An Introduction to Optimization Algorithms

Thomas Weise

2020-01-17

Contents

Preface

After I wrote *Global Optimization Algorithms – Theory and Applications* [163] during my time as PhD student more than ten years ago, I now want to write a more direct guide to optimization, optimization algorithms, and metaheuristics. Currently, this book is in a very early stage of development. It is work-in-progress, so expect many changes. This book is available as pdf, html, epub, and azw3.

My goal is to write an accessible and easy to read book on optimization that even undergraduate students with no background in the field should be able to understand without any problem. This book should give the reader a good intuition about how the algorithms work in practice, what things to look for when solving a problem, or how to get from a simple, working, proof-of-concept approach to an efficient solution 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 their actual results. We try to improve the algorithms step-by-step, moving from very simple approaches, which do not work well, to efficient metaheuristics. We will partially sacrifice the formal and abstract structure of [163] and introduce concepts "as they come," with the goal to increase the accessibility of the ideas.

```
@book{aitoa,
  author    = {Thomas Weise},
  title     = {An Introduction to Optimization Algorithms},
  year      = {2018--2020},
  publisher = {Institute of Applied Optimization ({IAO}),
               School of Artificial Intelligence and Big Data,
               Hefei University},
  address   = {Hefei, Anhui, China},
  url       = {http://thomasweise.github.io/aitoa/},
  edition   = {2020-01-17}
}
```

We use concrete examples and algorithm implementations written in Java. All source code is freely available in the repository *thomasWeise/aitoa-code* on GitHub. Often, we will just look at certain

portions of the code, maybe parts of a class, where we omit methods or member variables, or even just snippets from functions. Each source code listing is accompanied by a *(src)* link in the caption linking to the current full version of the file in the GitHub repository. If you discover an error in any of the examples, please file an issue.

This book is written using our automated book writing environment, which integrates GitHub, Travis CI, and docker-hub. The text of the book is actively written and available in the repository *thomasWeise/aitoa* on GitHub. There, you can also submit *issues*, such as change requests, suggestions, errors or typos, or you can inform me that something is unclear, so that I can improve the book.

repository: http://github.com/thomasWeise/aitoa
commit: 74f1ef2588a832a364469979088a32ca2bde428f
time and date: 2020-01-17 07:08:42 UTC+0000
example source repository: http://github.com/thomasWeise/aitoa-code
example source commit: 54488650fe9c55d18203aae5c2f191c5f61fc952
code for generating diagrams: http://github.com/thomasWeise/aitoaEvaluate

Prof. Dr. Thomas Weise
Institute of Applied Optimization (IAO),
Faculty of Computer Science and Technology,
Hefei University,
Hefei, Anhui, China.
Web: http://iao.hfuu.edu.cn/team/director
Email: tweise@hfuu.edu.cn, tweise@ustc.edu.cn

1 Introduction

Today, algorithms influence a bigger and bigger part in our daily life and the economy. They support us by suggesting good decisions in a variety of fields, ranging from engineering, timetabling and scheduling, product design, over travel and logistic planning to even product or movie recommendations. 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.

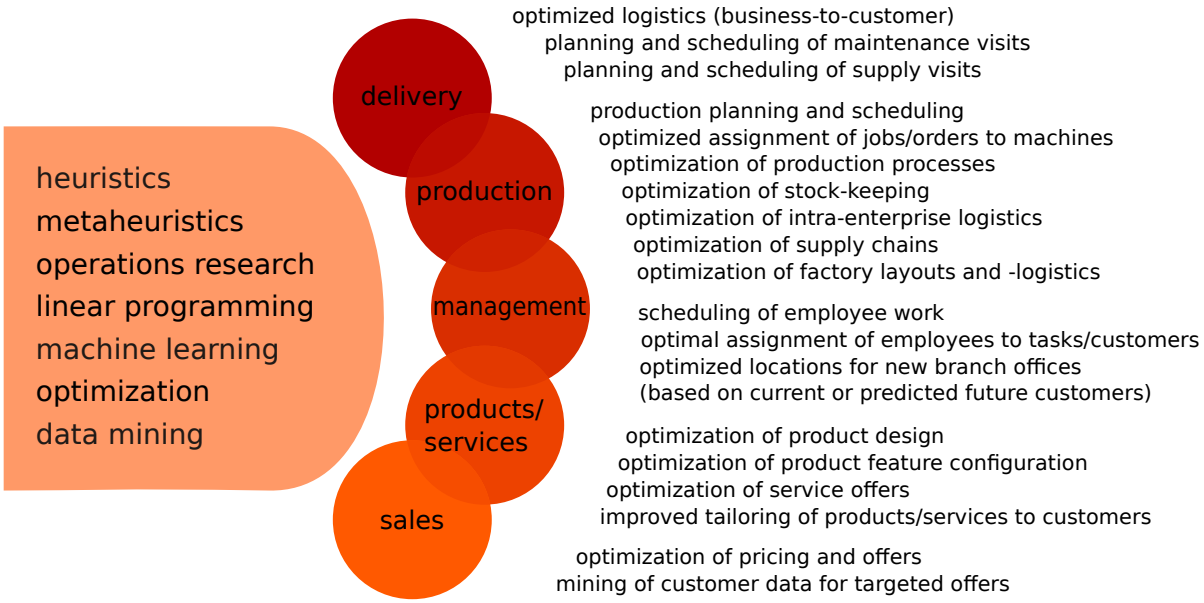

Figure 1.1: Examples for applications of optimization, computational intelligence, machine learning techniques in five fields of smart manufacturing: the production itself, the delivery of the products, the management of the production, the products and services, and the sales level.

Optimization and Operations Research provide us with algorithms that propose good solutions to such a wide range of questions. Usually, it is applied in scenarios where we can choose from many possible options. The goal is that the algorithms propose solutions which minimize (at least) one resource requirement, be it costs, energy, space, etc. If they can do this well, they also offer another important advantage: Solutions that minimize resource consumption are often not only cheaper from

an immediate economic perspective, but also better for the environment, i.e., with respect to ecological considerations.

Thus, we already know three reasons why optimization will be a key technology for the next century, which silently does its job behind the scenes:

1. Any form of intelligent production or smart manufacturing needs automated decisions and since these decisions should be *intelligent*, they can only come from a process which involves optimization in one way or another.
2. In global and local competition in all branches of industry and all service sectors those institutions who can reduce their resource consumption and costs while improving product quality and production efficiency will have the edge – and one 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,", i.e., often inherently leads to more environmentally friendly processes.

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

In this book, we will 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 get a feeling about typical use cases of optimization.

1.1.1 Example: Route Planning for a Logistics Company

One example field of application for optimization is logistics. Let us look at a typical real-world scenario from this field [169,170]: 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 then 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 to 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 1.2. The trucks it owns may have different capacities

and can carry one or two containers. Besides using trucks, which can travel freely on the map, it may also be possible to utilize trains. Trains may have vastly different capacities and follow specific schedules and arrive and depart at fixed times to/from fixed locations. For each vehicle, different costs could occur. Containers may be exchanged between vehicles at locations such as parking lots, depots, or train stations.

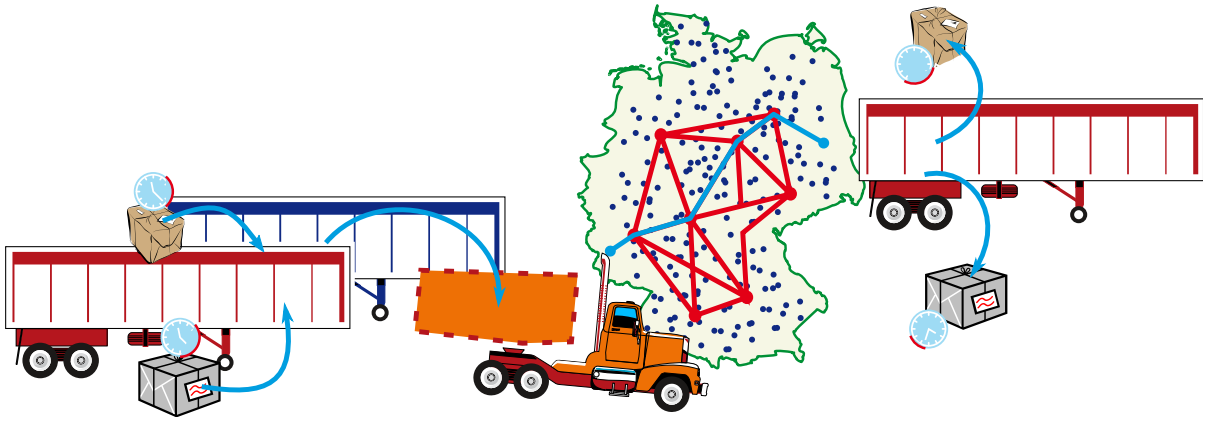

Figure 1.2: Illustrative sketch of logistics problems: Orders require us to pick up some items at source locations within certain time windows and deliver them to their destination locations, again within certain time windows. We need to decide which containers and vehicles to use and over which routes we should channel the vehicles.

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.

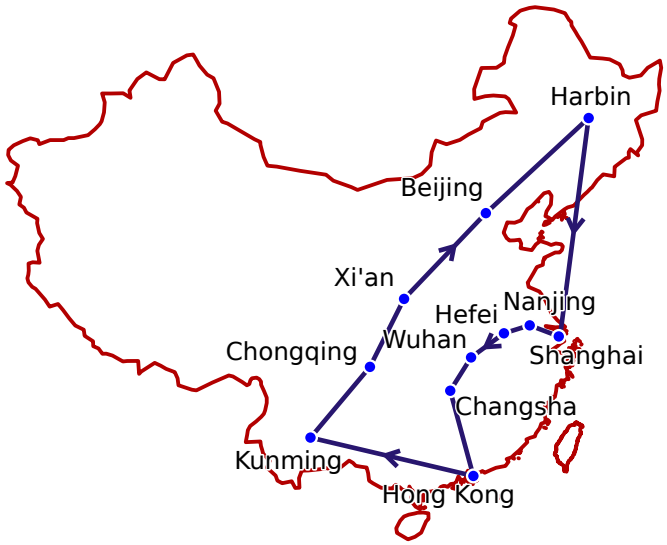

Figure 1.3: A Traveling Salesman Problem (TSP) through eleven cities in China.

Of course, there is a wide variety of possible logistics planning tasks. Besides our real-world example above, a classical task is the Traveling Salesman Problem (TSP) [9,79,109], where the goal is to find the shortest round-trip tour through$n$ cities, as sketched in Figure 1.3. 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 [76].

1.1.2 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 fridge, as sketched in Figure 1.4. 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 [56,144] aim to package sets of objects into containers as efficient 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.

