()
        np.copyto(
            x,
            np.fromstring(text, dtype=self.dtype,
                          sep=CSV_SEPARATOR).reshape(self.shape))
        return x

    def validate(self, x):
        # Checks if a Gantt chart if valid and feasible.
        if not isinstance(x, Gantt):
            raise type_error(x, "x", Gantt)
        # the rest of the checks is not printed for brevity reasons...
```

the schedule on this machine. Then the entry `[1, 70, 80]` follows, indicating that job 1 is processed for the 10 time units starting at time index 70 at machine 0. The third entry, `[3, 95, 115]` states that job 3 arrives at the machine 0 at time unit 95 and is processed for 20 time units until time index 115. The fourth and last entry, `[2, 150, 160]` denotes that job 2 is processed by the machine in the time window 150..160.

The second row is for machine 1. Its entries `[2, 30, 50]`, `[1, 50, 70]`, `[0, 70, 90]`, and `[3, 115, 130]` represent the sequence of operations we observed in Figure 4.1: job 2 is first, followed by job 1, job 0, and, finally, by job 3, which starts after 115 time units. The complete array contents are illustrated in Listing 4.3.

This way to represent Gantt charts as data structures is easy to read, understand, and visualize. Actually, we could also chose a more compact representation: We do not necessarily need to store the end times of the operations as well. We know how long each job i needs on any machine j takes based on the instance data $T_{i,j}$. We could furthermore order the elements of each row by job and not by starting time, in which case we would not need to store the job IDs either. Thus, having only the start times stored would be sufficient. Another form of representing a solution would therefore be to just map each operation to a starting time, leading to $m * n$ integer values per candidate solution [69].

However, also storing the job IDs end times of the operations will make our life a bit easier here. It allows the human operator to directly see what is going on. She can directly tell each machine or worker what to do and when to do it, without needing to look up any additional information from the problem instance data.

In Listing 4.4 we implement the `Space` interface from Listing 4.1 for Gantt charts. This code snippet is only here to show that we can indeed offer all the necessary functionality for creating and copying Gantt charts y and for converting them to and from strings. How these methods work exactly is not important here – the only thing that is important is that they are there. All in all, we now know how we can represent possible solutions y for the JSSP inside of our computer. We also know how we can operate on the set $\mathbb{Y}$ of all such solutions in the most primitive manner, e.g., copy them or print them as text. Combined with the knowledge how JSSP instances look like and how this layout is related to the layout of Gantt charts, we are one step further.

4.3.2 The Size of the Solution Space

We choose the set of all Gantt charts for m machines and n jobs as our solution space $\mathbb{Y}$. Now it is not directly clear how many such Gantt charts exist, i.e., how big $\mathbb{Y}$ is. If it is rather small, i.e., if there are only few possible different solutions y, then we can maybe simply enumerate all of them, check them one after the other, and pick the best one. If it is rather larger, then we may need to have a better idea. So how big is it?

If we allow arbitrary useless waiting times between operations, then we could create arbitrarily many different valid Gantt charts for any problem instance. Let us therefore assume that no time is wasted by waiting unnecessarily.

If there was only one machine, i.e., if $m = 1$, then there are $n! = \prod_{i=1}^{n} i$ possible Gantt charts (because there are exactly that many ways to arrange n jobs on one machine). Here, $n!$, called the factorial of n, is the number of different permutations (or orderings) of n objects. If we have three jobs a, b, and c, then there are $3! = 1 * 2 * 3 = 6$ possible permutations, namely (a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), and (c, b, a). Each permutation would equal one possible sequence in which we can process the jobs on *one* machine. If we have three jobs and one machine, then six is the number of possible different Gantt charts that do not waste time. If we have four jobs, then there are $4 * 3 * 2 * 1 = 24$ possible arrangements.

What happens if we have more than one machines ($m > 1$)? Then, the possible arrangements simply "multiply." If we have $m = 2$ machines and $n = 3$ jobs, then there are 6 possible arrangements on the first machine and, for *each* such arrangement, there are 6 possible arrangements on the second machine. Thus, if we have $n = 3$ jobs and $m = 2$ machines, we then would have $(3!) * (3!) = (3!)^2 = 36$ possible Gantt charts. For $m = 3$ machines, it is then $(n!)^3$, and so on. In the general case, we obtain Equation 4.1 for the size $|\mathbb{Y}|$ of the solution space $\mathbb{Y}$.

$$|\mathbb{Y}| = (n!)^m \tag{4.1}$$

We list some examples for the number $|\mathbb{Y}|$ of possible schedules which do not waste time uselessly for different values of n and m in Table 4.1. We find that even small problems with just $m = 5$ machines and $n = 5$ jobs have billions of possible solutions. The eight more realistic problem instances which we will use as benchmarks in our book already have more solutions than what we could ever enumerate, list, or store with any conceivable hardware or computer. For the smallest of them, `orb06`, which has ten jobs and ten machines, we already could theoretically construct $3.959 * 10^{65}$ possible Gantt charts. The biggest of them, `ta70`, has 50 jobs and 20 machines, which means that the number of theoretically possible solutions is about $4.587 * 10^{1'289}$, a number that would easily fill a whole single sheet of paper if written down...

From this, it becomes immediately clear: We cannot simply test all possible solutions and pick the best one. We will need some more sophisticated algorithms to solve these problems. This is the topic of this book.

However, the fact that we can generate many possible Gantt charts for a JSSP instance with n jobs and m machines does not mean that all of them are actual *feasible* solutions.

4.4 Summary

In this section we introduced a vital component of optimization problems: The solution space $\mathbb{Y}$. This space contains all the possible candidate solutions y for the problem. Now, we do not yet know whether a candidate solution y is a good or a bad solution. In the next sections, we will learn how to decide about that.

To ground the basic concepts of $\mathbb{Y}$ and y to some practical experience, we looked again at the JSSP as example. We found that Gantt charts are a proper way to express solutions to this scheduling problem. A Gantt chart basically tells which operation should be carried out by which machine at which time. We then briefly sketched that a Gantt chart can be represented as an instance of a `class` in Python and that we can implement basic functionality for handling such classes as an instance of `Space` interface. We can use this later on, for now it shall not be important.

What is important is that we decided to only consider Gantt charts that do not include useless waiting time. If an operation can be executed on a machine, we will not wait uselessly but execute it

Table 4.1: The size $|\mathbb{Y}|$ of the solution space $\mathbb{Y}$ (without schedules that stall uselessly) for selected values of the number n of jobs and the number m of machines of an JSSP instance $\mathcal{I}$ (later compare also with Figure 1.7).

example	n	m	$\|\mathbb{Y}\|$
	2	2	4
	2	3	8
	2	4	16
	2	5	32
	3	2	36
	3	3	216
	3	4	1'296
	3	5	7'776
	4	2	576
	4	3	13'824
	4	4	331'776
`demo`	4	5	7'962'624
	5	2	14'400
	5	3	1'728'000
	5	4	207'360'000
	5	5	24'883'200'000
`orb06`	10	10	$\approx 3.959 * 10^{65}$
`la38`	15	15	$\approx 5.591 * 10^{181}$
`abz8`	20	15	$\approx 6.193 * 10^{275}$
`yn4`	20	20	$\approx 5.278 * 10^{367}$
`swv14`	50	10	$\approx 6.772 * 10^{644}$
`dmu67`	40	20	$\approx 1.710 * 10^{958}$
`dmu72`	50	15	$\approx 1.762 * 10^{967}$
`ta70`	50	20	$\approx 4.587 * 10^{1'289}$

right away. While there is an infinite number of Gantt charts with arbitrary useless waiting time, the number of Gantt charts without such time is limited (Equation 4.1). However, while the search space size is finite, it grows very quickly and, thus, $|\mathbb{Y}|$ is very huge, even for seemingly small instances $\mathcal{I}$ (with small m and n). This is a very common situation in optimization and one of the fundamental forces that requires us to develop clever algorithms.

Rule of Thumb 3 Solution spaces tend to be huge, even for seemingly small problem instances.

Chapter 5

The Feasibility of the Solutions

For many optimization problems, we can pick any element y from the solution space $\mathbb{Y}$ and return it as output of an optimization algorithm. For many *other* optimization problems, some elements of $\mathbb{Y}$ are not permissible for different reasons. We we call these reasons *constraints*.

5.1 Definitions

Definition 5.1 (Constraint) A *constraint* is a rule imposed on the solution space $\mathbb{Y}$ which can either be fulfilled or violated by a candidate solution $y \in \mathbb{Y}$.

Definition 5.2 (Feasibility) A candidate solution $y \in \mathbb{Y}$ is *feasible* if and only if it fulfills all constraints.

Definition 5.3 (Infeasibility) A candidate solution $y \in \mathbb{Y}$ is *infeasible* if it is *not feasible*, i.e., if it violates at least one constraint.

5.2 A Programmer's Perspective

There is not much to discuss here from a programmer's perspective because there are many different ways to deal with constraints. In our moptipy `Space` API illustrated in Listing 4.1, we provide the method `validate`. The programmer can implement this method to check whether a solution is feasible or not. If not, the method is supposed to throw a `ValueError` if it is not.

5.3 Example: Job Shop Scheduling

In order to be a feasible solution for a JSSP instance, a Gantt chart must indeed fulfill a couple of *constraints*:

1. All operations of all jobs must be assigned to their respective machines and properly be completed.
2. Only the jobs and machines specified by the problem instance must occur in the chart.
3. An operation must be assigned a time window on its corresponding machine which is exactly as long as the operation needs on that machine.
4. The operations of one job cannot intersect or overlap. The next operation of one job can only begin after the previous operation of the job has been completed.
5. Each machine can only carry out one operation at a time.
6. Once a machine begins to process an operation, it cannot stop until the operation is complete, i.e., no preemption is possible

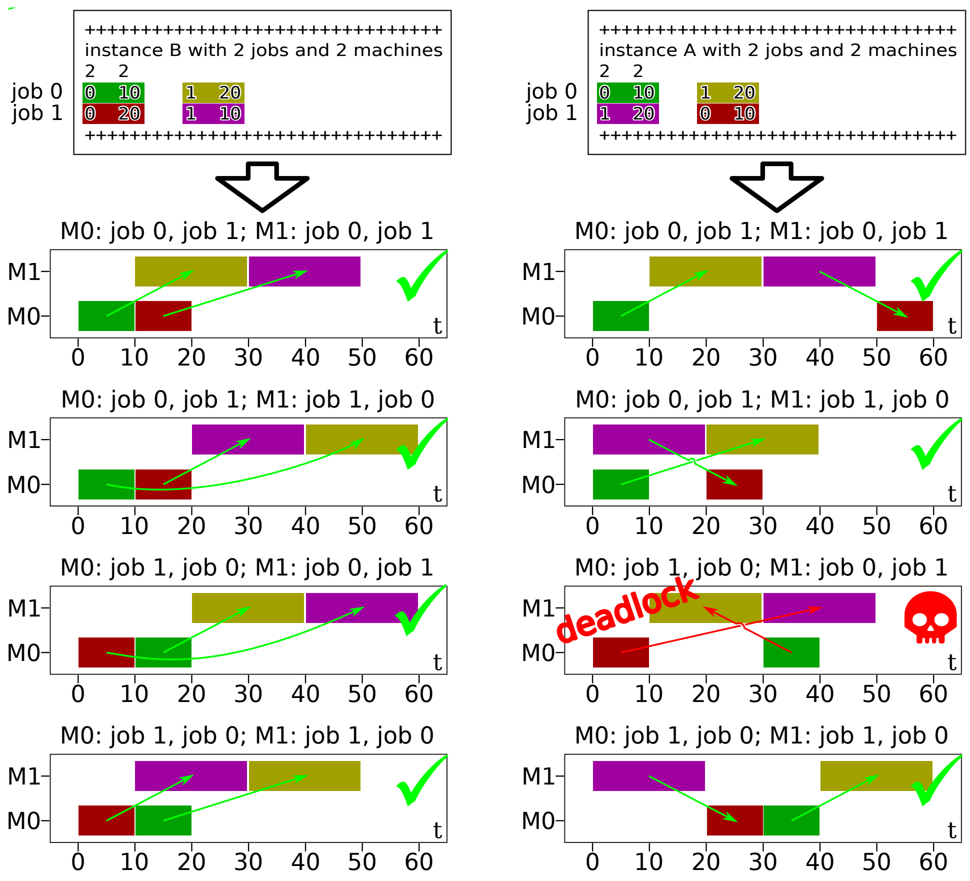


Figure 5.1: Two different JSSP instances with $m = 2$ machines and $n = 2$ jobs.The left one has four feasible corresponding Gantt charts. The right one has only three feasible solutions and one infeasible one.

7. The precedence constraints of the operations must be honored. In other words, if the instance says that operation 1 of job 1 should take place before operation 2 of job 1, then this must also be the case in the Gantt chart.

While the first six *constraints* are rather trivial, the latter one proofs problematic. Imagine a JSSP with $n = 2$ jobs and $m = 2$ machines. There are $(2!)^2 = (1 * 2)^2 = 4$ possible Gantt charts. Assume that the first job needs to first be processed by machine 0 and then by machine 1, while the second job first needs to go to machine 1 and then to machine 0. A Gantt chart which assigns the first job to be the first on machine 1 and the second job first to be the first on machine 0 cannot be executed in practice, i.e., is *infeasible*, as such an assignment does not honor the precedence constraints of the jobs. Instead, it contains a deadlock.

The third schedule in the right column of Figure 5.1 illustrates exactly this case. Machine 0 should begin by doing job 1. Job 1 can only start on machine 0 after it has been finished on machine 1. At machine 1, we should begin with job 0. Before job 0 can be put on machine 1, it must go through machine 0. So job 1 cannot go to machine 0 until it has passed through machine 1, but in order to be executed on machine 1, job 0 needs to be finished there first. Job 0 cannot begin on machine 1 until it has been passed through machine 0, but it cannot be executed there, because job 1 needs to be finished there first. A cyclic blockage has appeared: no job can be executed on any machine if we follow this schedule. This is called a deadlock.

No jobs overlap in the schedule. All operations are assigned to proper machines and receive the right processing times. Still, the schedule is infeasible, because it cannot be executed or written down without breaking the precedence constraint.

Hence, there are only three out of four possible Gantt charts that work for this problem instance. For a problem instance where the jobs need to pass through all machines in the same sequence, however, all possible Gantt charts will work, as also illustrated in the left column of Figure 5.1. The number of

Table 5.1: The minimal number $\min \#\text{feasible}$ of feasible solutions over all instances with specific sizes n and m (compare with Table 4.1).

example	n	m	$\min \#\text{feasible}$	$\lvert\mathbb{Y}\rvert$
Figure 5.1	2	2	3	4
	2	3	4	8
	2	4	5	16
	2	5	6	32
	3	2	22	36
	3	3	63	216
	3	4	147	1'296
	3	5	317	7'776
	4	2	244	576
	4	3	1'630	13'824
	4	4	7'451	331'776
	5	2	4'548	14'400
	5	3	91'461	1'728'000

actually feasible Gantt charts in $\mathbb{Y}$ thus can be different for different problem instances.

Different JSSP instances can have different numbers $\#\text{feasible}$ of possible *feasible* Gantt charts, even if they have the same numbers of machines and jobs. We just saw this in Figure 5.1. Naturally, for a given setting of m and n, we are interested in the minimum $\min \#\text{feasible}$ of this number, i.e., the *smallest value* that $\#\text{feasible}$ can take on over all possible instances with n jobs and m machines.

Now, I don't know how to compute this number in any efficient and we only want to see out of academic curiosity. So I just enumerated all the instances for a given combination of n and jsspMachines. For each instance, I enumerated all the possible Gantt charts and counted how many were feasible, i.e., computed $\#\text{feasible}$ for that instance. After I was done with enumerating all the instances, I know the smallest value of $\#\text{feasible}$, i.e., $\min \#\text{feasible}$, for that combination of n and m. I know that there must be a better way to get this number, probably with dynamic programming, but for now, the crude method will be sufficient. Of course, it only works for very small values of n and m.

The results in Table 5.1 show that, at least for some instances, most of the possible Gantt charts might be infeasible, as $\min \#\text{feasible}$ can be much smaller than $\lvert\mathbb{Y}\rvert$.

Rule of Thumb 4 If a problem has feasibility constraints, then we often find that most of the candidate solutions are infeasible.

This is very annoying. The potential existence of infeasible solutions means that we cannot just pick a good element from $\mathbb{Y}$ (according to whatever *good* means), we also must be sure that it is actually *feasible*. An optimization algorithm that may sometimes return infeasible solutions will not be acceptable.

5.4 Summary

In this section, we discovered that not every Gantt chart that we can draw for a given JSSP instance is also *feasible*. Some of them have deadlocks (Figure 5.1). Obviously, whatever we will later do to find *good* Gantt charts, we must never return an infeasible one, i.e., one that cannot actually be executed. To make matters worse, we discovered that it is entirely possible that the vast majority of Gantt charts that we could come up with for a given instance could be infeasible.

So the JSSP is not just a problem where the number of possible solutions is far too huge to test them all, most of these solutions may also be wrong. And the JSSP is an "average citizen" in optimization.

Not all optimization problems have feasibility constraints. Often, the constraints are very straightforward such as assigning each operation once to exactly one machine in a Gantt chart. Or to visit each city only once in a TSP. In such cases, we can relatively easily take care of them in our algorithm implementation and they do not cause us any specific headache.

Some constraints are structurally hard, such as not permitting deadlocks in a Gantt charts. They may or may not be hard to deal with in when building valid candidate solutions.

Other constraints may not be structural but instead of a metric nature, e.g., "…with a cost not exceeding 1000 RMB." Such constraints may not even be hard, e.g., permit some violation as long as it is small.

Either way: If there are constraints, then it is likely that most possible solutions are infeasible, as noted in Rule of Thumb 4. And for our example problem, the JSSP, this is the case. So optimization is indeed interesting.

Chapter 6

The Objective Function

We now know the most important elements of input and output data for an optimization algorithm: the problem instances $\mathcal{I}$ and candidate solutions $y \in \mathbb{Y}$, respectively. But we do not just want to produce some output. We do not just want to find "any" candidate solution. We want to find the "good" ones. For this, we need a measure rating the solution quality.

6.1 Definitions

Definition 6.1 (Objective Function f) An *objective function* $f : \mathbb{Y} \mapsto \mathbb{R}$ numerically rates the quality of a candidate solution $y \in \mathbb{Y}$ from the solution space $\mathbb{Y}$.

Definition 6.2 (Objective Value z) An *objective value* $z = f(y)$ of the candidate solution $y \in \mathbb{Y}$ is the value that the objective function f takes on for y.

Definition 6.3 (Minimization) An objective function is subject to *minimization* if smaller objective values indicate better solutions.

Definition 6.4 (Maximiation) An objective function is subject to *maximization* if larger objective values indicate better solutions.

Without loss of generality, we assume that all objective functions are subject to *minimization*. In this case, a candidate solution $y_1 \in \mathbb{Y}$ is better than another candidate solution $y_2 \in \mathbb{Y}$ if and only if $f(y_1) < f(y_2)$. If $f(y_1) > f(y_2)$, then y_2 would be better and for $f(y_1) = f(y_2)$, there would be no benefit in choosing either solution over the other, at least from the perspective of the optimization criterion f. The minimization scenario fits to situations where f represents a cost, a time requirement, or, in general, an amount of required resources.

In the TSP, for example, a tour is better if it is shorter. In the BPP, a packing plan is better if it requires fewer bins.

Maximization problems, i.e., where the candidate solution with the higher objective value is better, are problems where the objective function represents profits, gains, or any other form of positive output or result of a scenario. Maximization and minimization problems can be converted to each other by simply negating the objective function. In other words, if f is the objective function of a maximization problem, we can solve the minimization problem with objective $-f$ and get the same result, and vice versa.

6.2 A Programmer's Perspective

From the perspective of a programmer, the general concept of objective functions can be represented by the `class` given in Listing 6.1. The `evaluate` function of this class accepts one element `x` from the solution space and returns a numerical value. We can implement this function in any way we want, meaning that we can accommodate all types of solution spaces and optimization goals.

Listing 6.1: A base class for objective functions. (src)

```
class Objective(Component):
    """
    An objective function subject to minimization.

    An objective function represents one optimization criterion that
    is used for rating the solution quality. All objective functions in
    our system are subject to minimization, meaning that smaller values
    are better.
    """
    def evaluate(self, x):
        """
        Evaluate a solution ‘x‘ and return its objective value.

        The return value is either an integer or a float and must be
        finite. Smaller objective values are better, i.e., all objective
        functions are subject to minimization.

        :param x: the candidate solution
        :return: the objective value
        """
```

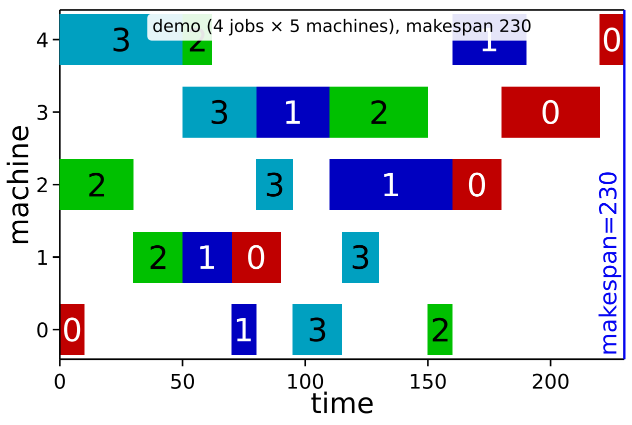


Figure 6.1: The makespan, i.e., the time when the last job is completed, for the example candidate solution illustrated in Figure 4.1 for the demo instance from Figure 3.1.

6.3 Example: Job Shop Scheduling

What could be a suitable objective function for the JSSP? As stated in Section 1.1.4, our goal is to complete the production jobs as soon as possible.

Definition 6.5 (Makespan) In manufacturing, the *makespan* is the time difference between the start and finish of a sequence of jobs or tasks.

Since we assume that all jobs begin at time index 0 in our Gantt charts, the makespan is the time when the last operation of the last job is finished. Obviously, the smaller this value, the earlier we are done with all jobs, the better is the plan. The makespan is therefore subject to minimization. As illustrated in Figure 6.1, the makespan is the time index of the right-most edge of any of the machine rows/schedules in the Gantt chart. In the figure, this happens to be the end time 230 of the last operation of job 0, executed on machine 4.

Our objective function f is thus equivalent to the makespan and subject to minimization. Based on our candidate solution data structure `Gantt` developed Listing 4.2, we can easily compute f. Remember that each instance of `Gantt` is basically a `np.ndarray` with three dimensions. The first dimension (accessed via the first index j) is the machine index. The second dimension (accessed via the second

Listing 6.2: An implementation of the `class Objective` Listing 6.1 to represent the makepan objective function for JSSPs. (src)

```python
@numba.njit(nogil=True, cache=True)
def makespan(x):
    """
    Get the makespan corresponding to a given `Gantt` chart.

    The makespan corresponds to the maximum of the end times of the
    last operation on each machine. This is jitted for performance.

    :param x: the Gantt chart.
    :return: the maximum of any end time stored in the chart
    """
    return int(x[:, -1, 2].max())  # maximum of end time of last op

class Makespan(Objective):
    """Compute the makespan of a `Gantt` chart (for minimization)."""
    def __init__(self, instance):
        #: The fast call forwarding to the makespan function.
        self.evaluate = makespan  # type: ignore
```

index k) lets us access the operations in the order in which they are carried out on that machine. To compute the makespan, we need to only access the very last operation done on any machine. For this, we can set second index to $k = -1$, which, in Python, gives us the last element of a `list` or `np.ndarray`. Finally, in the third dimension (accessed via the third index l), we store the job ID (at $l = 0$), the start time (at $l = 1$), and the end time of the operation (at $l = 2$). Obviously, we only need that last value, so we can set the third index to $l = 2$. So the second and third index into a `Gantt` chart matrix are defined and we only need to compute the maximum value that we get for any of the possible values of first index. In Listing 6.2, we implement exactly this concept in the easiest possible way. (For the Python aficionado: we apply the performance tricks from TODO.)

With this objective function f, subject to minimization, we have defined that a Gantt chart y_1 is better than another Gantt chart y_2 if and only if $f(y_1) < f(y_2)$.[1]

6.4 Summary

The objective function f is the third crucial component of optimization problems. The instance data $\mathcal{I}$ gives us the definition of a specific scenario as *input*. The solution space $\mathbb{Y}$ contains all the possible solutions and we will return at least one candidate solution $y \in \mathbb{Y}$ *output* of our algorithm.

Rule of Thumb 5 For each solution $y \in \mathbb{Y}$ returned as output of an optimization algorithm, we should also return the corresponding objective value $z = f(y)$.

[1]under the assumption that both are feasible, of course

Chapter 7

Global Optima of the Objective Function

We now know three key-components of an optimization problem. We are looking for a candidate solution $y^{\star} \in \mathbb{Y}$ that has the best objective value $f(y^{\star})$ for a given problem instance $\mathcal{I}$. But what is the meaning "best"?

7.1 Definitions

Assume that we have a single objective function $f : \mathbb{Y} \mapsto \mathbb{R}$ defined over a solution space $\mathbb{Y}$. This objective function is our primary guide during the search and we are looking for its *global optima*.

Definition 7.1 (Global Optimum) If a candidate solution $y^{\star} \in \mathbb{Y}$ is a *global optimum* for an optimization problem defined over the solution space $\mathbb{Y}$, then there is no other candidate solution in $\mathbb{Y}$ which is better.

If the objective function is subject to minimization, then each global optimum is a global minimum of the objective function.

Definition 7.2 (Global Minimum) For every *global minimum* $y^{\star} \in \mathbb{Y}$ of single-objective optimization problem with solution space $\mathbb{Y}$ and objective function $f : \mathbb{Y} \mapsto \mathbb{R}$ subject to minimization, it holds that $f(y) \geq f(y^{\star}) \forall y \in \mathbb{Y}$.

Notice that Definition 7.2 does not state that the objective value of $y^{\star}$ needs to be better than the objective value of all other possible solutions. The reason is that there may be more than one global optimum, in which case all of them have the same objective value. Thus, a global optimum is not defined as a candidate solutions better than all other solutions, but as a solution for which no better alternative exists.

The real-world meaning of "globally optimal" is nothing else than "superlative" [12]. If we solve a JSSP for a factory, our goal is to find the *shortest* makespan. If we try to pack the factory's products into containers, i.e., solve a BPP,, we look for the packing that needs the *least* amount of containers. If we solve a vehicle routing problem to serve several customers, then we may either want to serve the *most* customers, use the *least* amount of vehicles, or travel the *shortest* overall distance. Thus, optimization means searching for such superlatives, as illustrated in Figure 7.1. Vice versa, whenever we are looking for the cheapest, fastest, strongest, best, biggest or smallest "thing", then we have an optimization problem at our hands [42].

Definition 7.3 (Exact Algorithm) An *exact* algorithm guarantees to always find a globally optimal solution for an optimization problem if it is given sufficiently much runtime to complete its computation.

At the present state of research, any algorithm guaranteeing to always find the optimal solution of any $\mathcal{NP}$-hard optimization problem may require a runtime that is exponential in the *problem scale*. Very early in this book, in Figure 1.7, we saw that exponential growth is very problematic.

Figure 7.1: Optimization is the search for superlatives [12].

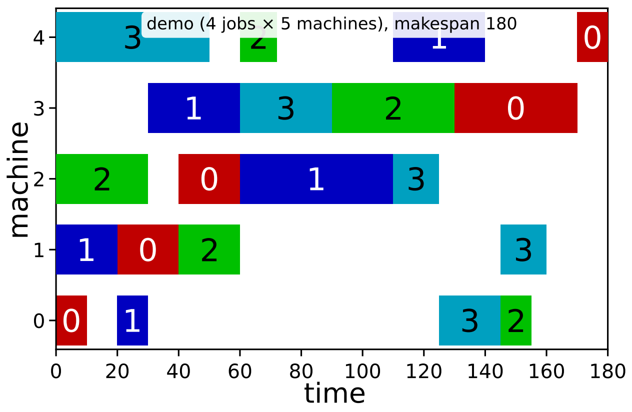


Figure 7.2: The optimal solution of the demo instance given in Figure 3.1.

This does not mean that exact algorithms *always* need a very long time. One the one hand, if the problem scale is small, then even an exponential runtime is not very long. On the other hand, we can imagine a TSP instance with millions of cities which all are located on a circle. Or we can imagine a BPP where all the items to be packed have the same size. Such problem instances would be very easy to solve, regardless of their scale.

However, for $\mathcal{NP}$-hard problems, there is a correlation between the size $|\mathbb{Y}|$ of the solution space $\mathbb{Y}$ and the runtime an *exact* algorithm may need.

Definition 7.4 (Heuristic Algorithm) A *heuristic* algorithm does *not guarantee* to find a globally optimal solution, but it will find one solution of hopefully good quality within a hopefully shorter runtime.

7.2 Example: Job Shop Scheduling

For the JSSP, there exists a simple and fast algorithm that can find the optimal schedules for problem instances with exactly $m = 2$ machines *and* where all n jobs need to be processed by the two machines in exactly the same order [40]. If our application always falls into such a special case of the problem we are dealing with, we may be lucky to find an efficient way to always solve it to optimality.

The general version of the JSSP, however, is indeed $\mathcal{NP}$-hard [15, 47], meaning that we cannot expect to solve it to global optimality in reasonable time. For the JSSP, the problem scale is defined by the numbers of machines m and jobs n involved. In Table 4.1 we saw that the size of the solution space $\mathbb{Y}$ grows very quickly with m and n. And so will the runtime of exact methods.

For small instances of the JSSP, however, we can still find the optimal solution by simply enumerating all possible Gantt charts and picking the one with shortest makespan, i.e., the best objective value. In

Figure 7.2, we illustrate the optimal solution for our small `demo` instance that was discovered this way. This method will obviously not work on bigger problems.

Rule of Thumb 6 Algorithms working well on small-scale toy problem instances do not necessarily work well on reasonably-sized or large-scale instances.

From this we can immediately conclude the following rule, which is an important basic wisdom that we must never forget throughout our career:

Rule of Thumb 7 In order to correctly understand the performance and behavior of an algorithm, testing it on small-scale instances is insufficient.

7.3 Summary

In this section, we learned a bit more about the nature of optimization. In particular, we formalized the concept of the best-possible solutions, the global optima $y^{\star}$ in the solution space $\mathbb{Y}$ based on the objective function f.

We also re-iterated the reason why we need metaheuristic optimization: Many optimization problems are $\mathcal{NP}$-hard and solving them to guaranteed optimality will often take too much time. Therefore, developing a good (meta-)heuristic algorithm, which cannot provide *guaranteed* optimality but will give close-to-optimal solutions in practice, is a good choice.

We also implicitly learned that, despite being $\mathcal{NP}$-hard, we can solve small-scale instances of the JSSP to optimality. But we can do this only for the really small instances. And this is the case for many of the well-known optimization problems.

Chapter 8

Bounds of the Objective Function

If we apply a heuristic algorithm, then we do not have the guarantee that the solution we get is optimal. Usually, we do not even know if the best solution we currently have is optimal or not. The most basic mistake that we can read in papers on optimization again and again is this: The claim that a metaheuristic returns optimal solutions (without further proof or considerations such as those below). It does not. None of them do. They can return good solutions, maybe better solutions than what we can get with any other algorithm within acceptable runtime. But we usually do not know if a solution is optimal or not.

Well. Usually. But not always.

In some cases, we be able to compute a *lower bound* $\mathrm{lb}(f)$ for the objective function.

8.1 Definitions

Definition 8.1 (Lower Bound) The *lower bound* $\mathrm{lb}(f)$ of a function f is a value such that $f(x) \geq \mathrm{lb}(f) \forall x$.

We know that it is not possible that any solution can have a quality better than $\mathrm{lb}(f)$. If a solution $y \in \mathbb{Y}$ exists with $f(y) = \mathrm{lb}(f)$, then this solution must be a global optimum, i.e., it is actually $y^{\star}$.

However, notice the "$\geq$" in Definition 8.1. It means that maybe no solution exists that has this objective value. In this case, the objective value $f(y^{\star})$ of the global optimum $y^{\star}$ would actually be larger, i.e., $f(y^{\star}) > \mathrm{lb}(f)$.

Having a lower bound of the objective function therefore is not directly useful for solving the problem itself. Still, it can at least help us to understand whether our method for solving the problem is good. For instance, if we have developed an algorithm for approximately solving a given problem and the qualities of the solutions we get are close to the lower bound, then we know that our algorithm is good. If we even find a solution whose quality equals the lower bound, then we 1. know that it is optimal and 2. can stop our algorithm immediately, as we cannot further improve on this.

If we get close to the lower bound, then we know that improving the result quality of the algorithm may be hard, maybe even impossible, and probably not worthwhile. However, if we cannot produce solutions as good as or close to the lower quality bound, this does not necessarily mean that our algorithm is bad.

It should be noted that it is *not* necessary to know the bounds of objective values. Lower bounds are a *"nice to have"* feature allowing us to better understand the performance of our algorithms.

8.2 A Programmer's Perspective

In Listing 6.1, we presented the API for implementing objective functions. We extend this API by a function `lower_bound()` that can return a lower bound or $-\infty$ if no bound is known. (Indeed, $-\infty$ is also a valid lower bound that fulfills Definition 8.1.) Furthermore, our experiment execution API will automatically stop the optimization processes if a solution $y^{\star}$ with $f(y^{\star}) = \mathrm{lb}(f)$ is reached.

8.3 Example: Job Shop Scheduling

We have already defined our solution space $\mathbb{Y}$ for the JSSP in Listing 4.2 and the objective function f in Listing 6.2. A Gantt chart with the shortest possible makespan is then a global optimum. There may be multiple globally optimal solutions, which then would all have the same makespan.

When facing a JSSP instance $\mathcal{I}$, we do not know whether a given Gantt chart is the globally optimal solution or not, because we do not know the shortest possible makespan. There is no direct way in which we can compute this optimal makespan. Actually, if we could, we would probably have solved the problem already. However, we can, at least, compute some *lower bound* $\mathrm{lb}(f)$ for the best possible makespan.

A trivial lower bound for the makespan is always 0 time units. No schedule can complete faster than that. Of course, this would also be a useless lower bound, because it does not tell us anything.

Rule of Thumb 8 A lower bound $\mathrm{lb}(f)$ of the objective function f is the better and the more useful, the higher it is.

The highest possible lower bound for the quality of an optimal solution would be exactly that quality itself (because otherwise it would violate Definition 8.1). As said, we are not able to build such a bound for the JSSP. But we can do other things.

For instance, we know that a job i needs at least as long to complete as the sum $\sum_{j=0}^{m-1} T_{i,j}$ over the processing times of all of its operations. Imagine that we have two machines. If there is one job whose first operation needs 20 time units on the first machine and its second operation needs 30 time units on the second machine, then we can never complete this JSSP in less than $20 + 30 = 50$ time units. It is also clear that no schedule can complete faster then the longest job. Even if the other jobs in this example would need less than 50 time units in total, we still need need at least 50 time units. So we already have one lower limit.

Furthermore, we know that the makespan of the optimal schedule also cannot be shorter than the latest "finishing time" of any machine j. This finishing time is at least as big as the sum b_j of the runtimes of all the operations for this machine. If we have two jobs, and one operation of the first job needs 30 time units on the first machine and some operation of the second job needs 60 time units on the first machine, then we will need at least $30 + 60 = 90$ time units to complete the schedule.

We can actually further refine this idea: Each machine may have some least initial idle time a_j: If the operations for machine j never come first in their job, then for each job, we need to sum up the runtimes of the operations coming *before* the one on machine j. The least initial idle time a_j is then the smallest of these sums. This may be 0, if there is at least one job that *first* goes to the machine, or greater than zero if no such job exists.

Similarly, each machine has a least idle time c_j at the end. This is greater than zero if there is no job whose *last* operation is on the machine. Then, whenever the last job assigned to the machine has been processed by it, it still needs to go to go elsewhere and the machine must always remain idle for some time at the end of the schedule. As lower bound for the fastest schedule that could theoretically exist, we therefore get:

$$\mathrm{lb}(f) = \max\left\{\max_i\left\{\sum_{j=0}^{m-1} T_{i,j}\right\}, \max_j\left\{a_j + b_j + c_j\right\}\right\} \tag{8.1}$$

Equation 8.1 is implemented in Listing 8.1, following the ideas from [66].

Figure 8.1 again illustrates the globally optimal solution for our small `demo` JSSP instance already shown in Figure 7.2. Here we were lucky: The objective value of this solution happens to be the same as the lower bound for the makespan. Upon closer inspection, the limiting machine is the one at index 3.

We will find this by again looking at Figure 3.1. Regardless with which job we would start here, it would need to initially wait at least $a_3 = 30$ time units. The reason is that not a single first operation of any job starts at machine 3. Every job starts at another machine. So machine 3 can never begin its work at time index 0, since every job first needs to be processed by at least one other machine before getting to machine 3. Job 0 would get to machine 3 the earliest after 50 time units, job 1 after 30, job 2 after 62, and job 3 after again 50 time units. From all of these values, only the smallest one is relevant, which here is $a_3 = 30$ time units.

Listing 8.1: An implementation of Taillard's algorithm [66] represented in Equation 8.1 to compute the lower bound of the makespan of a JSSP instance. (src)