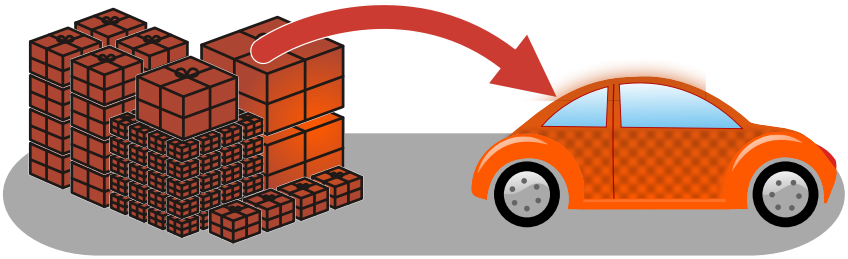

Figure 1.4: A sketch illustrating a packing problem.

Such bin packing problems exist in many variants and are very related to cutting stock problems [56]. They can be one-dimensional [50], 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.

A two-dimensional variant [112] 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).

1.1.3 Example: Job Shop Scheduling Problem

Another typical optimization task arises in manufacturing, namely the assignment ("scheduling") of tasks ("jobs") to machines and start times [133]. In the basic *Job Shop Scheduling Problem* (JSSP) [29,38,54,73,108,110], 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. So each production job has one sub-job for each machine on which it needs to be processed. We need to execute these sub-jobs in the right sequence. Of course, no machine can process more than one order at the same time. We can decide when which sub-job should begin and we are looking for the starting times that lead to the earliest completion of all jobs, i.e., the shortest makespan.

This general scenario "contains" 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. 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 easier problem where the machine order is fixed. An example for the general scenario is sketched in Figure 1.5, 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. We will introduce the JSSP in detail in Section 2.2.2.

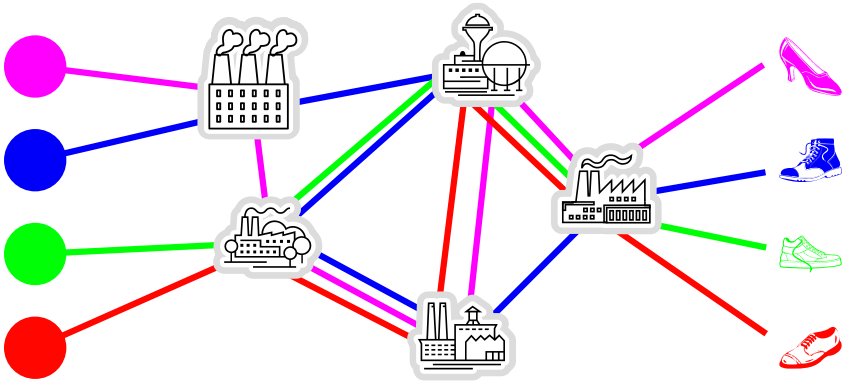

Figure 1.5: Illustrative sketch of a JSSP scenario with four jobs where four different types of shoe should be produced, which require different workshops ("machines") to perform different production steps.

The three examples we have discussed so far are, actually, quite related. They all fit into the broad area of smart manufacturing [46,87]. 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. The need for flexibility and a large degree of automation require 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?

Why should you read this book?

1.2.1 Good Solutions within Acceptable Time

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. Everyone 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 equation. They require algorithms, they require computer science.

Unfortunately, while we have learned many types of equations that can be solved easily in our mathematics classes, 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 [32,110]. Unless some unlikely breakthrough happens [39,102], there will be many such problems that we cannot solve exactly within reasonable time - and all of the example problems discussed so far are among them. (And: No, quantum computers are not the answer. Most likely, they cannot solve these problems qualitatively faster either [1].)

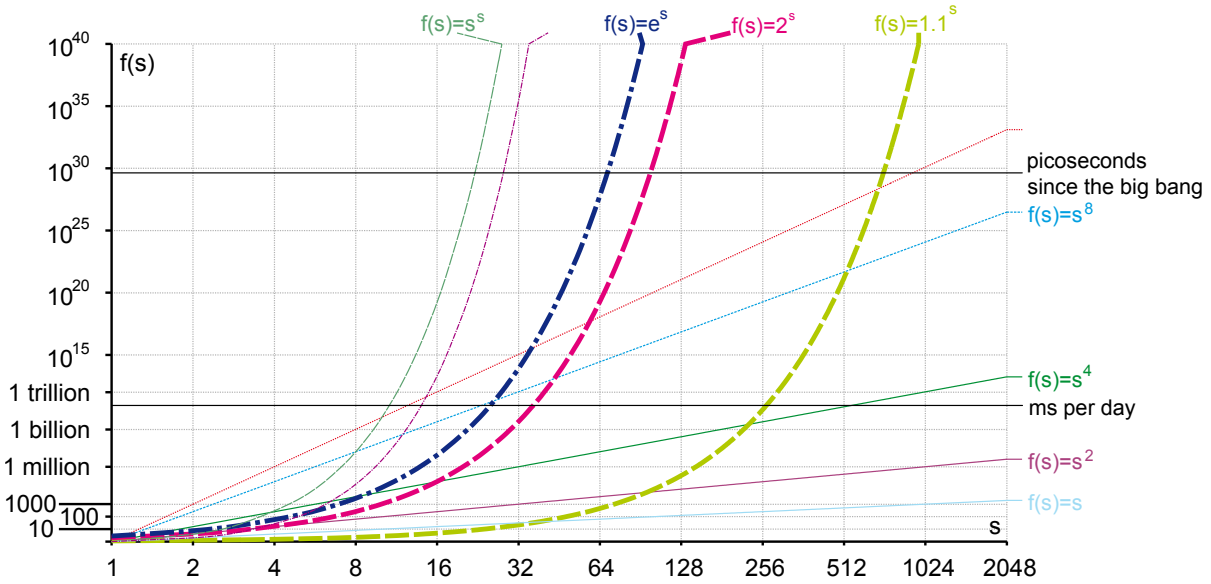

Figure 1.6: The growth of different functions in a log-log scaled plot. Exponential functions grow very fast, so that an algorithm which needs $\sim 2^s$ steps to solve an optimization problem of size s quickly becomes infeasible. (compare with Table 2.1 and Table 2.3)

Figure 1.6 illustrates that finding the solutions for problems with such exponential "time complexity" will quickly become infeasible, even for relatively small problem instances. Just throwing more computing power at the problems will not solve this fundamental issue. Our processing power is limited and parallelization can provide a linear speed-up at best. This cannot mitigate the exponentially growing runtime requirements of many optimization problems. Unfortunately, the example problems discussed so far are amongst this kind of problem.

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 their 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.

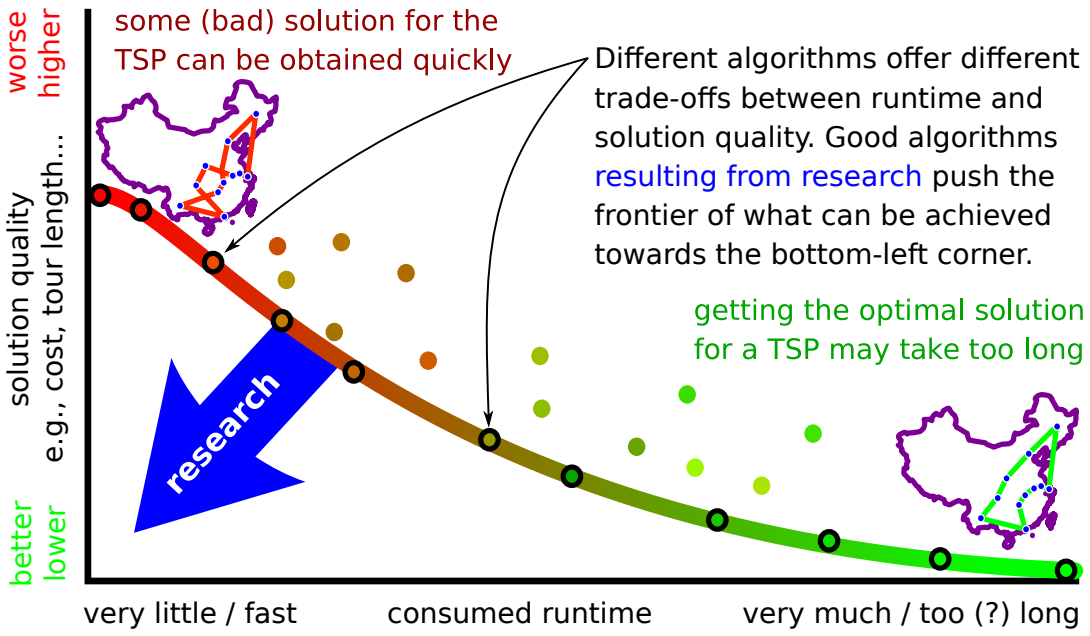

Figure 1.7: The trade-off between solution quality and runtime.

In Figure 1.7 we illustrate this idea on the example of the Traveling Salesman Problem [9,79,109] briefly mentioned in Section 1.1.1. The goal of solving the TSP is to find the shortest round trip tour through n cities. The TSP is $\mathcal{NP}$-hard [65,79]. Today, it is possible to solve many large instances of this problem to optimality by using sophisticated *exact* algorithms [40,41]. Yet, finding the *shortest possible tour* for a particular TSP may (still and probably always in the future) simply take way too long, e.g., in the scale

of many years. Finding just *one tour* is, however, 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 Algorithms within Acceptable 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 [40,41], 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 [86,124,177] and their implementation from the algorithm specification alone can take months [173]. 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. 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.

2 The Structure of Optimization

2.1 Introduction

From the examples that we have seen, we know that optimization problems come in different shapes and forms. Without training, it is not directly clear how to identify, define, understand, or solve them. The goal of this chapter 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 applied in many different scenarios as basis to apply different optimization algorithms.

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

Definition 1. An *optimization problem* is a situation which requires deciding for one choice from a set of possible alternatives in order to reach a predefined/required benefit at minimal costs.

Definition 1 presents an economical point of view on optimization in a rather informal manner. We can refine it to the more mathematical formulation given in Definition 2.

Definition 2. The goal of solving an *optimization problem* is finding an input value $y^\star \in \mathbb{Y}$ 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. First, there is the problem instance data $\mathcal{I}$, i.e., the concrete situation which defines the framework conditions for the solutions we try to find. This input data of the optimization algorithm is discussed in Section 2.2.
2. The second component is the data structure $\mathbb{Y}$ representing possible solutions to the problem. This is the output of the optimization software and is discussed in Section 2.3.
3. Finally, the objective function $f : \mathbb{Y} \mapsto \mathbb{R}$ rates the quality of the candidate solutions $y \in \mathbb{Y}$. This function embodies the goal that we try to achieve, e.g., (minimize) costs. It is discussed in Section 2.4.

If we want to solve a Traveling Salesmen Problem (see Section 1.1.1), then the instance data includes the names of the cities that we want to visit and a map with the information of all the roads between

them (or simply a distance matrix). The candidate solution data structure could simply be a "city list" containing each city exactly once and prescribing the visiting order. The objective function would take such a city list as input and compute the overall tour length. It would be subject to minimization.

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

1. a search space $\mathbb{X}$, i.e., a simpler data structure for internal use, which can more efficiently be processed by an optimization algorithm than $\mathbb{Y}$ (Section 2.6),
2. a representation mapping $\gamma : \mathbb{X} \mapsto \mathbb{Y}$, which translates "points" $x \in \mathbb{X}$ from the search space $\mathbb{X}$ to candidate solutions $y \in \mathbb{Y}$ in the solution space $\mathbb{Y}$ (Section 2.6),
3. search operators $\mathrm{searchOp} : \mathbb{X}^n \mapsto \mathbb{X}$, which allow for the iterative exploration of the search space $\mathbb{X}$ (Section 2.7), and
4. a termination criterion, which tells the optimization process when to stop (Section 2.8).

At first glance, 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 1.1.3 [29,54,73,108,110]. 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 I will intentionally and knowingly 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 here are not necessarily inefficient. While having much room for improvement, we eventually reach approaches that find decent solutions (see, e.g., Section 3.5.4).

2.2 Problem Instance Data

2.2.1 Definitions

We implicitly distinguish optimization problems (see Definition 2) from *problem instances*. While an optimization problem is the general blueprint of the tasks, e.g., the goal of scheduling production jobs

to machines, the problem instance is a concrete scenario of the task, e.g., a concrete lists of tasks, requirements, and machines.

Definition 3. A concrete instantiation of all information that are relevant from the perspective of solving an optimization problems is called a *problem instance* $\mathcal{I}$.

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 Java or a `struct` in C. The `class` defines which member variables exists and what their valid ranges are. An instance of the class is a piece of memory which holds concrete values for each member variable.

2.2.2 Example: Job Shop Scheduling

2.2.2.1 JSSP Instance Structure

So how can we characterize a JSSP instance $\mathcal{I}$? In the a basic and yet general scenario [54,73,108,110], our factory has $m \in \mathbb{N}_1$ machines.[1] At each point in time, a machine can either work on exactly one job or do nothing (be idle). 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 Java source code, and the example instances that we will use, we reference jobs and machines with zero-based indices from $0 \ldots (n-1)$ and $0 \ldots (m-1)$, respectively.

Each of the n jobs is composed of m sub-jobs, one for each machine. Each job may need to pass through these machines in a different order. The sub-job j of job i must be executed on machine $M_{i,j} \in 0 \ldots (m-1)$ and doing so needs $T_{i,j} \in \mathbb{N}_0$ time units for completion.

This definition at first seems strange, but upon closer inspection is quite versatile. Assume that we have a factory that produces exactly one product, but different customers may order different quantities. Here, 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 sub-job 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 or big cannisters. Then, different machines may be needed for different orders. This is similar to the situation illustrated in Figure 1.5, where a certain job i does not need to be executed on a machine j'. We then can simply set the required time $T_{i,j}$ to 0 for the sub-job j with $M_{i,j} = j'$.

[1] where $\mathbb{N}_1$ stands for the natural numbers greater than 0, i.e., 1, 2, 3, ...

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

2.2.2.2 Sources for JSSP Instances

In order to practically play around with optimization algorithms, we need some concrete instances of the JSSP. 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 are published in [3,10,51,54,62,111,152,182]. Their data can be found (sometimes partially) in several repositories in the internet, such as

- the *OR-Library* managed by Beasley [16],
- the comprehensive set of JSSP instances provided by van Hoorn [157,159], where also state-of-the-art results are listed,
- Oleg Shylo's Page [148], which, too, contains up-to-date experimental results,
- Éric Taillard's Page, or, finally,
- my own repository jsspInstancesAndResults [165], 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. 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 four instances, namely

1. instance `abz7` by Adams et al. [3] with 20 jobs and 15 machines
2. instance `la24` by Lawrence [111] with 15 jobs and 10 machines,
3. instance `swv15` by Storer et al. [152] with 50 jobs and 10 machines, and
4. instance `yn4` by Yamada and Nakano [182] with 20 jobs and 20 machines.

These instances are contained in text files available at http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files/jobshop1.txt, https://raw.githubusercontent.com/thomasWeise/jsspInstancesAndResults/master/data-raw/instance-data/instance_data.txt, and in http://jobshop.jjvh.nl/. Of course, if we really want to solve a new type of problem, we will usually use many benchmark problem instances 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 these above four problems.

2.2.2.3 File Format and demo Instance

For the sake of simplicity, we created one additional, smaller instance to describe the format of these files, as illustrated in Figure 2.1.

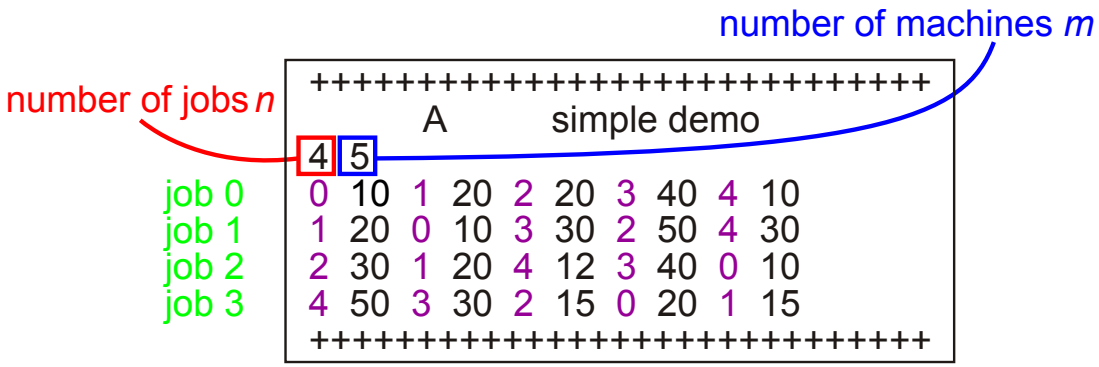

Figure 2.1: The meaning of the text representing our demo instance of the JSSP, as an example of the format used in the OR-Library.

In the simple text format used in OR-Library, several problem instances can be contained in one file. Each problem instance $\mathcal{I}$ is 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 indexes in Java. The JSSP instance definition is completed by n lines of text, each of which specifying the sub-jobs of one job $i \in 0 \ldots (n-1)$. Each sub-job 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 $0 \ldots (m-1)$, followed by the number of time units $T_{i,j}$ the job will take on that machine. The order of the sub-jobs 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 2.1, 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 at least 100 time units to be completed, if it can be scheduled without any delay or waiting period, i.e., if all of its sub-jobs 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 – althoug 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.

2.2.2.4 A Java Class for JSSP Instances

This structure of a JSSP instance can be represented by the simple Java class given in Listing 2.1.

Listing 2.1 Excerpt from a Java class for representing the data of a JSSP instance. (src)

```
1 public class JSSPInstance {
2   public int m;
3   public int n;
4   public int[][] jobs;
5 }
```

Here, the two-dimensional array `jobs` directly receives the data from sub-job lines in the text files, i.e., each row stands for a job and contains machine IDs and processing times in an alternating sequence. The actual source file of the class `JSSPInstance` accompanying our book also contains additional code, e.g., for reading such data from the text file, which we have omitted here as it is unimportant for the understanding of the scenario.

2.3 The Solution Space

2.3.1 Definitions

As stated in Definition 1, an optimization problem asks us to make a choice between different possible solutions. We call them *candidate solutions*.

Definition 4. A *candidate solution* y is one potential solution of an optimization problem.

Definition 5. The *solution space* $\mathbb{Y}$ of an optimization problem is the set of all of its candidate solutions $y \in \mathbb{Y}$.

Basically, the input of an optimization algorithm is the problem instance $\mathcal{I}$ and 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 for 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.

From the programmer's perspective, the solution space is again a data structure, e.g., a `class` in Java. An instance of this data structure is the candidate solution.

2.3.2 Example: Job Shop Scheduling

What would be a candidate solution to a JSSP instance as defined in Section 2.2.2? Recall from Section 1.1.3 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.

2.3.2.1 Idea: Gantt Chart

This is basically what Gantt chart [104,181] are for, as illustrated in Figure 2.2. A Gantt chart defines what each of our m machines has to do at each point in time. The sub-jobs of each job are assigned to time windows on their corresponding machines.

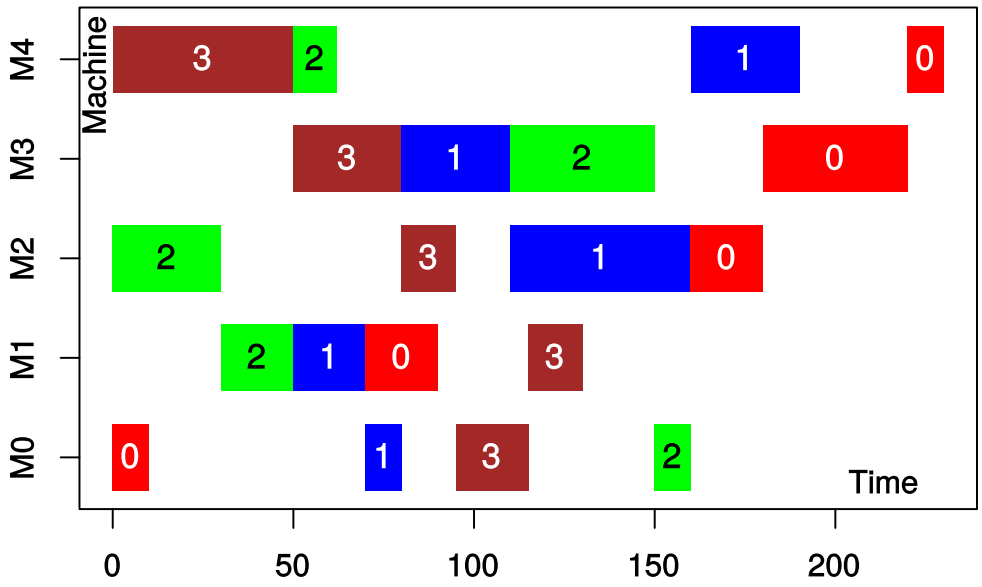

Figure 2.2: One example candidate solution for the demo instance given in Figure 2.1: A Gantt chart assigning a time window to each job on each machine.

The Gantt chart contains one row for each machine. It is to be read from left to right, where the x-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 sub-job j of job i is painted in the row of machine $M_{i,j}$ and its length equals the time requirement $T_{i,j}$.

The chart given in Figure 2.2, for instance, 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.

If we wanted to create a Java class to represent the complete information from a Gantt diagram, it could look like Listing 2.2. Here, for each of the m machines, we create one integer array of length $3n$. Such an array stores three numbers for each of the n sub-jobs to be executed on the machine: the job ID, the start time, and the end time.