```python
def compute_makespan_lower_bound(machines, jobs, matrix):
    # get the lower bound of the makespan with the algorithm by Taillard
    jobtimes = np.zeros(jobs, npu.DEFAULT_INT)  # get array for job times
    machinetimes = np.zeros(machines,
                            npu.DEFAULT_INT)  # machine times array
    machine_start_idle = npu.np_ints_max(machines, npu.DEFAULT_INT)
    machine_end_idle = npu.np_ints_max(machines, npu.DEFAULT_INT)

    for jobidx in range(jobs):  # iterate over all jobs
        jobtime = 0  # the job time sum
        for i in range(machines):  # iterate over all operations
            machine, time = matrix[jobidx, i]  # get operation data
            machinetimes[machine] += time  # add up operation times
            machine_start_idle[machine] = min(  # update with...
                machine_start_idle[machine], jobtime)  # ...job time
            jobtime += time  # update job time by adding operation time

        jobtimes[jobidx] = jobtime  # store job time
        jobremaining = jobtime  # iterate backwards to get end idle times
        for i in range(machines - 1, -1, -1):  # second iteration round
            machine, time = matrix[jobidx, i]  # get machine for operation
            machine_end_idle[machine] = min(  # update by computing...
                machine_end_idle[machine],  # the time that the job...
                jobtime - jobremaining)  # needs _after_ operation
            jobremaining -= time  # and update the remaining job time

    # get the maximum of the per-machine sums of the idle and work times
    machines_bound = (machine_start_idle + machine_end_idle +
                      machinetimes).max()
    # get the longest time any job needs in total
    jobs_bound = jobtimes.max()

    return int(max(machines_bound, jobs_bound))  # return bigger one
```

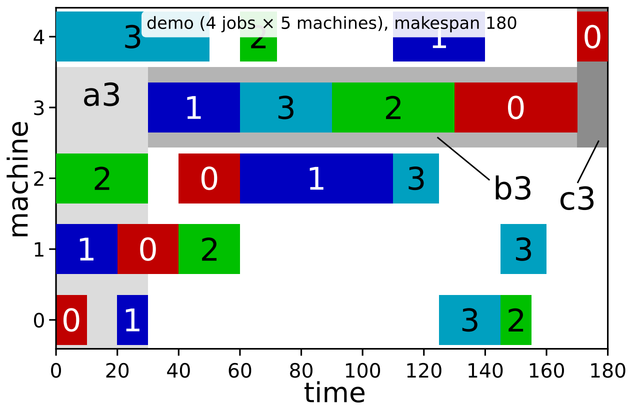


Figure 8.1: A globally optimal solution of the demo instance Figure 3.1, whose makespan happens to be the same as the lower bound from Equation 8.1.

Table 8.1: The lower bounds $\mathrm{lb}(f)$ for the makespan of the optimal solutions for our example problems. For some instances, research literature [74] (last column) provides better (i.e., higher) lower bounds $\mathrm{lb}(f)^{\star}$ than our algorithm in Listing 8.1.

name	n	m	$\mathrm{lb}(f)$	$\mathrm{lb}(f)^{\star}$	source for $\mathrm{lb}(f)^{\star}$
demo	5	4	180	180	Equation 8.1
abz8	20	15	566	648	[72]
dmu67	40	20	5454	5589	[29]
dmu72	50	15	6216	6395	[29]
la38	15	15	943	1196	[68]
orb06	10	10	930	1010	[38]
swv14	50	10	2968	2968	Equation 8.1
ta65	50	20	2725	2725	Equation 8.1
ta79	100	20	5358	5358	Equation 8.1
yn4	20	20	818	929	[72]

No job in the demo instance finishes at machine 3 either. This means that even if machine 3 finishes has completed all of its work, there definitely will be some operations that still need to be processed on another machine. Job 0, for instance, needs to be processed by machine 4 for 10 time units after it has passed through machine 3. So regardless when machine 3 finishes, the schedule will need at least 10 additional time units. Job 1 requires 80 more time units after finishing at machine 3, job 2 also 10 time units, and job 3 again 50 time units. From these values, again, the only smallest one is relevant, namely $c_3 = 10$ time units. In other words, machine 3 needs to wait at least 30 time units before it can commence its work and will remain idle for at least 10 time units after processing the last operation.

In between, it will need to work for exactly $b_3 = 140$ time units, the total sum of the running time of all operations assigned to it. This means that no schedule can complete faster than $F_3 = a_3 + b_3 + c_3 = 30 + 140 + 10 = 180$ time units. Thus, Figure 8.1 illustrates the optimal solution for the demo instance (and the lower bound).

Interestingly, while the last job in Figure 8.1 is on machine 4, machine 4 does not limit us. We could start by putting job 3 first on machine 4, so this machine could theoretically begin its work immediately at time unit 0. Also, we could finish both job 0 and job 1 at machine 4, so there also would not necessarily be an idle time at the end of the schedule on machine 4. Both of these facts can be seen in Figure 8.1. If we could schedule all jobs on machine 4 without delay, then we would need $50 + 12 + 30 + 10 = 102$ time units. Without starting delay and and ending idle time, this machine would limit schedules to not be shorter than 102 time units. Since we know that machine 3 imposes a makespan no less than 180 time units, machine 4 has no impact on the lower bound of the makespan.

The lower bounds for the makespans of our example problems are listed in Table 8.1. At the time of this writing, there are four instances (abz8, dmu67, dmu72, and yn4) in our benchmark set for which no solutions have been found yet whose makespan equals the best available lower bound. This either means that these four problems have not yet been solved to optimality or that their optimal solution has already been found but has a makespan worse than the lower bound. In the table, we also provide better, i.e., higher lower bounds $\mathrm{lb}(f)^{\star}$ that are available for some instances. We took these from our meta-study [74], which aggregates many results from different papers and existing studies.

8.4 Summary

Now, we do not know whether it is actually possible to find a schedule whose makespan equals the lower bound. There may simply not be any way to arrange the jobs such that no operation stalls any other operation too much. This is why the value $\mathrm{lb}(f)$ is called lower bound: We know no solution can be better than this, but we do not know whether a solution with such minimal makespan exists. However, if our algorithms produce solutions with a quality close to $\mathrm{lb}(f)$, we know that we are doing well. Also, if we would actually find a solution with that makespan, then we would know that we have perfectly solved the problem.

Chapter 9

Search Spaces and Encodings

The solution space $\mathbb{Y}$ is the data structure that "makes sense" from the perspective of the user, the decision maker, who will be supplied with one instance of this structure (a candidate solution y) at the end of the optimization procedure. But $\mathbb{Y}$ not necessarily is the space that is most suitable for searching inside.

We have already seen that there are several constraints that apply to the Gantt charts. For every problem instance, different solutions may be feasible. Besides the constraints, the space of Gantt charts also looks kind of unordered, unstructured, and messy.

Let's say that we want to generate a valid Gantt chart for a JSSP instance. We would step-by-step fill the Gantt chart by placing the different operations in the right order, without adding useless waiting times. Now if we wanted to take a feasible Gantt chart and try to tweak it in some way, we begin getting into trouble. If we did not insert useless waiting times, then each operation occurs at the earliest time where it can be placed based on all previously assigned operations. Thus, to move an operation around, we would first need to try to swap and move some previous operations. And then, if we succeeded somehow and the operation now indeed is moved to earlier time, we would need to move all the operations that come after it forward as well. During all of this, we need to keep track of the instance data $\mathcal{I}$ to not accidentally violate some constraint. Thus, while we do have found a reasonable representation of solutions for the JSSP in Gantt charts, it is only really suitable to represent single, fixed solutions. It is not really suitable to explore a space $\mathbb{Y}$ of different solutions – at least unless we are willing to write complicated code dealing with the feasibility constraints.

Actually, many data structures that represent real-world objects with their features, be it Gantt charts, construction plans of airplane wings, or plans for electronic circuits are specialized and do not lend them themselves to be directly processed by general algorithms.

It would thus be nice to have a compact, clear, and easy-to-understand way to explore different candidate solutions. Something that we can instantiate and modify in a simple way without getting in trouble

9.1 Definitions

Definition 9.1 (Search Space) The *search space* $\mathbb{X}$ is a representation of the solution space $\mathbb{Y}$ suitable for exploration by an algorithm.

Definition 9.2 (Point in the Search Space) The elements $x \in \mathbb{X}$ of the search space $\mathbb{X}$ are called *points* in the search space.

Definition 9.3 (Decoding Function) The *decoding function* $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ is a left-total relation which maps each point $x \in \mathbb{X}$ of the search space $\mathbb{X}$ to one candidate solution $y \in \mathbb{Y}$ in the solution space $\mathbb{Y}$.

"Left-total" here means that every point in $\mathbb{X}$ is mapped to one point in $\mathbb{Y}$. Some points in $\mathbb{X}$ may be mapped to the same point in $\mathbb{Y}$. There even may be some points in $\mathbb{Y}$ for which no corresponding point in $\mathbb{X}$ exists.

Definition 9.4 (Representation, Encoding) The solution space $\mathbb{Y}$, search space $\mathbb{X}$, and the decoding function γ together are called the *encoding* or the *representation*.

The solution space $\mathbb{Y}$ is what the user cares about. The optimization algorithm, however, *only* works on the search space $\mathbb{X}$. It does not need to know or care about what the candidate solutions are. The candidate solutions can be complex structures, such as the shape of the nose of a fast train [37, 43]. It is hard to imagine how to search inside the space of all possible such shapes in a targeted way. However, maybe we could encode the surfaces of the train noses as vectors of real numbers. The search space could then just be an n-dimensional real vector space. This changes everything. We know and understand these vector spaces since high school. We have all kinds of tools available to search in it in an ordered fashion, ranging from distance metrics to vector mathematics. Suddenly, the problem becomes easier to approach algorithmically. Moreover, since real vector spaces are very common, there already exists a wide variety of algorithms that can perform optimization over them.

A good encoding, i.e., a suitable search space $\mathbb{X}$ and mapping $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ can make our life much easier.

For applying an optimization algorithm, we therefore usually choose a data structure $\mathbb{X}$ which we can understand intuitively. Ideally, it should be possible to define concepts such as distances, similarity, or neighborhoods on this data structure. Spaces that are especially suitable for searching in include, for example:

1. subsets of n-dimensional real vectors, i.e., $\mathbb{R}^n$,
2. the set of all possible permutations of n objects, and
3. a number of n yes-no decisions, which can be represented as bit strings of length n, spanning the space $\{0, 1\}^n$.

For such spaces, we can relatively easily define good search methods and can rely on a large amount of existing research work and literature. If we are lucky, then our solution space $\mathbb{Y}$ is already "similar" to one of these well-known and well-researched data structures. Then, we can set $\mathbb{X} = \mathbb{Y}$ and use the identity mapping $\gamma(x) = x \; \forall x \in \mathbb{X}$ as decoding function. In other cases, we will often prefer to map $\mathbb{Y}$ to something similar to these spaces and define γ accordingly.

The mapping γ does not need to be injective, as it may map two points x_1 and x_2 to the same candidate solution even though they are different ($x_1 \neq x_2$). Then, there exists some redundancy in the search space. We would normally like to avoid redundancy, as it tends to slow down the optimization process [45]. Being injective is therefore a good feature for γ.

The mapping γ also does not necessarily need to be surjective, i.e., there can be candidate solutions $y \in \mathbb{Y}$ for which no $x \in \mathbb{X}$ with $\gamma(x) = y$ exists. However, such solutions then can never be discovered. If the optimal solution would be among those unreachable ones, then, well, it could not be found by the optimization process. Being surjective is therefore a good feature for γ.

Finally, and as a side note: Technically speaking, γ does not even necessarily be a function. It could be a randomized procedure, meaning that two invocations could lead to different results. But let's not take things too far here.

9.2 A Programmer's Perspective

In Listing 4.1, we already have defined a simple API to provide common operations for (solution) spaces. We can reuse this very same API for search spaces too. Additionally, we need a function that can convert from points in the search space to candidate solutions.

The class given in Listing 9.1 provides the blueprint for a function `decode` which translates one point `x` in the search space to a candidate solution instance `y` of the solution space. This `decode` function corresponds to the general definition $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ of the encoding. An implementation of `decode` will overwrite whatever contents were stored in the object `y`, i.e., we assume the objects `y` can be modified.

Thus, if you want to use a certain search space $\mathbb{X}$ and a different solution space $\mathbb{Y} \neq \mathbb{X}$, you would provide an encoding $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ by deriving a new class from `Encoding` and implementing γ as `decode`. The optimization algorithm can then search in the simple space $\mathbb{X}$ whereas the user and the objective

Listing 9.1: A base class for encodings. (src)