Listing 2.2 Excerpt from a Java class for representing the data of a candidate solution to a JSSP. (src)

```
1 public class JSSPCandidateSolution {
2   public int[][] schedule;
3 }
```

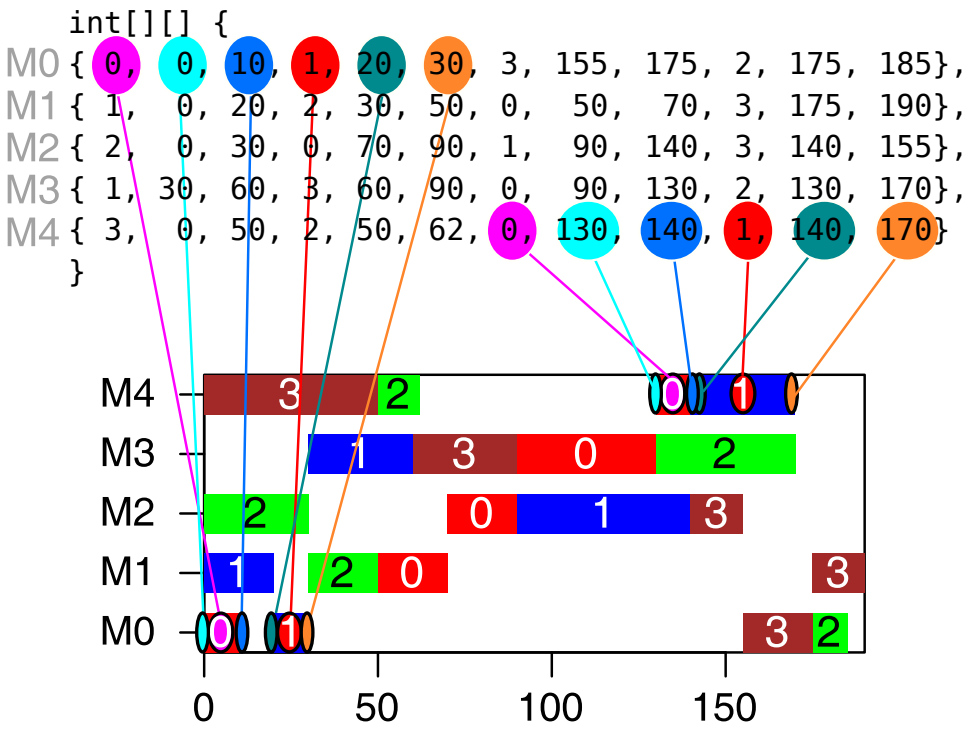

Figure 2.3: An example how the internal `int[][]` data of the `JSSPCandidateSolution` class maps to a Gantt chart.

Of course, we would not strictly need a class for that, as we could as well use the integer array `int[][]`

directly.

Also the third number, i.e., the end time, is not strictly necessary, as it can be computed based on the instance data as $start + T_{i,j'}$ for job i on machine j after searching j' such that $M_{i,j'} = j$. Another form of representing a solution would be to just map each sub-job to a starting time, leading to $m * n$ integer values per candidate solution [158]. But the presented structure – illustrated on an example in **??** – is handy and easier to understand. It allows the human operator to directly see what is going on, to 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.

2.3.2.2 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 we allow arbitrary useless waiting times between jobs, 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.

There are $n! = \prod_{i=1}^{n} i$ possible ways to arrange n jobs on one machine. $n!$, 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 would have $n = 3$ jobs and $m = 2$ machines, we then would have $(3!)^2 = 36$ possible Gantt charts, as for each of the 6 possible sequence of jobs on the first machines, there would be 6 possible arrangements on the second machine. For $m = 2$ machines, it is then $(n!)^3$, and so on. In the general case, we obtain Equation (2.1) for the size $|\mathbb{Y}|$ of the solution space $\mathbb{Y}$.

$$|\mathbb{Y}| = (n!)^m \tag{2.1}$$

However, the fact that we can generate $(n!)^m$ possible Gantt charts without useless delay for a JSSP with n jobs and m machines does not mean that all of them are actual *feasible* solutions.

2.3.2.3 The Feasibility of the Solutions

Definition 6. 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 7. A candidate solution $y \in \mathbb{Y}$ is *feasible* if and only if it fulfills all constraints.

Definition 8. A candidate solution $y \in \mathbb{Y}$ is *infeasible* if it is *not feasible*, i.e., if it violates at least one constraint.

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

1. all sub-jobs 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. a sub-job will must be assigned a time window on its corresponding machine which is exactly as long as the sub-job needs on that machine,
4. the sub-jobs cannot intersect or overlap, each machine can only carry out one job at a time, and
5. the precedence constraints of the sub-jobs must be honored.

While the first four *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 first to machine 1 and the second job first to 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.

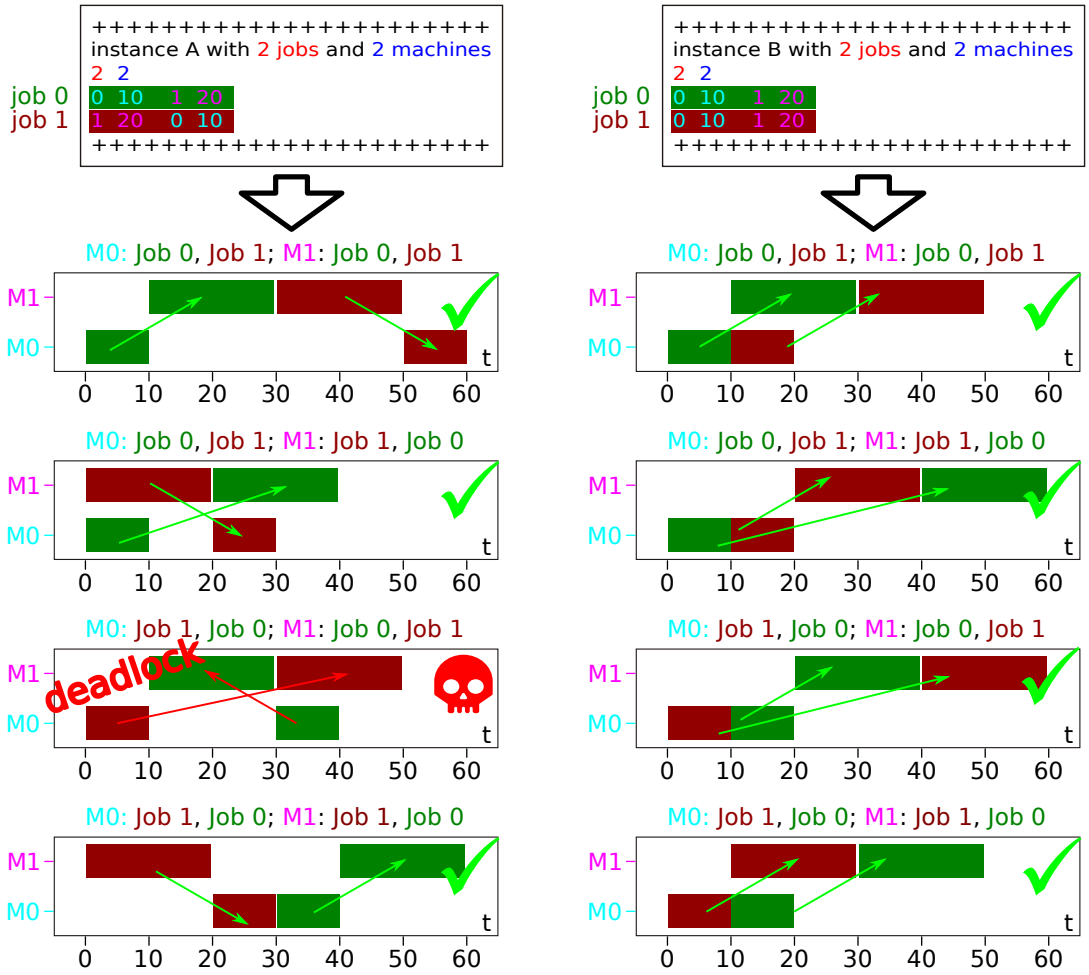

Figure 2.4: Two different JSSP instances with $m = 2$ machines and $n = 2$ jobs, one of which has only three feasible candidate solutions while the other has four.

The third schedule in the first column of Figure 2.4 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 sub-jobs 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 Figure 2.4. The number of actually feasible Gantt charts in $\mathbb{Y}$ thus is different for different problem instances.

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 which might sometimes return infeasible solutions will not be acceptable.

2.3.2.4 Summary

Table 2.1: 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 1.6).

name	n	m	$\min(\#\text{feasible})$	$\lvert\mathbb{Y}\rvert$
	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
	5	4		207’360’000
	5	5		24’883’200’000
demo	4	5		7’962’624

name	n	m	$\min(\#\text{feasible})$	$\lvert\mathbb{Y}\rvert$
la24	15	10		$\approx 1.462*10^{121}$
abz7	20	15		$\approx 6.193*10^{275}$
yn4	20	20		$\approx 5.278*10^{367}$
swv15	50	10		$\approx 6.772*10^{644}$

We illustrate some examples for the number $|\mathbb{Y}|$ of schedules which do not waste time uselessly for different values of n and m in Table 2.1. Since we use instances for testing our JSSP algorithms, we have added their settings as well and listed them in column "name". Of course, there are infinitely many JSSP instances for a given setting of n and m and our instances always only mark single examples for them.

We find that even small problems with $m = 5$ machines and $n = 5$ jobs already have billions of possible solutions. The four more realistic problem instances which we will try to solve here already have more solutions that what we could ever enumerate, list, or store with any conceivable hardware or computer. As 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 what we will discuss in the following.

The number $\#$feasible of possible *feasible* Gantt charts can be different, depending on the problem instance. For one setting of m and n, we are interested in the minimum $\min(\#\text{feasible})$ of this number, i.e., the *smallest value* that $\#$feasible can take on over all possible instances with n jobs and m machines. It is not so easy to find a formula for this minimum, so we won't do this here. Instead, in Table 2.1, we provided the corresponding numbers for a few selected instances. We find that, if we are unlucky, most of the possible Gantt charts for a problem instance might be infeasible, as $\min(\#\text{feasible})$ can be much smaller than $|\mathbb{Y}|$.

2.4 Objective Function

We now know the most important 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, 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.

2.4.1 Definitions

Definition 9. An *objective function* $f : \mathbb{Y} \mapsto \mathbb{R}$ rates the quality of a candidate solution $y \in \mathbb{Y}$ from the solution space $\mathbb{Y}$ as real number.

Definition 10. An *objective value* $f(y)$ of the candidate solution $y \in \mathbb{Y}$ is the value that the objective function f takes on for y.

Without loss of generality, we assume that all objective functions are subject to *minimization*, meaning that smaller objective values are better. 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 of 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, any number of required resources.

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 minimization problem, we can solve the maximization problem with $-f$ and get the same result, and vice versa.

From the perspective of a Java programmer, the general concept of objective functions can be represented by the interface given in Listing 2.3. The `evaluate` function of this interface accepts one instance of the solution space class Y and returns a `double` value. `double`s are floating point numbers in Java, i.e., represent a subset of the real numbers. We keep the interface generic, so that we can implement it for arbitrary solution spaces. Any actual objective function would then be an implementation of that interface.

Listing 2.3 A generic interface for objective functions. (src)

```
1 public interface IObjectiveFunction<Y> {
2   public abstract double evaluate(Y y);
3 }
```

2.4.2 Example: Job Shop Scheduling

As stated in Section 1.1.3, our goal is to complete the production jobs as soon as possible. This means that we want to minimize the makespan, the time when the last job finishes. Obviously, the smaller this value, the earlier we are done with all jobs, the better is the plan. As illustrated in Figure 2.5, 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 sub-job of job 0, executed on machine 4.

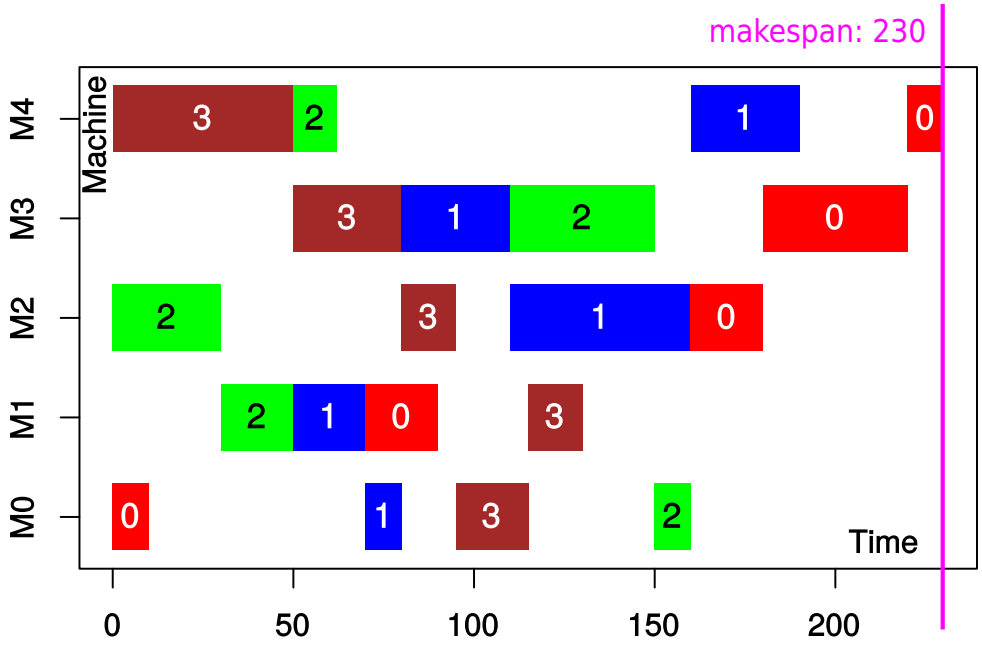

Figure 2.5: The makespan (purple), i.e., the time when the last job is completed, for the example candidate solution illustrated in Figure 2.2 for the demo instance from Figure 2.1.

Our objective function f is thus equivalent to the makespan and subject to minimization. Based on our candidate solution data structure from Listing 2.2, we can easily compute f. We simply have to look at the last number in each of the integer arrays stored in the member `schedule`, as it represents the end time of the last job processed by a machine. We then return the largest of these numbers. We implement the interface `IObjectiveFunction` in class `JSSPMakespanObjectiveFunction` accordingly in Listing 2.4.

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 $f(y_1) < f(y_2)$.[2]

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

Listing 2.4 Excerpt from a Java class computing the makespan resulting from a candidate solution to the JSSP. (src)

```
public class JSSPMakespanObjectiveFunction
    implements IObjectiveFunction<JSSPCandidateSolution> {
  public double evaluate(JSSPCandidateSolution y) {
    int makespan = 0;
// look at the schedule for each machine
    for (int[] machine : y.schedule) {
// the end time of the last job on the machine is the last number
// in the array, as array machine consists of "flattened" tuples
// of the form ((job, start, end), (job, start, end), ...)
      int end = machine[machine.length - 1];
      if (end > makespan) {
        makespan = end; // remember biggest end time
      }
    }
    return makespan;
  }
}
```

2.5 Global Optima and Lower Quality Bounds

We now know the 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"?

2.5.1 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 11. 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.

Definition 12. For every *global optimum* $y^\star \in \mathbb{Y}$ 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 12 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 a "globally optimal" is nothing else than "superlative" [27]. 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, we look for the packing that needs the *least* amount of containers. Thus, optimization means searching for such superlatives, as illustrated in Figure 2.6. Vice versa, whenever we are looking for the cheapest, fastest, strongest, best, biggest or smallest "thing", then we have an optimization problem at hand.

Figure 2.6: Optimization is the search for superlatives [27].

For example, 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* if all n jobs need to be processed by the two machines in exactly the same order [99]. If our application always falls into a special case of the problem, we may be lucky to find an efficient way to always solve it to optimality. The general version of the JSSP, however, is $\mathcal{NP}$-hard [32,110], meaning that we cannot expect to solve it to global optimality in reasonable time. Developing a good (meta-)heuristic algorithm, which cannot provide guaranteed optimality but will give close-to-optimal solutions in practice, is a good choice.

2.5.2 Bounds of the Objective Function

If we apply an approximation algorithm, then we do not have the guarantee that the solution we get is optimal. We often do not even know if the best solution we currently have is optimal or not. In some cases, we be able to compute a *lower bound* $\mathrm{lb}(f)$ for the objective value of an optimal solution, i.e., we know "It is not possible that any solution can have a quality better than $\mathrm{lb}(f)$, but we may not know whether a solution actually exists that has quality $\mathrm{lb}(f)$." This is not useful for solving the problem, but it can tell us whether our method for solving the problem is good. For instance, if we have developed an algorithm for approximately solving a given problem and we *know* that the qualities of the solutions

we get are close to a the lower bound, then we know that our algorithm is good. We then 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.

2.5.3 Example: Job Shop Scheduling

We have already defined our solution space $\mathbb{Y}$ for the JSSP in Listing 2.2 and the objective function f in Listing 2.3. 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 it. But we can, at least, compute some *lower bound* $\mathrm{lb}(f)$ for the best possible makespan.

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 sub-jobs. It is clear that no schedule can complete faster then the longest job. 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 sub-jobs assigned to this machine. But it may also include a least initial idle time a_j, namely if the sub-jobs for machine j never come first in their job. Similarly, there is a least idle time c_j at the end if these sub-jobs never come last in their job. As lower bound for the fastest schedule that could theoretically exist, we therefore get:

$$\mathrm{lb}(f) = \max \left\{ \max_i \left\{ \sum_{j=0}^{m-1} T_{i,j} \right\} \ , \ \max_j \left\{ a_j + b_j + c_j \right\} \right\} \tag{2.2}$$

More details are given in Section 6.1.1 and [54].

Of course, we cannot know whether a schedule exists that can achieve this lower bound makespan. There simply may not be any way to arrange the jobs such that no sub-job stalls any other sub-job. This is why the value $\mathrm{lb}(f)$ is a 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. The lower bounds for the makespans of our example problems are illustrated in Table 2.2.

Table 2.2: The lower bounds $\mathrm{lb}f$ for the makespan of the optimal solutions for our example problems. For the instances `abz7`, `la24`, and `yn4`, research literature (last column) provides better (i.e., higher) lower bounds $\mathrm{lb}(f)^\star$.

name	n	m	$\mathrm{lb}(f)$	$\mathrm{lb}(f)^\star$	source for $\mathrm{lb}(f)^\star$
`demo`	4	5	180	180	Equation (2.2)
`abz7`	20	15	638	656	[117,157,161,162]
`la24`	15	10	872	935	[10,157]
`swv15`	50	10	2885	2885	Equation (2.2)
`yn4`	20	20	818	929	[157,161,162]

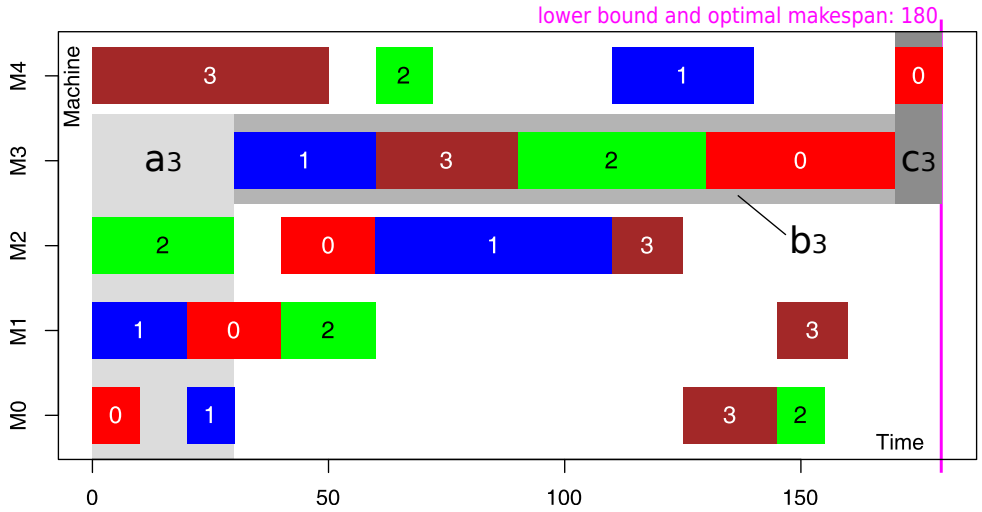

Figure 2.7: The globally optimal solution of the demo instance Figure 2.1, whose makespan happens to be the same as the lower bound.

Figure 2.7 illustrates the globally optimal solution for our small demo JSSP instance defined in Figure 2.1 (we will get to how to find such a solution later). 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 2.1. 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 no first sub-job of any job starts at 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. Also, no job in the demo instance finishes at machine 3. Job 0, for instance, needs to be processed by machine 4 for 10 time units after it has passed through machine 3. 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. 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 sub job. In between, it will need to work for exactly 140 time units, the total sum of the running time of all sub-jobs assigned to it. This means that no schedule can complete faster than $30 + 140 + 10 = 180$ time units. Thus, Figure 2.7 illustrates the optimal solution for the demo instance.

Then, all the jobs together on the machine will consume $b_3 = 150$ time units if we can execute them without further delay. Finally, it regardless with which job we finish on this machine, it will lead to a further waiting time of $c_3 = 10$ time units. This leads to a lower bound $\mathrm{lb}(f)$ of 180 and since we found the illustrated candidate solution with exactly this makespan, we have solved this (very easy) JSSP instance.

Listing 2.5 A generic interface for objective functions, now including a function for the lower bound. (src)

```
1 public interface IObjectiveFunction<Y> {
2   public abstract double evaluate(Y y);
3   public default double lowerBound() {
4     return Double.NEGATIVE_INFINITY;
5   }
6 }
```

We can extend our interface for objective functions in Listing 2.5 to now also allow us to implement a function `lowerBound` which returns, well, the lower bound. If we have no idea how to compute that for a given problem instance, this function can simply return $-\infty$.

2.6 The Search Space and Representation Mapping

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) at the end of the optimization procedure. But $\mathbb{Y}$ it 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. It would be nice to have a compact, clear, and easy-to-understand representation of the candidate solutions.

2.6.1 Definitions

Definition 13. The *search space* $\mathbb{X}$ is a representation of the solution space $\mathbb{Y}$ suitable for exploration by an algorithm.

Definition 14. The elements $x \in \mathbb{X}$ of the search space $\mathbb{X}$ are called *points* in the search space.