```python
class Encoding(Component):
    """The encoding translates from a search space to a solution space."""
    def decode(self, x, y):
        """
        Translate from search- to solution space.

        Map a point `x` from the search space to a point `y`
        in the solution space.

        :param x: the point in the search space, remaining unchanged.
        :param y: the destination data structure for the point in the
            solution space, whose contents will be overwritten
        """
```

function f only get to see the candidate solutions $y \in \mathbb{Y}$. This is possible because between these two components, the decoding function γ translates the points $x \in \mathbb{X}$ generated by the optimization algorithm to the candidate solutions in $\mathbb{Y}$.

9.3 Example: Job Shop Scheduling

In our JSSP example problem, the candidate solutions are Gantt charts. We developed the class `Gantt` given in Listing 4.2 to represent their data. This data can easily be interpreted and visualized by the user. Yet, it is not that clear how we can efficiently create such solutions, especially feasible ones, let alone how to *search* in the space of Gantt charts.[1] What we would like to have is a *search space* $\mathbb{X}$, which can represent the possible candidate solutions of the problem in a more machine-tangible, algorithm-friendly way.

The JSSP is a very well-known problem. Comprehensive overviews about different such search spaces for the JSSP can be found in [2, 16, 78, 83]. We here will develop only one single idea which I find particularly appealing.

9.3.1 Idea: 1-dimensional Encoding

Imagine you would like to construct a Gantt chart as candidate solution for a given JSSP instance. How would you do that? Well, we know that each of the n jobs has m operations, one for each machine. We could simply begin by choosing one job and placing its first operation on the machine to which it belongs, i.e., write it into the Gantt chart. Then we again pick a job, take the first not-yet-scheduled operation of this job, and "add" it to the end of the row of its corresponding machine in the Gantt chart. Of course, we cannot pick a job whose operations all have already be assigned. We can continue doing this until all operations of all jobs are assigned. The result will be a *feasible* solution.

This feasible solution is defined by the order in which we chose the jobs. Such an order can be described as a simple, linear string of job IDs, i.e., of integer numbers. Each job ID would appear m times, because that is how many operations the job has. We would process such a string from the beginning to the end. Whenever encountering a job ID, we assign the first not-yet assigned operation of the job to its correspoinding machine. We will always get a feasible Gantt chart as result. It is not possible to produce a deadlock (see Chapter 5), because we will only allocate an operation to a machine after having placed all operations that come before it in the same job.

This decoding procedure can best be described by an example. In the `demo` instance, we have $m = 5$ machines and $n = 4$ jobs. Each job thus has $m = 5$ operations that must be assigned to the machines. We use an integer string x of length $m * n = 20$ denoting the priority of the operations. We *know* the order of the operations per job as part of the problem instance data $\mathcal{I}$. We therefore do not need to store it in the string x. We just include the ID of each job $m = 5$ times in the string.

[1] Of course, there are many algorithms that can do that and we could design one such algorithm if we would seriously think about it, but here we take the educational route where we investigate the full scenario with $\mathbb{X} \neq \mathbb{Y}$.

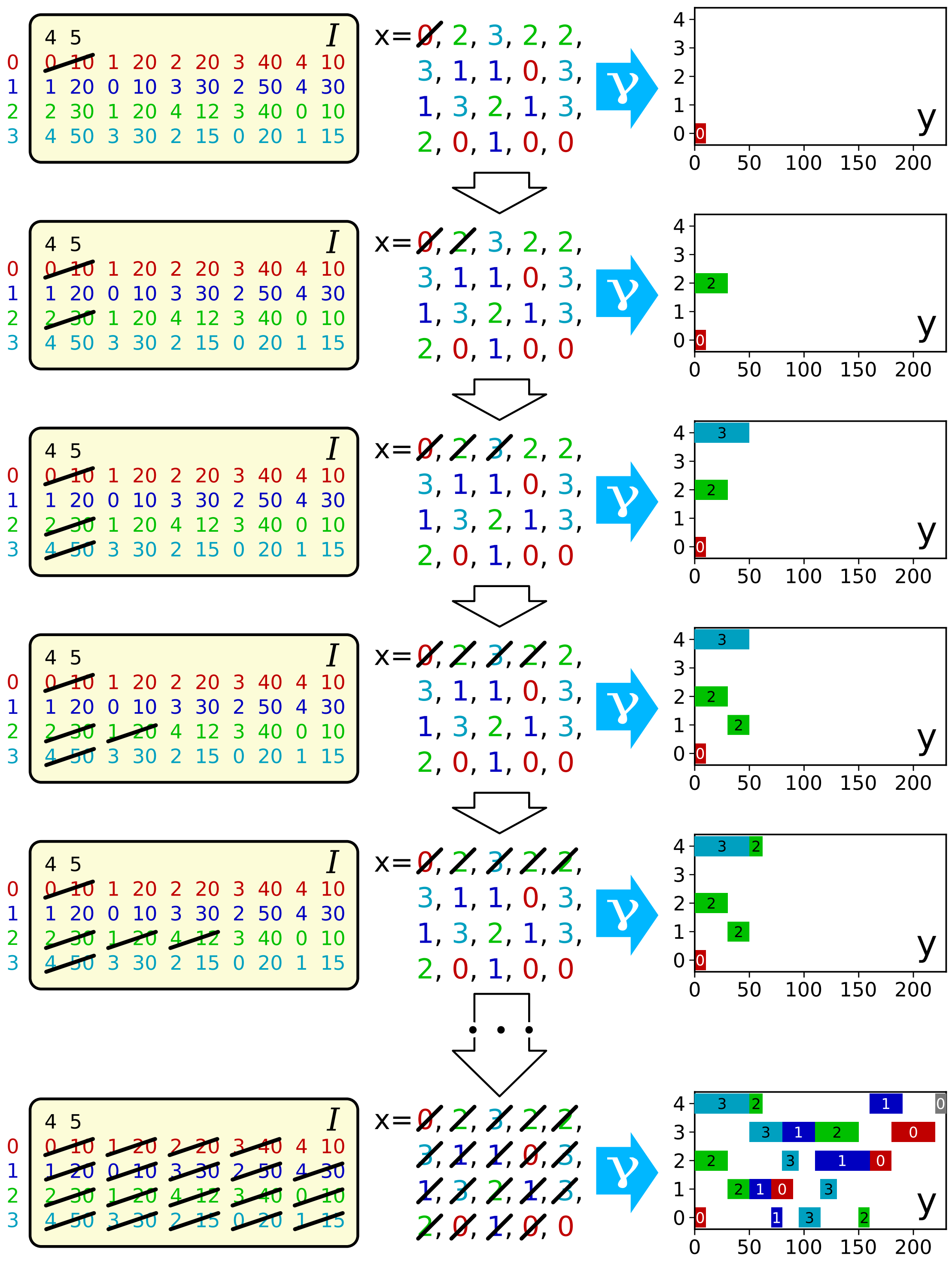


Figure 9.1: Illustration of the first five steps and the second-to-last step of the decoding of an example point in the search space to a candidate solution.

The encoding represents the order in which we assign the n jobs, and each job must be picked m times. Our search space is thus somehow similar to the set of permutations of $n*m$ objects mentioned earlier, but instead of permutations, we have *permutations with repetitions*.

A point $x \in \mathbb{X}$ in the search space $\mathbb{X}$ for the `demo` JSSP instance would thus be an integer string of length 20. As example, we choose $x = (0, 2, 3, 2, 2, 3, 1, 1, 0, 3, 1, 3, 2, 1, 3, 2, 0, 1, 0, 0)$.

Let us now exercise the decoding procedure $\gamma(x)$. In Figure 9.1, we sketch several of its steps. For each step, we show the instance data $\mathcal{I}$ on the left, the point x in the search space in the middle, and the current state of the Gantt chart y on the right hand side.

The decoding of the string x starts with an empty Gantt chart. This string is interpreted from left to right, as illustrated in the figure. The first value is 0, which means that, in the first step, job 0 is assigned to a machine. From the instance data, we know that job 0 first must be executed for 10 time units on machine 0. The job is thus inserted on machine 0 in the chart. Since machine 0 is initially idle and thus immediately ready to be used, the operation can be placed at time index 0. We also know that this operation can definitely be executed, i.e., won't cause a deadlock, because it is the first operation of the job. Once we have placed it in the chart, we cross it out to mark it as assigned. This happens in the first line in our figure.

The next value in the string is 2, meaning that we now need to insert an operation from job 2. No operation from this job was processed yet, so we pick its first operation. From the instance data, we see that it should go to machine 2 for 30 time units. In our current Gantt chart, no job has yet been assigned to machine 2, so we can place it there at time index 0. We now cross out this operation as well. This is done in the second line of the figure.

We then encounter the value 3 in the string for the first time. Job 3 first goes to machine 4 for 50 time units. Machine 4 is still unused, so we can place it there directly and mark it as assigned, as shown in line 3 of our figure.

Next we encounter job 2 again, i.e., for the second time. We have already marked its first operation as assigned. So we now need to allocate its second operation. It should go to machine 1 for 20 time units. While machine 1 has no job assigned to it yet, we cannot place the operation at time index 0. We first need to wait for the first operation of job 2 to complete, which happens at time index 30. Hence, we can start the operation at that time.

In the fifth row of Figure 9.1, we again find job 2. We need to assign its third operation, which goes to machine 4 and will need 12 time units there. It can only start after the second operation of the job is completed, which happens after $30 + 20 = 50$ time units. Also, machine 4 is used by job 3, whose first operation will be completed there also after 50 time units. Therefore, the third operation of job 2 can begin there at time index 50.

We then again encounter job 3 in x, which means we need to assign its second operation. This operation goes to machine 3 and can start at time index 50, after the first operation of the job has been completed. Then we will encounter job 1 for the first time and allocate its first operation to machine 1 for 20 time units. It can start after the second operation of job 2, which is already assigned to that machine, completes, i.e., at time index 50.[2] Directly afterwards, job 1 needs to be assigned again and its second operation will go to machine 0. It can start at time index $50 + 20 = 70$, namely after its first operation is completed.

We continue this iterative process. The last row of Figure 9.1 illustrates the second-to-last decoding step. It places the second-to-last operation of job 0 onto machine 3. The previous (i.e., third) operation of the job was on machine 2 and completed there after 180 time units. Machine 3 is idle at that time, so operation 4 of job 1 will occupy it from time index 180 to 220. This only leaves only the last operation of job 1 to be assigned, which will take 10 time units on machine 4. It can start there directly at time index 220 and will finish on time index 230 (illustrated in dark gray in the figure).

The Gantt chart then is complete. Whenever we assigned a operation $i > 0$ of any given job to a machine, then we already had assigned all operations at smaller indices first. No deadlock can occur and y must therefore be feasible.

In Listing 9.2, we illustrate how such an encoding can be implemented. It basically is a function translating an `np.ndarray` of integers to a `Gantt` chart. We put the algorithm into a function `decode`, so that we can mark it for compilation with numba to improve the performance, utilizing the performance

[2]The reader will notice: Actually, we could let it start at time index 0, which would not interfere with the operation of job 2. However, this would make the mapping more complicated, so we stick to the easier approach of just adding it at the end here.

Listing 9.2: An excerpt of the implementation of the operation-based encoding for the JSSP. (src)

```python
@numba.njit(nogil=True, cache=True)
def decode(x, machine_idx, job_time, job_idx, instance, y):
    """
    Map an operation-based encoded array to a Gantt chart.