Definition 15. The *representation mapping* $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ is a left-total relation which maps the points $x \in \mathbb{X}$ of the search space $\mathbb{X}$ to the candidate solutions $y \in \mathbb{Y}$ in the solution space $\mathbb{Y}$.

For applying an optimization algorithm, we 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 instances:

1. subsets of s-dimensional real vectors, i.e., $\mathbb{R}^s$,
2. the set $\mathbb{P}(s)$ of sequences/permutations of s objects, and
3. a number of s yes-no decisions, which can be represented as bit strings of length s and spans the space $\{0,1\}^s$.

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 representation mapping. 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 [106]. 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 reside in the set of such solutions to which no point in the search space can be mapped, then, well, it could not be found by the optimization process. Being surjective is therefore a good feature for γ.

The interface given in Listing 2.6 provides a function `map` which maps one point `x` in the search space class `X` to a candidate solution instance `y` of the solution space class `Y`. We define the interface as generic, because we here do not make any assumption about the nature of `X` and `Y`. This interface therefore truly corresponds to the general definition $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ of the representation mapping. Side note: An implementation of `map` will overwrite whatever contents were stored in object `y` in the process, i.e., we assume that `Y` is a class whose instances can be modified.

Listing 2.6 A general interface for representation mappings. (src)

```
1 public interface IRepresentationMapping<X, Y> {
2   public abstract void map(Random random, X x,
3       Y y);
4 }
```

2.6.2 Example: Job Shop Scheduling

In our JSSP example, we have developed the class `JSSPCandidateSolution` given in Listing 2.2 to represent the data of a Gantt chart (candidate solution). It can easily be interpreted by the user and we have defined a suitable objective function for it in Listing 2.4. 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.ˆ[Of course, there are many algorithms that can do that and we could discover one if we think about it for a bit, but here we take the educational route where we investigate the full scenario with $\mathbb{X} \neq \mathbb{Y}$.] 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. While comprehensive overviews about different such search spaces for the JSSP can be found in [2:[@YN1997GAFJSSP],35:[@W2013GAFSSPAS]], we here develop only one single idea which I find particularly appealing.

2.6.2.1 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 sub-jobs, one for each machine. We could simply begin by choosing one job and placing its first sub-job 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 sub-job 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 sub-jobs all have already be assigned. We can continue doing this until all jobs are assigned – and we will get a valid solution.

This 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. If we process such a string from the beginning to the end and step-by-step assign the jobs, we get a feasible Gantt chart as result.

The encoding and corresponding representation mapping can best be described by an example. In the demo instance, we have $m = 5$ machines and $n = 4$ jobs. Each job has $m = 5$ sub-jobs that must be distributed to the machines. We use a string of length $m * n = 20$ denoting the priority of the sub-jobs. We *know* the order of the sub-jobs per job as part of the problem instance data $\mathcal{I}$. We therefore do not

need to encode it. This means that we just include each job's id $m = 5$ times in the string. This was the original idea: 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 $\mathbb{P}(n * m)$ of permutations of $n * m$ objects mentioned earlier, but instead of permutations, we have *permutations with repetitions*.

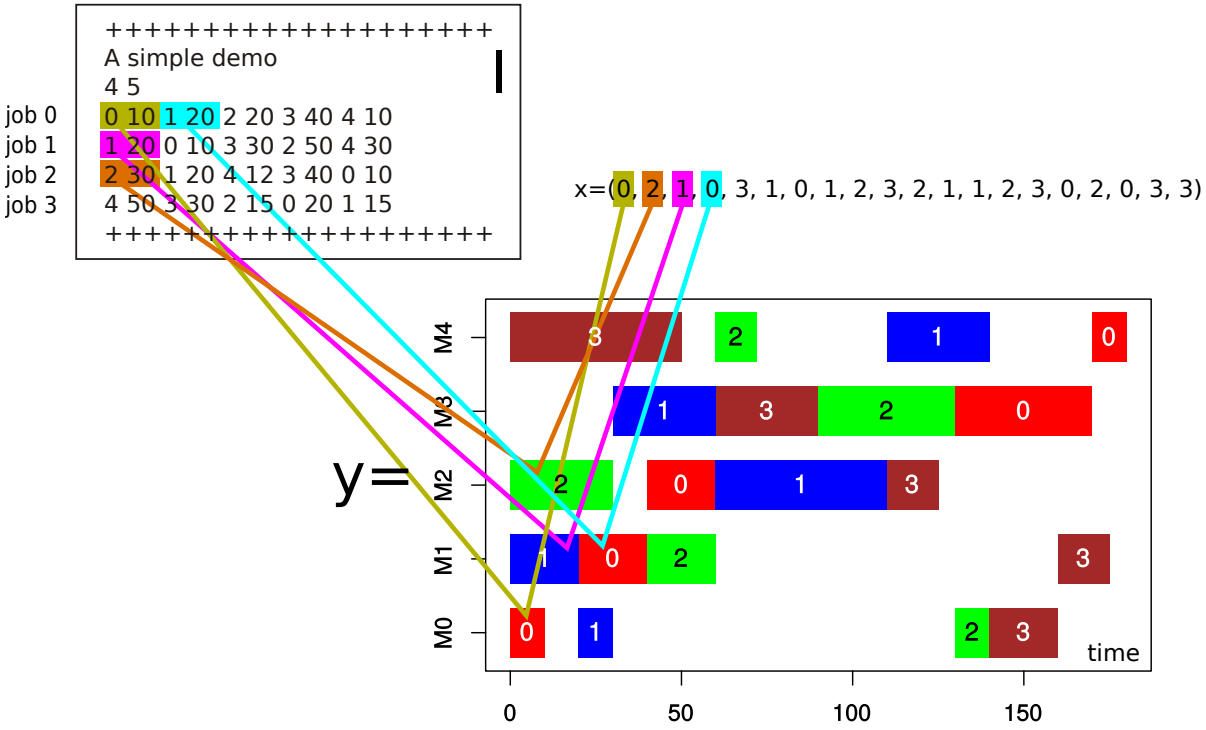

Figure 2.8: Illustration of the first four steps of the representation mapping of an example point in the search space to a candidate solution.

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 chose $x = (0, 2, 1, 0, 3, 1, 0, 1, 2, 3, 2, 1, 1, 2, 3, 0, 2, 0, 3, 3)$ in Figure 2.8. The representation mapping 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 job 0 is assigned to a machine first. 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, it can be placed at time index 0. We also know that this sub-job can definitely be executed, i.e., won't cause a deadlock, because it is the first sub-job of the job.

The next number in the string is 2, so job 2 is next. This job needs to go for 30 time units to machine 2, which also is initially idle. So it can be inserted into the candidate solution directly as well (and cannot cause any deadlock either).

Then job 1 is next in x, and from the instance data we can see that it will go to machine 1 for 20 time units. This machine is idle as well, so the job can start immediately.

We now encounter job 0 again in the integer string. Since we have already performed the first sub-job of job 0, we now would like to schedule its second sub-job. According to the instance data, the second sub-job takes place on machine 1 and will take 20 time units. We know that completing the first sub-job took 10 time units. We also know that machine 1 first has to process job 1 for 20 time units. The earliest possible time at which we can begin with the second sub-job of job 0 is thus at time unit 20, namely the bigger one of the above two values. This means that job 0 has to wait for 10 time units after completing its first sub-job and then can be processed by machine 1. No deadlock can occur, as we made sure that the first sub-job of job 0 has been scheduled before the second one.

We now encounter job 3 in the integer string, and we know that job 3 first goes to machine 4, which currently is idle. It can thus directly be placed on machine 4, which it will occupy for 50 time units.

Then we again encounter job 1 in the integer string. Job 1 should, in its second sub-job, go to machine 0. Its first sub-job to 20 time units on machine 1, while machine 0 was occupied for 10 time units by job 0. We can thus start the second sub-job of job 1 directly at time index 20.

Further processing of y leads us to job 0 again, which means we will need to schedule its third sub-job, which will need 20 time units on machine 2. Machine 2 is occupied by job 2 from time unit 0 to 30 and becomes idle thereafter. The second sub-job of job 0 finishes on time index 40 at machine 1. Hence, we can begin with the third sub-job at time index 40 at machine 2, which had to idle for 10 time units.

We continue this iterative processing until reaching the end of the string x. We now have constructed the complete Gantt chart y illustrated in Figure 2.8. Whenever we assign a sub-job $i > 0$ of any given job to a machine, then we already had assigned all sub-jobs at smaller indices first. No deadlock can occur and y must therefore be feasible.

In Listing 2.7, we illustrate how such a mapping can be implemented. It basically is a function translating an instance of `int[]` to `JSSPCandidateSolution`. This is done by keeping track of time that has passed for each machine and each job, as well as by remembering the next sub-job for each job and the position in the schedule of each machine.

2.6.2.2 Advantages of a very simple Encoding

This is a very nice and natural way to represent Gantt charts with a much simpler data structure. As a result, it has been discovered by several researchers independently, the earliest being Gen et al. [67], Bierwirth [21,22], and Shi et al. [146], all in the 1990s.

But what do we gain by using this search space and representation mapping? First, well, we now have a very simple data structure $\mathbb{X}$ to represent our candidate solutions. Second, we also have very simple

rules for validating a point x in the search space: If it contains the numbers $0 \ldots (n-1)$ each exactly m times, it represents a feasible candidate solution.

Third, the candidate solution corresponding to a valid point from the search space will always be *feasible* [21]. The mapping γ will ensure that the order of the sub-jobs per job is always observed. We do not need to worry about the issue of deadlocks mentioned in Section 2.3.2.3. We know from Table 2.1, that the vast majority of the possible Gantt charts for a given problem may be infeasible – and now we do no longer need to worry about that. Our mapping also makes sure of the more trivial constraints, such as that each machine will process at most one job at a time and that all sub-jobs are eventually processed.

Finally, we also could modify our representation mapping 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 time requirement for carrying a job from one machine to another, then we could facilitate this by changing γ so that it adds this time to the starting time of the job. If there was a job-dependent setup time for each machine [5], which could be different if job 1 follows job 0 instead of job 2, then this could be facilitated easily as well. If our sub-jobs 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 sub-jobs 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 to simple scenarios, such as the Flow Shop Problem, where all jobs pass through the machines in the same, pre-determined order [54,66,175].

Many such different problem flavors can now be reduced to investigating the same space $\mathbb{X}$ using the same optimization algorithms, just with different representation mappings γ and/or objective functions f. Additionally, it became very 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.

2.6.2.3 Size of the Search Space

It is relatively easy to compute the size $|\mathbb{X}|$ of our proposed search space $\mathbb{X}$ [146]. We do not need to make any assumptions regarding "no useless waiting time", as in Section 2.3.2.2, since this is not possible 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 (2.3).

$$|\mathbb{X}| = \frac{(m*n)!}{(m!)^n} \tag{2.3}$$

Table 2.3: The sizes $|\mathbb{X}|$ and $|\mathbb{Y}|$ of the search and solution spaces for selected values of the number n of jobs and the number m of machines of an JSSP instance $\mathcal{I}$. (compare with Figure 1.6 and with the size $|\mathbb{Y}|$ of the solution space); compare with Figure 5.8

name	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
	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	623'360'743'125'120
demo	4	5	7'962'624	11'732'745'024
la24	15	10	$\approx 1.462*10^{121}$	$\approx 2.293*10^{164}$
abz7	20	15	$\approx 6.193*10^{275}$	$\approx 1.432*10^{372}$
yn4	20	20	$\approx 5.278*10^{367}$	$\approx 1.213*10^{501}$
swv15	50	10	$\approx 6.772*10^{644}$	$\approx 1.254*10^{806}$

We give some example values for this search space size $|\mathbb{X}|$ in Table 2.3. 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 an JSSP instance $\mathcal{I}$.

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 and 7 million possible, non-wasteful candidate solutions.

We now find the drawback of our encoding: There is some redundancy in our mapping, γ here is not injective. If we would exchange the first three numbers in the example string in Figure 2.8, we would obtain the same Gantt chart, as jobs 0, 1, and 2 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.

Listing 2.7 Excerpt from a Java class for implementing the representation mapping. (src)

```
public class JSSPRepresentationMapping implements
    IRepresentationMapping<int[], JSSPCandidateSolution> {
  public void map(Random random, int[] x,
      JSSPCandidateSolution y) {
// create variables machineState, machineTime of length m and
// jobState, jobTime of length n, filled with 0 [omitted brevity]
// iterate over the jobs in the solution
    for (int nextJob : x) {
// get the definition of the steps that we need to take for
// nextJob from the instance data stored in this.m_jobs
      int[] jobSteps = this.m_jobs[nextJob];
// jobState tells us the index in this list for the next step to
// do, but since the list contains machine/time pairs, we
// multiply by 2 (by left-shifting by 1)
      int jobStep = (jobState[nextJob]++) << 1;

// so we know the machine where the job needs to go next
      int machine = jobSteps[jobStep]; // get machine

// start time is maximum of the next time when the machine
// becomes idle and the time we have already spent on the job
      int start =
          Math.max(machineTime[machine], jobTime[nextJob]);
// the end time is simply the start time plus the time the job
// needs to spend on the machine
      int end = start + jobSteps[jobStep + 1]; // end time
// it holds for both the machine (it will become idle after end)
// and the job (it can go to the next station after end)
      jobTime[nextJob] = machineTime[machine] = end;

// update the schedule with the data we have just computed
      int[] schedule = y.schedule[machine];
      schedule[machineState[machine]++] = nextJob;
      schedule[machineState[machine]++] = start;
      schedule[machineState[machine]++] = end;
    }
  }
}
```

2.7 Search Operations

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

2.7.1 Definitions

Definition 16. An n-ary *search operator* $\mathrm{searchOp} : \mathbb{X}^n \mapsto \mathbb{X}$ is a left-total relation which accepts n points in the search space $\mathbb{X}$ as input and returns one point in the search space as output.

Special cases of search operators are

- nullary operators ($n = 0$, see Listing 2.8), which sample a new point from the search space without using any information from an existing points,
- unary operators ($n = 1$, see Listing 2.9), which sample a new point from the search space based on the information of one existing point,
- binary operators ($n = 2$, see Listing 2.10), which sample a new point from the search space by combining information from two existing points, and
- ternary operators ($n = 3$), which sample a new point from the search space by combining information from three existing points.

Listing 2.8 A generic interface for nullary search operators. (src)

```
1 public interface INullarySearchOperator<X> {
2   public abstract void apply(X dest, Random random);
3 }
```

Listing 2.9 A generic interface for unary search operators. (src)

```
1 public interface IUnarySearchOperator<X> {
2   public abstract void apply(X x, X dest,
3       Random random);
4 }
```

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. This is why Listings 2.8 to 2.10 all accept an instance of

Listing 2.10 A generic interface for binary search operators. (src)

```
1 public interface IBinarySearchOperator<X> {
2   public abstract void apply(X x0, X x1,
3       X dest, Random random);
4 }
```

`java.util.Random`, a pseudorandom number generator. They allow us to define proximity-based relationships over the search space, such as the common concept of neighborhoods.

Definition 17. A unary operator $\mathrm{searchOp} : \mathbb{X} \mapsto \mathbb{X}$ defines a *neighborhood* relationship over a search space where a point $x_1 \in \mathbb{X}$ is called a *neighbor* of a point $x_2 \in \mathbb{X}$ are called neighbors if and only if x_1 could be the result of an application of $\mathrm{searchOp}$ to x_2.

2.7.2 Example: Job Shop Scheduling

We will step-by-step introduce the concepts of nullary, unary, and binary search operators in the subsections of chapter 3 on metaheuristics as they come. This makes more sense from a didactic perspective.

2.8 The Termination Criterion and the Problem of Measuring Time

We have seen that the search space for even small instances of the JSSP can already be quite large. We simply cannot enumerate all of them, as it would take too long. This raises the question: "If we cannot look at all possible solutions, how can we find the global optimum?" We may also ask: "If we cannot look at all possible solutions, how can we know whether a given candidate solution is the global optimum or not?" In many optimization scenarios, we can use theoretical bounds to solve that issue, but a priori, these questions are valid and their answer is simply: *No*. *No*, without any further theoretical investigation of the optimization problem, we don't know if the best solution 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 or should continue trying to solve the problem? There are two basic answers: Either when the time is up or when we found a reasonably-good solution.

2.8.1 Definitions

Definition 18. The *termination criterion* is a function of the state of the optimization process which becomes `true` if the optimization process should stop (and then remains `true`) and remains `false` as long as it can continue.

Listing 2.11 A general interface for termination criteria. (src)

```
1 public interface ITerminationCriterion {
2   public abstract boolean shouldTerminate();
3 }
```

With a termination criterion defined as implementation of the interface given in Listing 2.11, we can embed any combination of time or solution quality limits. We could, for instance, define a goal objective value 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 $f(y) \leq g$, i.e., which is at least as good as the goal. 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. Discussions of both approaches from the perspective of measuring algorithm performance are given in Sections 4.2 and 4.3.

2.8.2 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 [32,110]. So can we guarantee to find a solution which is, say, 1% worse, until she finishes her drink? Well, it was shown that there is *no* algorithm which can guarantee us to find a solution only 25% worse than the optimum within a runtime polynomial in the problem size [97,180] 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 [116]. 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 makes little sense.

Hence, we should rely on the simple practical concern: The operator drinks a coffee. A termination criterion granting three 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, e.g., whether a proposed schedule can be implemented/completed within the working hours of a single day. We might let the algorithm run longer than three minutes until such a solution was discovered. But, as said before, if a very odd scenario occurs, it might take a long time to discover such a solution, if ever. The operator may also need to be given the ability to manually stop the process and extract the best-so-far solution if needed. For our benchmark instances, however, this is not relevant and we can limit ourselves to the runtime-based termination criterion.

2.9 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 exercise a 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, i.e., the input data $\mathcal{I}$ that our program will receive.
2. The second step is to understand what our users will consider as a solution – a Gantt chart, for instance. 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 don't 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:

5. 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 $\mathbb{X}$ is much easier

than $\mathbb{Y}$. Of course, defining such a data structure $\mathbb{X}$ makes only sense if we can also define a mapping γ from $\mathbb{X}$ to $\mathbb{Y}$.

6. 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.
7. We test, benchmark, and compare several algorithms to pick those with the best and most reliable performance (see chapter 4).

3 Metaheuristic Optimization Algorithms

Metaheuristics [37,68,69,163] are the most important class of algorithms that we explore in this book.

Definition 19. A *metaheuristic* is a general algorithm that can produce approximate solutions for a class of different optimization problems.

These algorithms have the advantage that we can easily adapt them to new optimization problems. As long as we can construct the elements discussed in chapter 2 for a problem, we can attack it with a metaheuristic. In this chapter, we will introduce several such general algorithms. We explore them by again using the Job Shop Scheduling Problem (JSSP) from Section 1.1.3 as example.

3.1 Common Characteristics

Before we delve into our first algorithms, let us first take a look on some things that all metaheuristics have in common.

3.1.1 Anytime Algorithms

Definition 20. An *anytime algorithm* is an algorithm which can produce an approximate result during almost any time of its execution.

All metaheuristics – and many other optimization and machine learning methods – are anytime algorithms [24]. The idea behind anytime algorithms is that they start with (potentially bad) guess about what a good solution would be. During their course, they try to improve their approximation quality, by trying to produce better and better candidate solutions. At any point in time, we can extract the current best guess about the optimum. This fits to the optimization situation that we have discussed in Section 2.8: We often cannot find out whether the best solution we currently have is the globally optimal solution for the given problem instance or not, so we simply continue trying to improve upon it until a *termination criterion* tells us to quit, e.g., until the time is up.

3.1.2 Return the Best-So-Far Candidate Solution

This one is actually quite simple, yet often ignored and misunderstood by novices to the subject: Regardless what the optimization algorithm does, it will never *NEVER* forget the best-so-far candidate solution. Often, this is not explicitly written in the formal definition of the algorithms, but there *always* exists a special variable somewhere storing that solution. This variable is updated each time a better solution is found. Its value is returned when the algorithm stops.

3.1.3 Randomization

Often, metaheuristics make randomized choices. In cases where it is not clear whether doing "A" or doing "B" is better, it makes sense to simply flip a coin and do "A" if it is heads and "B" if it is tails. That our search operator interfaces in Listings 2.8 to 2.10 all accept a pseudorandom number generator as parameter is one manifestation of this issue. Random number generators are objects which provide functions that can return numbers from certain ranges, say from $[0, 1)$ or an integer interval. Whenever we call such a function, it may return any value from the allowed range, but we do not know which one it will be. Also, the returned value should be independent from those returned before, i.e., from known the past random numbers, we should *not* be able to guess the next one. By using such random number generators, we can let an algorithm make random choices, randomly pick elements from a set, or change a variable's value in some unpredictable way.

3.1.4 Black-Box Optimization

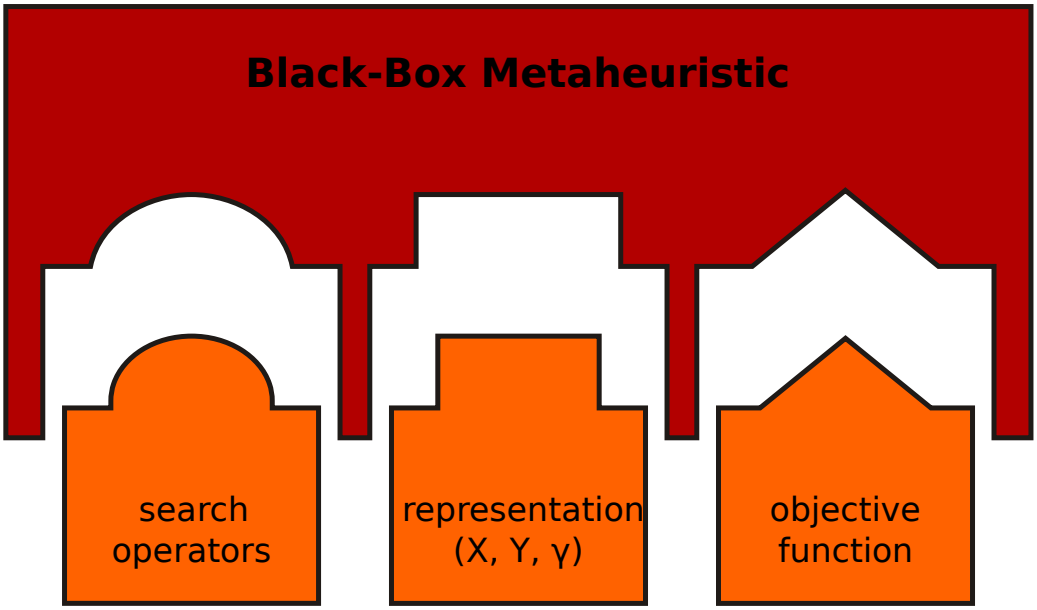

Figure 3.1: The black-box character of many metaheuristics, which can often accept arbitrary search operators, representations, and objective functions.