    :param x: the source array, i.e., multi-permutation
    :param machine_idx: array of length ‘m‘ for machine indices
    :param job_time: array of length ‘n‘ for job times
    :param job_idx: length ‘n‘ array of current job operations
    :param instance: the instance data matrix
    :param y: the output array, i.e., the Gantt chart
    """
    machine_idx.fill(-1)  # all machines start by having done no jobs
    job_time.fill(0)  # each job has initially consumed 0 time units
    job_idx.fill(0)  # each job starts at its first operation

    for job in x:  # iterate over multi-permutation
        idx = job_idx[job]  # get the current operation of the job
        job_idx[job] = idx + 1  # and step it to the next operation
        machine = instance[job, idx, 0]  # get the machine id
        start = job_time[job]  # end time of previous operation of job
        mi = machine_idx[machine]  # get jobs finished on machine - 1
        if mi >= 0:  # we already have one job done?
            start = max(start, y[machine, mi, 2])  # earliest start
        mi += 1  # step the machine index
        machine_idx[machine] = mi  # step the machine index
        end = start + instance[job, idx, 1]  # compute end time
        y[machine, mi, 0] = job  # store job index
        y[machine, mi, 1] = start  # store start of job's operation
        y[machine, mi, 2] = end  # store end of job's operation
        job_time[job] = end  # time next operation of job can start

class OperationBasedEncoding(Encoding):
    # reusable variables __machine_time, __job_time, and __job_idx are
    # allocated in __init__; __matrix refers to instance data matrix

    def decode(self, x, y):
        decode(x, self.__machine_idx, self.__job_time, self.__job_idx,
               self.__instance, y)
```

tips discussed in TODO.

Besides implementing the decoding function γ, we also need to provide the functionality of our `Space` API for (search) spaces that are permutations with repetitions. This functionality will be needed to create, copy, store, load, and check the points in the search space. In Listing 9.3, we provide a very small excerpt of the implementation of the `Space` API for permutations of numbers stored in `np.ndarrays`. This class is quite general: We provide a `blueprint` string of the numbers that we want to arrange in the permutations. This could be a true permutation, e.g., $[1, 2, 3]$, or a permutation with repetitions, such as $[1, 1, 2, 2, 3, 3]$. The `create` method of the `Space` implementation will always return a copy of that blueprint array. We omit the conversion to and from text strings, as it can be implemented similarly as in `lst:jssp_gantt_space`. Validation can simply check whether each job ID occurs exactly as needed, i.e., m times in our case, and is thus also not printed. The static method `with_repetitions` instantiates the space for m repetitions of the elements $0..n-1$.

9.3.2 Advantages of this very simple Encoding

We now have a natural way to represent Gantt charts with a much simpler data structure. Because this method is so intuitive, it has been discovered by several researchers independently, the earliest being Gen, Tsujimura, and Kubota [28], Bierwirth [9, 10], and Shi, Iima, and Sannomiya [62], all in

Listing 9.3: Excerpt of the implementation of the `Space` API from Listing 4.1 for permutations with (or without) repetitions. (src)

```python
class Permutations(IntSpace):
    def __init__(self, base_string):
        #: a numpy array of the right type with the base string
        self.blueprint  =\
            np.array(string, dtype=self.dtype)

    def create(self):
        return self.blueprint.copy()  # Create copy of the blueprint.

    @staticmethod
    def with_repetitions(n, repetitions):
        return Permutations(list(range(n)) * repetitions)
```

the 1990s.

But what do we gain by using this encoding?

Well, we now have a very simple data structure $\mathbb{X}$ to represent our candidate solutions. It is much simpler than the Gantt charts. It is just a linear sequence of n numbers, each occuring a fixed amount m of times.

As a direct result, we also have very simple rules for validating a point $x \in \mathbb{X}$ in the search space: If it contains the numbers 0..$(n-1)$ each exactly m times, it represents a feasible candidate solution.

The candidate solution corresponding to a valid point from the search space will always be *feasible* [9]. The mapping γ will ensure that the order of the operations per job is always observed. Thus, we have solved the issue of deadlocks mentioned in Chapter 5. We know from Table 5.1, that the vast majority of the possible Gantt charts for a given problem might actually be infeasible. Now we do no longer need to worry about that. Our mapping γ also obeys the more trivial constraints, such as that each machine will process at most one job at a time and that all operations are eventually processed.

Finally, we also could modify our decoding function γ to adapt to more complicated and constraint versions of the JSSP if need be: For example, imagine that it would take a job- and machine-dependent amount of time for carrying a the material produced by on job from the current machine to the next machine. We could easily facilitate this by changing γ by adding this time to the starting time of the job. If there was a job-dependent setup time for each machine [4], which could be different if job 1 follows job 0 instead of job 2, then this could be facilitated easily as well. If our operations would be assigned to "machine types" instead of "machines" and there could be more than one machine per machine type, then the representation mapping could assign the operations to the next machine of their type which becomes idle. Our representation also trivially covers the situation where each job may have more than m operations, i.e., where a job may need to cycle back and pass one machine twice. It is also suitable for simpler scenarios, such as the Flow Shop Scheduling Problem (FSSP), where all jobs pass through the machines in the same, pre-determined order [27, 66, 78].

Many such different problem flavors can now be reduced to investigating the same space $\mathbb{X}$ using the same optimization algorithms, just with slightly different decoding function γ and/or objective functions f. Additionally, it becomes easy to indirectly create and modify candidate solutions by sampling points from the search space and moving to similar points, as we will see in the following chapters.

9.3.3 Size of the Search Space

It is relatively easy to compute the size $|\mathbb{X}|$ of our proposed search space $\mathbb{X}$ [62]. We do not need to make any assumptions regarding "no useless waiting time", as in Section 4.3.2, since useless delays cannot occur by default. Each element $x \in \mathbb{X}$ is a permutation of a multiset where each of the n elements occurs exactly m times. This means that the size of the search space can be computed as given in Equation 9.1.

$$|\mathbb{X}| = \frac{(m*n)!}{(m!)^n} \tag{9.1}$$

Table 9.1: The sizes $|\mathbb{X}|$ and $|\mathbb{Y}|$ of the search and solution spaces for selected values of the numbers of jobs n and machines m of a JSSP instance $\mathcal{I}$.

example	n	m	$\lvert\mathbb{Y}\rvert$	$\lvert\mathbb{X}\rvert$
	3	2	36	90
	3	3	216	1'680
	3	4	1'296	34'650
	3	5	7'776	756'756
	4	2	576	2'520
	4	3	13'824	369'600
	4	4	331'776	63'063'000
`demo`	4	5	7'962'624	11'732'745'024
	5	2	14'400	113'400
	5	3	1'728'000	168'168'000
	5	4	207'360'000	305'540'235'000
	5	5	24'883'200'000	$\approx 6.234 * 10^{14}$
`orb06`	10	10	$\approx 3.959 * 10^{65}$	$\approx 2.357 * 10^{92}$
`la38`	15	15	$\approx 5.591 * 10^{181}$	$\approx 2.252 * 10^{251}$
`abz8`	20	15	$\approx 6.193 * 10^{275}$	$\approx 1.432 * 10^{372}$
`yn4`	20	20	$\approx 5.278 * 10^{367}$	$\approx 1.213 * 10^{501}$
`swv14`	50	10	$\approx 6.772 * 10^{644}$	$\approx 1.254 * 10^{806}$
`dmu72`	50	15	$\approx 1.762 * 10^{967}$	$\approx 3.862 * 10^{1'226}$
`dmu67`	40	20	$\approx 1.710 * 10^{958}$	$\approx 2.768 * 10^{1'241}$
`ta70`	50	20	$\approx 4.587 * 10^{1'289}$	$\approx 1.988 * 101'648$

To better understand this equation, imagine the space of all permutations of the sequence $(1, 2, 3, 4)$. Clearly there are $4! = 4 * 3 * 2 * 1 = 24$ such permutations. Now let us imagine that one number, say 3, appears twice, e.g., we want all the permutations of $(1, 2, 3, 3, 4)$. There are five elements now, so there are $5! = 120$ possible ways to arrange them. However, the number 3 appears twice and it does not matter which of the 3s appears first, so we get $5!/2 = 60$ possible unique permutations. If we add another 3, i.e., 3 would appear three times and we have $(1, 2, 3, 3, 3, 4)$, then there would be $6!/3! = 720/6 = 120$ possible arrangements. If we now add another two 4s, we get $(1, 2, 3, 3, 3, 4, 4, 4)$. This sequence can be arranged in $8!/(3!*3!) = 40320/36 = 120$ possible ways. If each one of n different numbers appears m times, we hence have $(n * m)/(m)^n$ different possible permutations, i.e., obtain Equation 9.1.

We give some example values for this search space size $|\mathbb{X}|$ in Table 9.1. From the table, we can immediately see that the number of points in the search space, too, grows very quickly with both the number of jobs n and the number of machines m of a JSSP instance $\mathcal{I}$. If we compare the ordering of our example JSSP instances by search space size with the ordering by solution space, we find that there only is one disagreement: `dmu67` has the larger search space $\mathbb{X}$ but a smaller solution space $\mathbb{Y}$ compared to `dmu72`. Apart from this, larger solution spaces tend to correspond to larger search spaces as well.

For our `demo` JSSP instance with $n = 4$ jobs and $m = 5$ machines, we already have about 12 billion different points in the search space that represent 7 million possible non-wasteful candidate solutions.

We now find the drawback of our encoding: There is much redundancy in our mapping. The mapping γ is not injective. Some elements of the search space $\mathbb{X}$ will map to the same element in the solution space $\mathbb{Y}$.

For example, we could arbitrarily swap the first three numbers in the example string in Figure 9.1 and would obtain the same Gantt chart, because jobs 0, 2, and 3 start at different machines.

As said before, we should avoid redundancy in the search space. However, here we will stick with our proposed mapping because it is very simple, it solves the problem of feasibility of candidate solutions, and it allows us to relatively easily introduce and discuss many different approaches, algorithms, and sub-algorithms.

9.4 Summary

So far, we have focussed on elements that always must be present. We have learned that a concrete set of input data for the optimization algorithms is called a problem instance $\mathcal{I}$. We have learned that we need a data structure $\mathbb{Y}$ to represent candidate solutions y. We have learned that we need an objective function $f : \mathbb{Y} \mapsto \mathbb{R}$ that computes the cost of the candidate solutions (and that, usually, smaller values of f indicate better solutions).

In this section, however, we have learned *optional* components of the structure of optimization: the search space $\mathbb{X}$ and the decoding function $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ that translates points $x \in \mathbb{X}$ to candidate solutions $y \in \mathbb{Y}$. In many cases, neither $\mathbb{X}$ nor γ are needed and we can search in $\mathbb{Y}$ as-is. However, in many *other* cases, the data structure $\mathbb{Y}$ that we can present to the user and rate with the objective function f is not suitable to be explored directly during the optimization process. It may either be too constraint or too complicated.

Then, we may come up with a way γ to translate some very simple space $\mathbb{X}$, let's say bit strings, real vectors, or permutations to $\mathbb{Y}$. If we can do this, we immediately can reap several benefits: Many algorithms that work well on such spaces already exist. The spaces are simpler and it is easier to design search operators for them (we will learn later what that is). Maybe we can even take care of some of the constraints during the mapping γ, as is the case in our JSSP example.

Nevertheless, search space $\mathbb{X}$ and the decoding function γ are purely internal constructs. They are meaningful *only* for the optimization algorithm. They are entirely meaningless for the user. (Imagine handing a permutation with repetition as solution for a JSSP to the human operator of a factory...) They are also entirely meaningless from the perspective of the objective function f. Both the user and f only are interested in the candidate solutions $y \in \mathbb{Y}$. So the $\mathbb{X}$ and γ are hidden components that work their magic in the dark bowels of the optimization algorithms.

Chapter 10

Search Operators

One of the most important design choices of a metaheuristic optimization algorithm are the search operators employed.

10.1 Definitions

Definition 10.1 (Search Operator) An k-ary *search operator* $\mathrm{move} : \mathbb{X}^k \times \mathfrak{R} \mapsto \mathbb{X}$ is a left-total function which accepts k points in the search space $\mathbb{X}$ (and a source $\mathfrak{R}$ of randomness) as input and returns one point in the search space as output.

Special cases of search operators are

- nullary operators ($k = 0$, see Listing 10.1) sample a new point from the search space without using any information from an existing points,
- unary operators ($k = 1$, see Listing 10.2) sample a new point from the search space based on the information of one existing point, and
- binary operators ($k = 2$, see Listing 10.3) sample a new point from the search space by combining information from two existing points.

Operators that take existing points in the search space as input tend to sample new points which, in some sort, are similar to their inputs. They allow us to define proximity-based relationships over the search space, such as the common concept of neighborhoods.

Definition 10.2 (Neighborhood) A unary operator $\mathrm{move} : \mathbb{X} \times \mathfrak{R} \mapsto \mathbb{X}$ defines a *neighborhood* relationship over a search space where a point $x_1 \in \mathbb{X}$ is called a *neighbor* of a point $x_2 \in \mathbb{X}$ if and only if x_1 could be the result of an application of move to x_2.

10.2 A Programmer's Perspective

If we look at this from the perspective of the programmer, then a search operator is basically an object with a function that accepts a random number generator and k existing points in the search space $\mathbb{X}$. Since our search space data structures are re-useable containers such as `lists` or `np.ndarrays`, we also pass in one such container as destination to receive the newly sampled point. We can thus define a few very simple API for search operators as components:Whether, which, and how such such operators are used depends on the nature of the optimization algorithms and will be discussed later on.

Search operators are often *randomized*, which means invoking the same operator with the same input multiple times may yield different results. In the definitions, this is signified by the component $\mathfrak{R}$ in their input. Therefore Listings 10.1 to 10.3 all expect an instance of `np.random.Generator`, a pseudorandom number generator of the numpy library, as parameter.

Listing 10.1: A base class for nullary search operators. (src)

```python
class Op0(Component):
    """A base class to implement a nullary search operator."""
    def op0(self, random, dest):
        """
        Apply the nullary search operator to fill object ‘dest‘.

        Afterwards ‘dest‘ will hold a valid point in the search space.
        Often, this would be a point uniformly randomly sampled from the
        search space, but it could also be the result of a heuristic or
        even a specific solution.

        :param random: the random number generator
        :param dest: the destination data structure
        """
        raise ValueError("Method not implemented!")
```

Listing 10.2: A base class for unary search operators. (src)

```python
class Op1(Component):
    """A base class to implement a unary search operator."""
    def op1(self, random, dest, x):
        """
        Fill ‘dest‘ with a modified copy of ‘x‘.

        :param random: the random number generator
        :param dest: the destination data structure
        :param x: the source point in the search space
        """
        raise ValueError("Method not implemented!")
```

Listing 10.3: A base class for binary search operators. (src)

```python
class Op2(Component):
    """A base class to implement a binary search operator."""
    def op2(self, random, dest, x0, x1):
        """
        Fill ‘dest‘ with a combination of ‘x0‘ and ‘x1‘.

        :param random: the random number generator
        :param dest: the destination data structure
        :param x0: the first source point in the search space
        :param x1: the second source point in the search space
        """
        raise ValueError("Method not implemented!")
```

10.3 Example: Job Shop Scheduling

We will step-by-step introduce the concepts of nullary, unary, and binary search operators in the subsections of TODO on metaheuristics as they come. This makes more sense from a didactic perspective. However, we can guess what search operators would probably do in this context:

1. a nullary operator would randomly create a permutation with repetition where each of the values in $0..n-1$ occurs exactly m times.
2. a unary operator would accept such a permutation with repetitions and create a somewhat modified copy of it (that still obeys the validity constraint above), and
3. a binary operator would accept two and try to merge them in some reasonable (and yet, still somewhat random) way.

As to why and how we will do that ... that we will learn later.

10.4 Summary

There is not much to sum up here. We will really dig into the concept of search operators when we explore metaheuristic optimization methods. However, at this point, the keen reader may already anticipate some of the concepts that we learn later. On one side, we do have a space $\mathbb{Y}$ of solutions y and an objective function f that rates their quality. In the case of our JSSP example, we chose Gantt charts as $\mathbb{Y}$, which are not trivial to deal with. Therefore, on the other side, we defined a much simpler search space $\mathbb{X}$ and a decoding function γ that can map one element $x \in \mathbb{X}$ to an element $y \in \mathbb{Y}$. One main ingredient that seems to be missing before we can actually tackle an optimization problem are a method to sample points in $\mathbb{X}$ and to maybe navigate from one existing such point x_1 to another one x_2. This is what search operators can do. And *how* they do it, as said, will be explored a bit later.

Chapter 11

The Termination Criterion

We have seen that the search spaces for even small instances of the JSSP can already be quite large. We simply cannot enumerate all points in them, as it would take too long. While optimization algorithms usually do something much cleverer than simply checking all possible solutions, we already discovered the profoundly simply question: "If we cannot look at all possible solutions, how can we find the global optimum?" We then went a step further and asked: "If we cannot look at all possible solutions, how can we know whether a given candidate solution is the global optimum or not?" In some optimization scenarios, we can have good lower bounds $\mathrm{lb}(f)$ of the objective function f and if we find a solution $y^{\star} \in \mathbb{Y}$ with $f(y^{\star}) \leq \mathrm{lb}(f)$, then we know that we have solved the problem. However, usually the answer to both question is simply: *We cannot.*

Usually, we simply don't know if the best solution y_b we know so far is the global optimum or not. This leads us to another problem: If we do not know whether we found the best-possible solution or not, how do we know if we can stop the optimization process (whatever that might be) or should continue trying to solve the problem?

At first glance, there are two very simple answers to this question: We stop either "when the time is up" or "when we found a reasonably-good solution." Additionally, there are many more complicated answers [30], such as "when we were not able to improve upon y_b for a while."

11.1 Definitions

Definition 11.1 (Termination Criterion) The *termination criterion* $\mathrm{shouldTerminate} : \mathbb{S} \mapsto \{$`False`, `True`$\}$ is a function of the state $\mathbb{S}$ of the optimization process which becomes `True` if the optimization process should stop and remains `False` as long as it can continue.

Into such a termination criterion we can embed any combination of time or solution quality limits. We could, for instance, define a goal objective value z_g good enough so that we can stop the optimization procedure as soon as a candidate solution $y \in \mathbb{Y}$ has been discovered with $f(y) \leq z_g$, i.e., which is at least as good as the goal. We could set $z_g = \mathrm{lb}(f)$ if we have a reasonably good lower bound. We could also set to some acceptable quality limit resulting from our practical application scenario.

Alternatively – or in addition – we may define a maximum amount of time the user is willing to wait for an answer, i.e., a computational budget after which we simply need to stop.

Most optimization processes iteratively try to find better solutions. We could stop when, for a certain amount of time, no better solution was discovered. While the current-best solution may not be "reasonably good," continuing to try finding better solutions may be pointless anyway when the last improvement was made one minute after we started the optimization process . . . and, by now, that is ten hours ago. . .

Some algorithms maintain sets of solutions and iteratively work on all the solutions in the sets. A maximum number of such iterations was used as stopping criterion in some older works, but this is known to be a bad practice and, thus, strongly discouraged [59].

11.2 Example: Job Shop Scheduling

In our example domain, the JSSP, we can assume that the human operator will input the instance data $\mathcal{I}$ into the computer. Then she may go drink a coffee and expect the results to be ready upon her return. While she does so, can we solve the problem? Unfortunately, probably not. As said, for finding the best possible solution, if we are unlucky, we would need in invest a runtime growing exponentially with the problem size, i.e., m and n [15, 47]. So can we guarantee to find a solution which is, say, at most 1% worse, until she finishes her drink? Unfortunately, it was shown that there is *no* algorithm which can guarantee us to find a solution at most only 25% worse than the optimum within a runtime polynomial in the problem size [39, 80] in 1997. Since 2011, we know that *any* algorithm guaranteeing to provide schedules that are only be a constant factor (be it 25% or 1'000'000) worse than the optimum may need the dreaded exponential runtime [52]. So whatever algorithm we will develop for the JSSP, defining a some limit solution quality based on the lower bound of the objective value at which we can stop as the *only* termination criterion makes little sense, because the runtime needed to get there may simply be too long.

Hence, we let us look at this practically: The operator enters the problem instance data. Then she drinks a coffee. A termination criterion granting two minutes of runtime seems to be reasonable to me here. We should look for the algorithm implementation that can give us the best solution quality within that time window.

Of course, there may also be other constraints based on the application scenario. For example, it could be that only Gantt charts that can be implemented/completed within the working hours of a single day are acceptable (*feasible*) solutions in a real-world JSSP. We then might let the algorithm run longer than two minutes until such a solution was discovered. But, as said before, if an odd scenario occurs, it might take a long time to discover such a solution, if ever.

The human operator may also need to be given the ability to manually stop the process and extract the best-so-far solution y_b if need be. For our benchmark instances, however, this is not relevant and we can limit ourselves to the runtime-based termination criterion.

We will investigate many different optimization methods. Some are designed to make quick improvements and to find *reasonably good* solutions quickly. Others are designed to perform a broad search, to investigate many possible beneficial solution traits, in the hope to eventually find a *very good* solution. Now with two minutes, we set a fairly short runtime limit. Algorithms which do a broader search will appear to not perform well, because there will not be enough time for their advantages to become visible. Alternatively, if we had chosen a long runtime limit, then the quickly improving algorithms would appear worse. We discuss this issue in-depth in TODO. We will try out all the algorithms that we discuss. Please keep in mind that the fact that some will work out worse than others in our particular scenario does not mean that they are worse in general.

11.3 Summary

In this section, we raised the question when an optimization should stop. This is an important question, since we already know that we usually do not know whether we found the optimal solution or not. Sometimes, termination criteria arise from our practical scenario, e.g., our application defines a time limit and then, that is when we will stop. Sometimes, we have a lot of time available and could let the optimization process go on for quite a bit. However, even then, we do not want to waste computational time when it is already clear that our algorithm cannot make any improvement anymore. Finally, we may have a goal quality threshold z_g and after reaching it, we can stop. Sometimes we have a combination of the above. Either way, we have to stop eventually.

Chapter 12

Solving Optimization Problems

Thank you for sticking with me during this long and a bit dry introduction chapter. Why did we go through all of this long discussion? We did not even solve the JSSP yet…

Well, in the following you will see that we now are actually only a few steps away from getting good solutions for the JSSP. Or *any* optimization problem. Because we now have actually exercised a good share of the basic process that we need to go through whenever we want to solve a new optimization task.

1. The first thing to do is to understand the scenario information. This is the input data $\mathcal{I}$ that our program will receive, i.e., the problem instance.
2. The second step is to understand what our users will consider as a solution – a Gantt chart, for example. Then we need to define a data structure $\mathbb{Y}$ which can hold all the information of such a candidate solution.
3. Once we have the data structure $\mathbb{Y}$ representing a complete candidate solution, we need to know when a solution is good. We will define the objective function f, which returns one number (say the makespan) for a given candidate solution.
4. If we want to apply any of the optimization algorithms introduced in the following chapters, then we also to know when to stop. As already discussed, we usually cannot solve instances of a new problem to optimality within feasible time and often do not know whether the current-best solution is optimal or not. Hence, a termination criterion usually arises from practical constraints, such as the acceptable runtime.

All the above points need to be tackled in close collaboration with the user. The user may be the person who will eventually, well, use the software we build or at least a domain expert. The following steps then are our own responsibility:

1. In the future, we will need to generate many candidate solutions quickly, and these better be feasible. Can this be done easily using the data structure $\mathbb{Y}$? If yes, then we are good. If not, then we should think about whether we can define an alternative search space $\mathbb{X}$, a simpler data structure. Creating and modifying instances of such a simple data structure $\mathbb{X}$ should be much easier than $\mathbb{Y}$. Of course, defining such a data structure $\mathbb{X}$ makes only sense if we can also define an decoding function γ from $\mathbb{X}$ to $\mathbb{Y}$.
2. We select optimization algorithms and plug in the representation and objective function. We may need to implement some other algorithmic modules, such as search operations. In the following chapters, we discuss a variety of methods for this.
3. We test, benchmark, and compare several algorithms to pick those with the best and most reliable performance.

Rules of Thumb

Rule of Thumb 1: One part of the *input* of optimization algorithm is the instance data $\mathcal{I}$.

Rule of Thumb 2: The best candidate solution(s) that an optimization algorithm has discovered are (one part) of its *output*.

Rule of Thumb 3: Solution spaces tend to be huge, even for seemingly small problem instances.

Rule of Thumb 4: If a problem has feasibility constraints, then we often find that most of the candidate solutions are infeasible.

Rule of Thumb 5: For each solution $y \in \mathbb{Y}$ returned as output of an optimization algorithm, we should also return the corresponding objective value $z = f(y)$.

Rule of Thumb 6: Algorithms working well on small-scale toy problem instances do not necessarily work well on reasonably-sized or large-scale instances.

Rule of Thumb 7: In order to correctly understand the performance and behavior of an algorithm, testing it on small-scale instances is insufficient.

Rule of Thumb 8: A lower bound $\mathrm{lb}(f)$ of the objective function f is the better and the more useful, the higher it is.

Symbols

$\circ^\star$ the superscript $\star$ symbol denotes that $\circ$ is a global optimum.

$\circ_b$ the subscript b denotes that $\circ$ is the best-so-far element of an optimization process.

$\#\mathrm{feasible}$ the number of feasible solutions in the solution space $\mathbb{Y}$, see Definition 5.2.

$f(y)$ an objective function $f : \mathbb{Y} \mapsto \mathbb{R}$ computes the cost of a candidate solution $y \in \mathbb{Y}$, see Definition 6.1.

$\gamma(x)$ the decoding function $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ maps the candidate solutions from their encoded form (a point in the search space $\mathbb{X}$ that is processed by search operators) to their decoded, human-understandable form (a point in the solution space $\mathbb{Y}$ that can be processed by the objective function f), see Definition 9.3.

$\mathcal{I}$ the problem instance data, see Definition 3.1.

$\mathrm{lb}(f(x))$ the lower bound of a function f, i.e., $\mathrm{lb}(f) \leq f(x) \forall x$, see Definition 8.1.

M in a JSSP instance, the machine on which the operation j of job i must be executed is $M_{i,j}$.

m the number of machines in a JSSP instance, see Section 3.3.1.

$\mathrm{move}(x)$ A search operator $\mathrm{move} : \mathfrak{R} \times \mathbb{X}^k \mapsto \mathbb{X}$ with $k \in \mathbb{N}_0$ takes zero or more points from the search space $\mathbb{X}$ together with a source $\mathfrak{R}$ of randomness as input and produces one new element of $\mathbb{X}$ as output, see Definition 10.1.

n the number of jobs in a JSSP instance, see Section 3.3.1.

$\mathbb{N}_0$ the set of the natural numbers *including* 0, i.e., 0, 1, 2, 3, and so on.

$\mathbb{N}_1$ the set of the natural numbers *excluding* 0, i.e., 1, 2, 3, 4, and so on.

$\mathcal{NP}$-hard Algorithms that guarantee to find the optimal solutions of $\mathcal{NP}$-hard problems need a runtime that is exponential in the problem scale in the worst case..

$\mathfrak{R}$ a source of random number numbers; in a real algorithm implementation, this is a pseudo-random number generator, e.g., a `np.random.Generator`.

$\mathbb{R}$ the set of the real numbers.

$\mathrm{shouldTerminate}$ the termination criterion that turns `True` when the optimization process can stop and is `False` until then, see Definition 11.1.

T in a JSSP instance, the time which the operation j of job i needs is $T_{i,j}$.

$\mathbb{X}$ the search space, i.e., the candidate solutions of an optimization problem, but in an encoded form that is easier to be processed with search operators, see Definition 9.1.

x an element of the search space $\mathbb{X}$, see Definition 9.2.

$\mathbb{Y}$ the solution space, i.e., the set of possible solutions to an optimization problem, see Definition 4.2.

y a candidate solution to an optimization problem, i.e., an element of the solution space $\mathbb{Y}$, see Definition 4.1.

z an objective value returned by the objective function f, see Definition 6.2.

z_g a goal objective value z_g is a pre-defined limit at which the optimization process can stop, i.e., if a solution y with $f(y) \leq z_g$ is discovered, the optimization process can terminate and $\mathrm{shouldTerminate}$ becomes `True`, see Section 11.1..

Acronyms

BPP	Bin Packing Problem
FSSP	Flow Shop Scheduling Problem
JSSP	Job Shop Scheduling Problem
QAP	Quadratic Assignment Problem
TSP	Traveling Salesperson Problem

Bibliography

[1] Scott Aaronson. "The Limits of Quantum Computers". *Scientific American* 298(3):62–69, Mar. 2008. ISSN: 1946-7087. doi:10.1038/scientificamerican0308-62. URL: http://www.cs.virginia.edu/~robins/The_Limits_of_Quantum_Computers.pdf (cit. on p. 6).

[2] Tamer F. Abdelmaguid. "Representations in Genetic Algorithm for the Job Shop Scheduling Problem: A Computational Study". *Journal of Software Engineering and Applications (JSEA)* 3(12):1155–1162, Dec. 2010. ISSN: 1945-3116. doi:10.4236/jsea.2010.312135. URL: http://www.scirp.org/journal/paperinformation.aspx?paperid=3561 (visited on 2023-12-18) (cit. on p. 38).

[3] Joseph Adams, Egon Balas, and Daniel Zawack. "The Shifting Bottleneck Procedure for Job Shop Scheduling". *Management Science* 34(3):391–401, Mar. 1988. ISSN: 0025-1909. doi:10.1287/mnsc.34.3.391 (cit. on p. 12).

[4] Ali Allahverdi, Chi-To Daniel Ng, Edwin Tai Chiu Cheng, and Mikhail Y. Kovalyov. "A Survey of Scheduling Problems with Setup Times or Costs". *European Journal of Operational Research (EJOR)* 187(3):985–1032, June 2008. ISSN: 0377-2217. doi:10.1016/j.ejor.2006.06.060 (cit. on p. 42).

[5] David Lee Applegate, Robert E. Bixby, Vašek Chvátal, and William John Cook. *The Traveling Salesman Problem: A Computational Study*. 2nd ed. Vol. 17 of Princeton Series in Applied Mathematics. Princeton, NJ, USA: Princeton University Press, 2007. ISBN: 0-691-12993-2 (cit. on pp. 3, 6).

[6] David Lee Applegate and William John Cook. "A Computational Study of the Job-Shop Scheduling Problem". *ORSA Journal on Computing* 3(2):149–156, May 1991. ISSN: 0899-1499. doi:10.1287/ijoc.3.2.149. The JSSP instances used were generated in Bonn, Germany in 1986 (cit. on pp. 12, 13).

[7] John Edward Beasley. "OR-Library: Distributing Test Problems by Electronic Mail". *The Journal of the Operational Research Society (JORS)* 41:1069–1072, Nov. 1990. ISSN: 0160-5682. doi:10.1057/jors.1990.166 (cit. on p. 12).

[8] Martin Beckmann and Tjalling Charles Koopmans. "Assignment Problems and the Location of Economic Activities". *Econometrica* 25(1):53–76, Jan. 1957. ISSN: 0012-9682 (cit. on p. 3).

[9] Christian Bierwirth. "A Generalized Permutation Approach to Job Shop Scheduling with Genetic Algorithms". *Operations-Research-Spektrum (OR Spectrum)* 17(2–3):87–92, June 1995. ISSN: 0171-6468. doi:10.1007/BF01719250. URL: https://www.researchgate.net/publication/240263036 (visited on 2023-12-18) (cit. on pp. 41, 42).

[10] Christian Bierwirth, Dirk C. Mattfeld, and Herbert Kopfer. "On Permutation Representations for Scheduling Problems". In: *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature (PPSN IV), Sept. 22–24, 1996, Berlin, Germany*. Ed. by Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel. Vol. 1141/1996 of Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer-Verlag GmbH, 1996, pp. 310–318. ISSN: 0302-9743. ISBN: 3-540-61723-X. doi:10.1007/3-540-61723-X_995. URL: https://www.researchgate.net/publication/2753293 (visited on 2023-12-18) (cit. on p. 41).

[11] Jacek Błażewicz, Wolfgang Domschke, and Erwin Pesch. "The Job Shop Scheduling Problem: Conventional and New Solution Techniques". *European Journal of Operational Research (EJOR)* 93(1):1–33, Aug. 1996. ISSN: 0377-2217. doi:10.1016/0377-2217(95)00362-2 (cit. on pp. 5, 10).

[12] Alexander M. Bronstein and Michael M. Bronstein. "Numerical Optimization". In: *Project TOSCA – Tools for Non-Rigid Shape Comparison and Analysis*. Israel, Haifa: Technion – Israel Institute of Technology, Computer Science Department, Jan. 25, 2009. URL: https://tosca.cs.technion.ac.il/book/slides/Stanford09_optimization.ppt (visited on 2023-12-16). Slides related to the book *Numerical Geometry of Non-Rigid Shapes* [13] (cit. on pp. 29, 30).

[13] Alexander M. Bronstein, Michael M. Bronstein, and Ron Kimmel. *Numerical Geometry of Non-Rigid Shapes*. Monographs in Computer Science. New York, NY, USA: Springer, Sept. 2008. ISSN: 0172-603X. ISBN: 978-0-387-73300-5. doi:10.1007/978-0-387-73301-2 (cit. on p. 56).

[14] Rainer E. Burkard, Eranda Çela, Panos Miltiades Pardalos, and Leonidas S. Pitsoulis. "The Quadratic Assignment Problem". In: *Handbook of Combinatorial Optimization*. Ed. by Panos Miltiades Pardalos, Ding-Zhu Du, and Ronald Lewis Graham. 1st ed. Boston, MA, USA: Springer, 1998, pp. 1713–1809. ISBN: 978-1-4613-7987-4. doi:10.1007/978-1-4613-0303-9_27. URL: https://www.researchgate.net/publication/2762357 (visited on 2023-12-05). See also Bericht Nr. 126 – Mai 1998, Spezialforschungsbereich F 003: Optimierung und Kontrolle, Projektbereich Diskrete Optimierung of the Karl-Franzens-Universität Graz and the Technische Universität Graz (cit. on p. 3).

[15] Bo Chen, Chris N. Potts, and Gerhard J. Woeginger. "A Review of Machine Scheduling: Complexity, Algorithms and Approximability". In: *Handbook of Combinatorial Optimization*. Ed. by Ding-Zhu Du and Panos M. Pardalos. 1st ed. Boston, MA, USA: Springer, 1998, pp. 1493–1641. ISBN: 978-1-4613-7987-4. doi:10.1007/978-1-4613-0303-9_25. See also pages 21–169 in volume 3/3 by Norwell, MA, USA: Kluwer Academic Publishers (cit. on pp. 6, 30, 49).

[16] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. "A Tutorial Survey of Job-Shop Scheduling Problems using Genetic Algorithms – I. Representation". *Computers & Industrial Engineering* 30(4):983–997, Sept. 1996. ISSN: 0360-8352. doi:10.1016/0360-8352(96)00047-2 (cit. on p. 38).

[17] Philippe Chrétienne, Edward G. Coffman, Jan Karel Lenstra, and Zhen Liu, eds. *Scheduling Theory and Its Applications*. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., Sept. 1995. ISBN: 978-0-471-94059-3 (cit. on p. 5).

[18] Stephen Arthur Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC'71), May 3–5, 1971, Shaker Heights, OH, USA*. Ed. by Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman. New York, NY, USA: Association for Computing Machinery (ACM), 1971, pp. 151–158. ISBN: 978-1-4503-7464-4. doi:10.1145/800157.805047 (cit. on p. 6).

[19] William John Cook. *World TSP*. Waterloo, ON, Canada: University of Waterloo, Department of Combinatorics and Optimization, Feb. 15, 2021. URL: https://www.math.uwaterloo.ca/tsp/world/ (visited on 2023-12-07) (cit. on p. 7).

[20] William John Cook, Daniel G. Espinoza, and Marcos Goycoolea. "Computing with Domino-Parity Inequalities for the Traveling Salesman Problem (TSP)". *INFORMS Journal on Computing* 19(3):356–365, Aug. 2007. ISSN: 1091-9856. doi:10.1287/ijoc.1060.0204. URL: http://www.dii.uchile.cl/~daespino/PApers/DP_paper.pdf (visited on 2023-12-07) (cit. on p. 7).

[21] Jim Davis, Thomas F. Edgar, James Porter, John Bernaden, and Michael Sarli. "Smart Manufacturing, Manufacturing Intelligence and Demand-Dynamic Performance". *Computers & Chemical Engineering* 47:145–156, Dec. 2012. ISSN: 0098-1354. doi:10.1016/j.compchemeng.2012.06.037 (cit. on p. 5).

[22] Maxence Delorme, Manuel Iori, and Silvano Martello. "Bin Packing and Cutting Stock Problems: Mathematical Models and Exact Algorithms". *European Journal of Operational Research (EJOR)* 255(1):1–20, Nov. 2016. ISSN: 0377-2217. doi:10.1016/j.ejor.2016.04.030 (cit. on p. 4).

[23] Ebru Demirkol, Sanjay V. Mehta, and Reha Uzsoy. "Benchmarks for Shop Scheduling Problems". *European Journal of Operational Research (EJOR)* 109(1):137–141, Aug. 1998. ISSN: 0377-2217. doi:10.1016/S0377-2217(97)00019-2 (cit. on pp. 12, 13).

[24] Harald Dyckhoff and Ute Finke. *Cutting and Packing in Production and Distribution: A Typology and Bibliography*. Contributions to Management Science. Heidelberg, Baden-Württemberg, Germany: Physica-Verlag, 1992. ISSN: 1431-1941. ISBN: 978-3-642-63487-1. doi:10.1007/978-3-642-58165-6 (cit. on pp. 3, 4).

[25] Henry Fisher and Gerald L. Thompson. “Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules”. In: *Industrial Scheduling*. Ed. by John F. Muth and Gerald L. Thompson. Englewood Cliffs, NJ, USA: Prentice-Hall, 1963. Chap. 3.2, pp. 225–251 (cit. on p. 12).

[26] Michael R. Garey and David Stifler Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman and Company, 1979. ISBN: 0-7167-1045-5 (cit. on p. 7).

[27] Michael R. Garey, David Stifler Johnson, and Ravi Sethi. “The Complexity of Flowshop and Jobshop Scheduling”. *Mathematics of Operations Research (MOR)* 1(2):117–129, May 1976. ISSN: 0364-765X. doi:10.1287/moor.1.2.117 (cit. on p. 42).

[28] Mitsuo Gen, Yasuhiro Tsujimura, and Erika Kubota. “Solving Job-Shop Scheduling Problems by Genetic Algorithm”. In: *Humans, Information and Technology: Proceedings of the 1994 IEEE International Conference on Systems, Man and Cybernetics (SMC), Oct. 2–5, 1994, San Antonio, TX, USA*. Vol. 2. Piscataway, NJ, USA: IEEE, 1994. ISBN: 0-7803-2129-4. doi:10.1109/ICSMC.1994.400072 (cit. on p. 41).

[29] Anis Gharbi and Mohamed Labidi. “Extending the Single Machine-Based Relaxation Scheme for the Job Shop Scheduling Problem”. *Electronic Notes in Discrete Mathematics* 36:1057–1064, Aug. 2010. ISSN: 1571-0653. doi:10.1016/j.endm.2010.05.134 (cit. on p. 35).

[30] Seyyedeh Newsha Ghoreishi, Anders Clausen, and Bo Nørregaard Jørgensen. “Termination Criteria in Evolutionary Algorithms: A Survey”. In: *Proceedings of the 9th International Joint Conference on Computational Intelligence (IJCCI’17), Nov. 1-3, 2017, Funchal, Madeira, Portugal*. Ed. by Christophe Sabourin, Juan Julián Merelo Guervós, Una-May O’Reilly, Kurosh Madani, and Kevin Warwick. Setúbal, Portugal: SciTePress, Science and Technology Publications, Lda, 2017, pp. 373–384. ISSN: 2184-3236. ISBN: 978-989-758-274-5. doi:10.5220/0006577903730384. URL: https://pdfs.semanticscholar.org/05ca/33f534824b924d415623dce9eec420b33baf.pdf (cit. on p. 48).

[31] Ronald Lewis Graham, Eugene Leighton Lawler, Jan Karel Lenstra, and Alexander Hendrik George Rinnooy Kan. “Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey”. In: *Discrete Optimization II: Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver, Aug. 1-31, 1977, B.C. Canada*. Ed. by Peter L. Hammer, E. L. Johnson, and Bernhard H. Korte. Vol. 5 of Annals of Discrete Mathematics. Amsterdam, The Netherlands: Elsevier, 1979, pp. 287–326. ISSN: 0167-5060. doi:10.1016/S0167-5060(08)70356-X. URL: https://ir.cwi.nl/pub/18052/18052A.pdf (visited on 2023-12-06) (cit. on pp. 5, 10, 11).

[32] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. “Optimal Control of Plotting and Drilling Machines: A Case Study”. *Zeitschrift für Operations Research (ZOR) – Methods and Models of Operations Research* 35(1):61–84, Jan. 1991. ISSN: 0340-9422. doi:10.1007/BF01415960 (cit. on p. 3).

[33] Gregory Z. Gutin and Abraham P. Punnen, eds. *The Traveling Salesman Problem and its Variations*. Vol. 12 of Combinatorial Optimization (COOP). New York, NY, USA: Springer, 2002. ISBN: 0-306-48213-4. doi:10.1007/b101971 (cit. on pp. 3, 6, 7).

[34] Keld Helsgaun. “General k-opt Submoves for the Lin-Kernighan TSP Heuristic”. *Mathematical Programming Computation (MPC): A Publication of the Mathematical Optimization Society* 1(2-3):119–163, Oct. 2009. ISSN: 1867-2949. doi:10.1007/s12532-009-0004-6 (cit. on p. 7).

[35] André Henning. “Praktische Job-Shop Scheduling-Probleme”. PhD thesis. Jena, Thüringen, Germany: Friedrich-Schiller-Universität Jena, Fakultät für Mathematik und Informatik, Aug. 2002. URL: https://www.db-thueringen.de/servlets/MCRFileNodeServlet/dbt_derivate_00001373/Dissertation.pdf (visited on 2023-12-07) (cit. on pp. 12, 13).

[36] Mario Hermann, Tobias Pentek, and Boris Otto. “Design Principles for Industrie 4.0 Scenarios”. In: *Proceedings of the 2016 49th Hawaii International Conference on System Sciences (HICSS), Jan. 5–8, 2016, Koloa, HI, USA*. Ed. by Tung X. Bui and Ralph H. Sprague Jr. Washington, DC, USA: IEEE Computer Society, 2016, pp. 3928–3937. ISBN: 978-0-7695-5670-3. doi:10.1109/HICSS.2016.488 (cit. on p. 5).

[37] Masanobu Iida, Tsuyoshi Matsumura, Koji Nakatani, Takashi Fukuda, and Tatsuo Maeda. “Effective Nose Shape for Reducing Tunnel Sonic Boom”. *Railway Technical Research Institute, Quarterly Reports (QR)* 38(4):206–211, Nov. 1997. ISSN: 0033-9008 (cit. on p. 37).

[38] Anant Singh Jain and Sheik Meeran. "Deterministic Job-Shop Scheduling: Past, Present and Future". *European Journal of Operational Research (EJOR)* 113(2):390–434, Mar. 1999. ISSN: 0377-2217. doi:10.1016/S0377-2217(98)00113-1 (cit. on pp. 13, 35).

[39] Klaus Jansen, Monaldo Mastrolilli, and Roberto Solis-Oba. "Approximation Schemes for Job Shop Scheduling Problems with Controllable Processing Times". *European Journal of Operational Research (EJOR)* 167(2):297–319, Dec. 2005. ISSN: 0377-2217. doi:10.1016/j.ejor.2004.03.025. URL: http://people.idsia.ch/~monaldo/papers/EJOR-varJsp-05.pdf (visited on 2023-12-20) (cit. on p. 49).

[40] Selmer Martin Johnson. "Optimal Two- and Three-Stage Production Schedules with Setup Times Included". *Naval Research Logistics Quarterly* I(1):61–68, Mar. 1954. doi:10.1002/nav.3800010110. URL: https://www.rand.org/content/dam/rand/pubs/papers/2008/P402.pdf (visited on 2023-12-06) (cit. on pp. 5, 30).

[41] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held Mar. 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. The IBM Research Symposia Series. Boston, MA, USA: Springer, 1972, pp. 85–103. ISBN: 978-1-4684-2003-6. doi:10.1007/978-1-4684-2001-2_9 (cit. on p. 6).

[42] Graham Kendall, Ruibin Bai, Jacek Błażewicz, Patrick De Causmaecker, Michel Gendreau, Robert I. John, Jiawei Li, Barry McCollum, Erwin Pesch, Rong Qu, Nasser Sabar, Greet Vanden Berghe, and Angelina Yee. "Good Laboratory Practice for Optimization Research". *The Journal of the Operational Research Society (JORS)* 67(4):676–689, 2016. ISSN: 0160-5682. doi:10.1057/jors.2015.77. URL: http://www.graham-kendall.com/papers/ketal2016.pdf (visited on 2023-12-16) (cit. on p. 29).

[43] Katsuhiro Kikuchi, Masanobu Iida, and Takashi Fukuda. "Optimization of Train Nose Shape for Reducing Micro-Pressure Wave Radiated from Tunnel Exit". *Journal of Low Frequency Noise, Vibration and Active Control* 30(1):1–19, Mar. 2011. ISSN: 1461-3484. doi:10.1260/0263-0923.30.1.1 (cit. on p. 37).

[44] Robert Klein. *Scheduling of Resource-Constrained Projects*. Vol. 10 of Operations Research/-Computer Science Interfaces Series. New York, NY, USA: Springer, 2000. ISSN: 1387-666X. ISBN: 978-0-7923-8637-7. doi:10.1007/978-1-4615-4629-0 (cit. on p. 17).

[45] Joshua Damian Knowles and Richard A. Watson. "On the Utility of Redundant Encodings in Mutation-Based Evolutionary Search". In: *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature (PPSN VII), Sept. 7–11, 2002, Granada, Spain*. Ed. by Juan Julián Merelo Guervós, Panagiotis Adamidis, Hans-Georg Beyer, Hans-Paul Schwefel, and José-Luis Fernández-Villacañas. Vol. 2439 of Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer-Verlag GmbH, 2002, pp. 88–98. ISSN: 0302-9743. ISBN: 978-3-540-44139-7. doi:10.1007/3-540-45712-7_9. URL: https://www.researchgate.net/publication/220701781 (visited on 2023-12-18) (cit. on p. 37).

[46] Eugene Leighton Lawler. "Recent Results in the Theory of Machine Scheduling". In: *Math Programming: The State of the Art*. Ed. by Achim Bachem, Bernhard H. Korte, and Martin Grötschel. Bonn, North Rhine-Westphalia, Germany: Springer-Verlag GmbH, 1982. Chap. 8, pp. 202–234. ISBN: 978-3-642-68876-8. doi:10.1007/978-3-642-68874-4_9 (cit. on pp. 5, 10, 11).

[47] Eugene Leighton Lawler, Jan Karel Lenstra, Alexander Hendrik George Rinnooy Kan, and David B. Shmoys. "Sequencing and Scheduling: Algorithms and Complexity". In: *Production Planning and Inventory*. Ed. by Stephen C. Graves, Alexander Hendrik George Rinnooy Kan, and Paul H. Zipkin. Vol. IV of Handbooks of Operations Research and Management Science. Amsterdam, The Netherlands: Elsevier, 1993. Chap. 9, pp. 445–522. ISSN: 0927-0507. ISBN: 978-0-444-87472-6. doi:10.1016/S0927-0507(05)80189-6. URL: http://alexandria.tue.nl/repository/books/339776.pdf (visited on 2023-12-06) (cit. on pp. 5, 6, 10, 11, 30, 49).

[48] Eugene Leighton Lawler, Jan Karel Lenstra, Alexander Hendrik George Rinnooy Kan, and David B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Estimation, Simulation, and Control – Wiley-Interscience Series in Discrete Mathematics and Optimization. Chichester, West Sussex, England, UK: Wiley Interscience, Sept. 1985. ISSN: 0277-2698. ISBN: 0-471-90413-9 (cit. on pp. 3, 6).

[49] Stephen R. Lawrence. "Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement)". PhD thesis. Pittsburgh, PA, USA: Carnegie-Mellon University, Graduate School of Industrial Administration (GSIA), 1984 (cit. on pp. 12, 13).

[50] Andrea Lodi, Silvano Martello, and Michele Monaci. "Two-Dimensional Packing Problems: A Survey". *European Journal of Operational Research (EJOR)* 141(2):241–252, Sept. 2002. ISSN: 0377-2217. doi:10.1016/S0377-2217(02)00123-6 (cit. on p. 4).

[51] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley Series in Discrete Mathematics and Optimization. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 1990. ISBN: 978-0471924203. URL: http://www.or.deis.unibo.it/knapsack.html (visited on 2023-12-07) (cit. on p. 4).

[52] Monaldo Mastrolilli and Ola Svensson. "Hardness of Approximating Flow and Job Shop Scheduling Problems". *Journal of the Association for Computing Machinery (JACM)* 58(5):20:1–20:32, Oct. 2011. ISSN: 0004-5411. doi:10.1145/2027216.2027218. URL: http://theory.epfl.ch/osven/Ola%20Svensson_publications/JACM11.pdf (visited on 2023-12-20) (cit. on p. 49).

[53] Zbigniew Michalewicz, Leonardo Arantes, and Matthew Michalewicz. *The Rise of Artificial Intelligence: Real-world Applications for Revenue and Margin Growth*. Ormond, VIC, Australia: Hybrid Publishers, 2021. ISBN: 978-1925736625 (cit. on p. 1).

[54] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. 2nd ed. Berlin/Heidelberg, Germany: Springer-Verlag GmbH, 2004. ISBN: 978-3-540-22494-5. doi:10.1007/978-3-662-07807-5 (cit. on p. 7).

[55] Yuichi Nagata and Shigenobu Kobayashi. "A Powerful Genetic Algorithm using Edge Assembly Crossover for the Traveling Salesman Problem". *INFORMS Journal on Computing* 25(2):346–363, Spr. 2013. ISSN: 1091-9856. doi:10.1287/ijoc.1120.0506 (cit. on p. 7).

[56] Panos Miltiades Pardalos, Ding-Zhu Du, and Ronald Lewis Graham, eds. *Handbook of Combinatorial Optimization*. 1st ed. Boston, MA, USA: Springer, 1998. ISBN: 978-1-4613-7987-4.

[57] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. 5th ed. Cham, Switzerland: Springer International Publishing AG, 2016. ISBN: 978-3-319-26578-0. doi:10.1007/978-3-319-26580-3 (cit. on p. 4).

[58] Chris N. Potts and Vitaly A. Strusevich. "Fifty Years of Scheduling: A Survey of Milestones". *The Journal of the Operational Research Society (JORS)* 60(sup1):S41–S68, 2009. ISSN: 0160-5682. doi:10.1057/jors.2009.2. URL: http://eprints.soton.ac.uk/145495/1/Scheduling50YearsE-Print.pdf (visited on 2023-12-06). Special Issue: Milestones in OR (cit. on p. 4).

[59] Miha Ravber, Shih-Hsi Liu, Marjan Mernik, and Matej Črepinšek. *Applied Soft Computing (ASOC)* 128(109478), Oct. 2022. ISSN: 1568-4946. doi:10.1016/J.ASOC.2022.109478. URL: https://www.researchgate.net/publication/362640980 (cit. on p. 48).

[60] Sartaj Sahni and Teofilo Francisco Gonzalez Arce. "P-Complete Approximation Problems". *Journal of the Association for Computing Machinery (JACM)* 23(3):555–565, July 1976. ISSN: 0004-5411. doi:10.1145/321958.321975. URL: https://sites.cs.ucsb.edu/~teo/papers/JACM-PCom.pdf (visited on 2023-12-05) (cit. on p. 3).

[61] Guntram Scheithauer. *Introduction to Cutting and Packing Optimization: Problems, Modeling Approaches, Solution Methods*. Vol. 263 of International Series in Operations Research & Management Science (ISOR). Cham, Switzerland: Springer International Publishing AG, 2018. ISSN: 0884-8289. ISBN: 978-3-319-64402-8. doi:10.1007/978-3-319-64403-5 (cit. on p. 3).

[62] Guoyong Shi, Hitoshi Iima, and Nobuo Sannomiya. “New Encoding Scheme for Solving Job Shop Problems by Genetic Algorithm”. In: *Proceedings of the 35th IEEE Conference on Decision and Control (CDC'96), Dec. 11–13, 1996, Kobe, Japan*. Vol. 4. Piscataway, NJ, USA: IEEE, 1997, pp. 4395–4400. ISBN: 0-7803-3590-2. doi:10.1109/CDC.1996.577484. URL: https://www.researchgate.net/publication/224238934 (visited on 2023-12-18) (cit. on pp. 41, 42).

[63] Oleg V. Shylo. *Job Shop Scheduling (Personal Homepage)*. Aug. 2019. URL: http://optimizizer.com/jobshop.php (visited on 2023-12-07) (cit. on p. 12).

[64] Leon Steinberg. “The Backboard Wiring Problem: A Placement Algorithm”. *SIAM Review* 3(1):37–50, Jan. 1961. ISSN: 0036-1445. doi:10.1137/1003003 (cit. on p. 3).

[65] Robert H. Storer, S. David Wu, and Renzo Vaccari. “New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling”. *Management Science* 38(10):1495–1509, 1992. ISSN: 0025-1909. doi:10.1287/mnsc.38.10.1495 (cit. on pp. 12, 13).

[66] Éric D. Taillard. “Benchmarks for Basic Scheduling Problems”. *European Journal of Operational Research (EJOR)* 64(2):278–285, Jan. 1993. ISSN: 0377-2217. doi:10.1016/0377-2217(93)90182-M. URL: http://mistic.heig-vd.ch/taillard/articles.dir/Taillard1993EJOR.pdf (visited on 2023-12-06) (cit. on pp. 5, 10–13, 33, 34, 42).

[67] Éric D. Taillard. *Scheduling Instances*. Yverdon, Switzerland: University of Applied Sciences of Western Switzerland, 1993. URL: http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html (visited on 2023-12-07) (cit. on p. 12).

[68] Rob J. M. Vaessens, Emile H. L. Aarts, and Jan Karel Lenstra. “Job Shop Scheduling by Local Search”. *INFORMS Journal on Computing* 8(3):302–317, Sum. 1996. ISSN: 1091-9856. doi:10.1287/ijoc.8.3.302 (cit. on p. 35).

[69] Jelke Jeroen van Hoorn. “Dynamic Programming for Routing and Scheduling: Optimizing Sequences of Decisions”. PhD thesis. Amsterdam, The Netherlands: Vrije Universiteit Amsterdam, June 2016. ISBN: 978-94-6332-008-5. URL: http://jobshop.jjvh.nl/dissertation (visited on 2023-12-07) (cit. on p. 19).

[70] Jelke Jeroen van Hoorn. *Job Shop Instances and Solutions*. 2015. URL: http://jobshop.jjvh.nl (visited on 2023-12-07) (cit. on p. 12).

[71] Jelke Jeroen van Hoorn. “The Current State of Bounds on Benchmark Instances of the Job-Shop Scheduling Problem”. *Journal of Scheduling* 21(1):127–128, Feb. 2018. ISSN: 1094-6136. doi:10.1007/s10951-017-0547-8 (cit. on p. 12).

[72] Petr Vilím, Philippe Laborie, and Paul Shaw. “Failure-Directed Search for Constraint-Based Scheduling”. In: *Proceedings of 12th International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems (CPAIOR'2015), May 18-22, 2015, Barcelona, Spain*. Ed. by Laurent Michel. Vol. 9075 of Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer-Verlag GmbH, 2015, pp. 437–453. ISSN: 0302-9743. ISBN: 978-3-319-18007-6. doi:10.1007/978-3-319-18008-3_30. URL: https://www.researchgate.net/publication/280223626 (visited on 2023-12-17) (cit. on p. 35).

[73] Thomas Weise. *Global Optimization Algorithms – Theory and Application*. Hefei, Anhui, China: Institute of Applied Optimization, Hefei University, 2009. URL: http://iao.hfuu.edu.cn/images/publications/W2009GOEB.pdf (visited on 2023-12-05) (cit. on pp. iii, 7).

[74] Thomas Weise. *jsspInstancesAndResults: Results, Data, and Instances of the Job Shop Scheduling Problem*. Hefei, Anhui, China: Institute of Applied Optimization, Hefei University, 2019. URL: http://github.com/thomasWeise/jsspInstancesAndResults (visited on 2023-12-07). A GitHub repository with the common benchmark instances for the Job Shop Scheduling Problem as well as results from the literature, both in form of CSV files and R program code to access them. (Cit. on pp. 12, 35).

[75] Thomas Weise, Alexander Podlich, and Christian Gorldt. “Solving Real-World Vehicle Routing Problems with Evolutionary Algorithms”. In: *Natural Intelligence for Scheduling, Planning and Packing Problems*. Ed. by Raymond Chiong and Sandeep Dhakal. Vol. 250 of Studies in Computational Intelligence (SCI). Berlin/Heidelberg, Germany: Springer-Verlag GmbH, Oct. 2009. Chap. 2, pp. 29–53. ISSN: 1860-949X. ISBN: 978-3-642-04038-2. doi:10.1007/978-3-642-04039-9_2. URL: http://iao.hfuu.edu.cn/images/publications/WPG2009SRWVRPWEA.pdf (visited on 2023-12-05) (cit. on p. 3).

[76] Thomas Weise, Alexander Podlich, Kai Reinhard, Christian Gorldt, and Kurt Geihs. "Evolutionary Freight Transportation Planning". In: *Applications of Evolutionary Computing – Proceedings of EvoWorkshops 2009: EvoCOMNET, EvoENVIRONMENT, EvoFIN, EvoGAMES, EvoHOT, EvoIASP, EvoINTERACTION, EvoMUSART, EvoNUM, EvoSTOC, EvoTRANSLOG, Apr. 15–17, 2009, Tübingen, Baden-Württemberg, Germany*. Ed. by Mario Giacobini, Penousal Machado, Anthony Brabazon, Jon McCormack, Stefano Cagnoni, Michael O'Neill, Gianni A. Di Caro, Ferrante Neri, Anikó Ekárt, Mike Preuß, Anna Isabel Esparcia-Alcázar, Franz Rothlauf, Muddassar Farooq, Ernesto Tarantino, Andreas Fink, and Shengxiang Yang. Vol. 5484/2009 of Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer-Verlag GmbH, 2009, pp. 768–777. ISSN: 0302-9743. doi:10.1007/978-3-642-01129-0_87. URL: http://iao.hfuu.edu.cn/images/publications/WPRGG2009EFTP.pdf (visited on 2023-12-07) (cit. on p. 3).

[77] Thomas Weise, Yuezhong Wu, Weichen Liu, and Raymond Chiong. "Implementation Issues in Optimization Algorithms: Do They Matter?" *Journal of Experimental & Theoretical Artificial Intelligence (JETAI)* 31(4):533–554, 2019. ISSN: 0952-813X. doi:10.1080/0952813X.2019.1574908 (cit. on p. 7).

[78] Frank Werner. "Genetic Algorithms for Shop Scheduling Problems: A Survey". In: *Heuristics: Theory and Applications*. Ed. by Patrick Siarry. Hauppauge, NY, USA: Nova Science Publishers, 2013. Chap. 8, pp. 161–222. ISBN: 1624176372. URL: https://www.researchgate.net/publication/236267741 (visited on 2023-12-18) (cit. on pp. 38, 42).

[79] L. Darrell Whitley. "Blind No More: Deterministic Partition Crossover and Deterministic Improving Moves". In: *Companion Material Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'16), July 20-24, 2016, Denver, CO, USA*. Ed. by Tobias Friedrich, Frank Neumann, and Andrew M. Sutton. New York, NY, USA: Association for Computing Machinery (ACM), 2016, pp. 515–532. ISBN: 978-1-4503-4323-7. doi:10.1145/2908961.2926987 (cit. on p. 7).

[80] David Paul Williamson, Leslie A. Hall, J. A. (Han) Hoogeveen, Cor A. J. Hurkens, Jan Karel Lenstra, Sergey Vasil'evich Sevast'janov, and David B. Shmoys. "Short Shop Schedules". *Operations Research* 45(2):288–294, Mar.–Apr. 1997. ISSN: 1526-5463. doi:10.1287/opre.45.2.288. URL: https://www.researchgate.net/publication/2254626 (visited on 2023-12-20) (cit. on p. 49).

[81] James M. Wilson. "Gantt Charts: A Centenary Appreciation". *European Journal of Operational Research (EJOR)* 149(2):430–437, Sept. 2003. ISSN: 0377-2217. doi:10.1016/S0377-2217(02)00769-5. URL: http://www-public.imtbs-tsp.eu/~gibson/Teaching/Teaching-ReadingMaterial/Wilson03.pdf (visited on 2023-12-15) (cit. on p. 17).

[82] Takeshi Yamada and Ryohei Nakano. "A Genetic Algorithm Applicable to Large-Scale Job-Shop Instances". In: *Proceedings of Parallel Problem Solving from Nature 2 (PPSN II), Sept. 28–30, 1992, Brussels, Belgium*. Ed. by Reinhard Männer and Bernard Manderick. New York, NY, USA: Elsevier Science Inc., 1992, pp. 281–290. ISBN: 978-0-444-89730-5. URL: https://www.researchgate.net/publication/220701684 (visited on 2023-12-07) (cit. on pp. 12, 13).

[83] Takeshi Yamada and Ryohei Nakano. "Genetic Algorithms for Job-Shop Scheduling Problems". In: *Proceedings of Modern Heuristic for Decision Support, Mar.18–19, 1997, London, England, UK*. Uxbridge, London, England, UK: UNICOM Seminars Ltd., 1997, pp. 67–81. URL: https://www.researchgate.net/publication/2804648 (visited on 2023-12-18) (cit. on p. 38).