The concept of general metaheuristics, the idea to attack a very wide class of optimization problems with one basic algorithm design, can be realized when following a *black-box* approach. If we want to have one algorithm that can be applied to all the examples given in in the introduction, then this can best be done if we hide all details of the problems under the hood of the structural elements introduced in chapter 2. For a black-box metaheuristic, it does not matter how the objective function f works. The only thing that matters is that gives a rating of a candidate solution $y \in \mathbb{Y}$ and that smaller ratings are better. For a black-box metaheuristic, it does not matter what exactly the search operators do or even what data structure is used as search space $\mathbb{X}$. It only matters that these operators can be used to get to new points in the search space (which can be mapped to candidate solutions y via a representation mapping γ whose nature is also unimportant for the metaheuristic). Indeed, even the nature of the candidate solutions $y \in \mathbb{Y}$ and the solution space $\mathbb{Y}$ play no big role for black-box optimization methods, as they only work on and explore the search space $\mathbb{X}$. The solution space is relevant for the human operator using the algorithm only, the search space is what the algorithm works on. Thus, a black-box metaheuristic is a general algorithm into which we can plug search operators, representations, and objective functions as needed by a specific application, as illustrated in Figure 3.1. Black-box optimization is the highest level of abstraction on which we can work when trying to solve complex problems.

3.1.5 Putting it Together: A simple API

In our following considerations and discussions of algorithms, we will therefore attempt to define an API for black-box optimization. We will fill the abstract interfaces making up the API with simple and clear implementations of algorithms and their adaptation to the JSSP.

We first need to consider what an optimization needs as input. Obviously, in the most common case, these are all the items we have discussed in the previous section, ranging from the termination criterion over the search operators and the representation mapping to the objective function. Let us therefore define an interface that can provide all these components with corresponding "getter methods". We call this interface `IBlackBoxProcess<X,Y>` from which an excerpt is given in Listing 3.1. It is generic, meaning it allows us to provide a search space $\mathbb{X}$ as type parameter X and a solution space $\mathbb{Y}$ via the type parameter Y.

Listing 3.1 A generic interface for representing black-box processes to an optimization algorithm. (src)

```
1  public interface IBlackBoxProcess<X, Y> extends
2      IObjectiveFunction<X>, ITerminationCriterion, Closeable {
3    public abstract Random getRandom();
4    public abstract ISpace<X> getSearchSpace();
5    public abstract INullarySearchOperator<X>
6        getNullarySearchOperator();
7    public abstract IUnarySearchOperator<X>
8        getUnarySearchOperator();
9    public abstract IBinarySearchOperator<X>
10       getBinarySearchOperator();
11   public abstract double getBestF();
12   public abstract double getGoalF();
13   public abstract void getBestX(X dest);
14   public abstract void getBestY(Y dest);
15   public abstract long getConsumedFEs();
16   public abstract long getLastImprovementFE();
17   public abstract long getMaxFEs();
18   public abstract long getConsumedTime();
19   public abstract long getLastImprovementTime();
20   public abstract long getMaxTime();
21 }
```

If we define such an interface to an optimization algorithm of whatever nature, this also allows us to do one trick: We can directly keep track of the state of the search and remember, e.g., the best solution encountered so far or the time passed. This can then be used to write logging information to a file and to implement the termination criterion. All in all, this interface allows us to

1. provide a random number generator to the algorithm,
2. wrap an objective function f together with a representation mapping γ to allow us to evaluate a point in the search space $x \in \mathbb{X}$ in a single step, effectively performing $f(\gamma(x))$,
3. keep track of the elapsed runtime and FEs as well as when the last improvement was made by updating said information when necessary during the invocations of the "wrapped" objective,
4. keep track of the best points in the search space and solution space so far as well as their associated objective value in special variables by updating them whenever the "wrapped" objective function discovers an improvement (taking care of the issue from Section 3.1.2 automatically),
5. represent a termination criterion based on the above information (e.g., maximum FEs, maximum runtime, reaching a goal objective value), and
6. log the improvements that the algorithm makes to a text file, so that we can use them to make tables and draw diagrams.

Along with the interface class `IBlackBoxProcess`, we also provide a builder for instantiation. The actual implementation behind this interface does not matter here. It is clear what it does, and the actual code is simple and not contributing to the understand of the algorithms or processes. Thus, you do not need to bother with it, just the assumption that an object implementing `IBlackBoxProcess` has the abilities listed above shall suffice here.

When instantiating this interface by our utility functions, besides the search operators, termination criterion, representation mapping, and objective function, we also need to provide the functionality to instantiate and copy the data structures making up the spaces $\mathbb{X}$ and $\mathbb{Y}$. If the black-box optimization algorithm does not make any assumption about the Java `classes` corresponding to these spaces, it needs to be provided with some functionality to instantiate. For this purpose, we add another easy-to-implement and very simple interface, namely `ISpace`, see Listing 3.2.

Listing 3.2 A excerpt of the generic interface `ISpace` for representing basic functionality of search and solution spaces needed by Listing 3.1. (src)

```
1 public interface ISpace<Z> {
2   public abstract Z create();
3   public abstract void copy(Z from, Z to);
4 }
```

Equipped with this, defining an interface for black-box metaheuristics becomes easy: The optimization algorithms themselves then are implementations of the generic interface `IMetaheuristic<X,Y>` given in Listing 3.3. As you can see, this interface only needs a single method, `solve`. This method will use the functionality provided by the `IBlackBoxProcess` handed to it as parameter `process` to generate new points in the search space `X` and sending them to the `evaluate` method of `process`.

This is the core behavior of every basic metaheuristic, and in the rest of this chapter, we will learn how different algorithms realize it.

Listing 3.3 A generic interface of a metaheuristic optimization algorithm. (src)

```
1 public interface IMetaheuristic<X, Y> {
2   public abstract void
3       solve(IBlackBoxProcess<X, Y> process);
4 }
```

Notice that the interface `IMetaheuristic` is, again, generic, allowing us to specify a search space $\mathbb{X}$ as type parameter X and a solution space $\mathbb{Y}$ via the type parameter Y. Whether an implementation of this interface is generic too or whether it ties down X or Y to concrete types will then depend on the algorithms we try to realize. A "fully black-box" metaheuristic may be able to deal with any search- and solution space, as long it is provided with the right operators. However, we could also implement an algorithm specified to numerical problems where $\mathbb{X} \subset \mathbb{R}^n$, by tying down X to `double[]` in the algorithm class specification.

3.1.6 Example: Job Shop Scheduling

What we need to provide for our JSSP example are implementations of our `ISpace` interface for both the search and the solution space, which are given in Listing 3.4 and Listing 3.5, respectively. These classes implement the methods that an `IBlackBoxProcess` implementation needs under the hood to, e.g., copy and store candidate solutions and points in the search space.

Listing 3.4 An excerpt of the implementation of our `ISpace` interface for the search space for the JSSP problem. (src)

```
1 public class JSSPSearchSpace implements ISpace<int[]> {
2   public int[] create() {
3     return new int[this.length];
4   }
5   public void copy(int[] from, int[] to) {
6     System.arraycopy(from, 0, to, 0, this.length);
7   }
8 }
```

With the exception of the search operators, which we will introduce "when they are needed," we have already discussed how the other components needed to solve a JSSP can be realized in Section 2.3.2.1, Section 2.6.2, Section 2.4.2, and Section 2.8.2.

Listing 3.5 An excerpt of the implementation of the `ISpace` interface for the solution space for the JSSP problem. (src)

```
1  public class JSSPSolutionSpace
2      implements ISpace<JSSPCandidateSolution> {
3    public JSSPCandidateSolution create() {
4      return new JSSPCandidateSolution(this.instance.m,
5          this.instance.n);
6    }
7    public void copy(JSSPCandidateSolution from,
8        JSSPCandidateSolution to) {
9      int n = this.instance.n * 3;
10     int i = 0;
11     for (int[] s : from.schedule) {
12       System.arraycopy(s, 0, to.schedule[i++], 0, n);
13     }
14   }
15 }
```

3.2 Random Sampling

If we have our optimization problem and its components properly defined according to chapter 2, then we already have the proper tools to solve the problem. We know

- how a solution can internally be represented as "point" x in the search space $\mathbb{X}$ (Section 2.6),
- how we can map such a point $x \in \mathbb{X}$ to a candidate solution y in the solution space $\mathbb{Y}$ (Section 2.3) via the representation mapping $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ (Section 2.6), and
- how to rate a candidate solution $y \in \mathbb{Y}$ with the objective function f (Section 2.4).

The only question left for us to answer is how to "create" a point x in the search space. We can then apply $\gamma(x)$ and get a candidate solution y whose quality we can assess via $f(y)$. Let us look at the problem as a black box (Section 3.1.4). In other words, we do not really know what "makes" a candidate solution good. Hence, we do not know how to "create" a good solution either. Then the best we can do is just create the solutions randomly.

3.2.1 Ingredient: Nullary Search Operation for the JSSP

For this purpose, we need to implement the nullary search operation from Listing 2.8. We create a new search operator which needs no input and returns a point in the search space. Recall that our representation (Section 2.6.2) requires that each index $i \in 0 \ldots (n-1)$ of the n must occur exactly m

times in the integer array of length $m * n$, where m is the number of machines in the JSSP instance. In Listing 3.6, we achieve this by first creating the sequence $(n-1, n-2, \ldots, 0)$ and then copying it m times in the destination array `dest`. We then randomly shuffle `dest` by applying the Fisher–Yates shuffle algorithm [64,107], which simply brings the array into an entirely random order.

Listing 3.6 An excerpt of the implementation of the nullary search operation interface Listing 2.8 for the JSSP, which will create one random point in the search space. (src)

```
1  public class JSSPNullaryOperator
2      implements INullarySearchOperator<int[]> {
3    public void apply(int[] dest,
4        Random random) {
5  // create first sequence of jobs: n-1, n-2, ..., 0
6      for (int i = this.n; (--i) >= 0;) {
7        dest[i] = i;
8      }
9  // copy this m-1 times: n-1, n-2, ..., 0, n-1, ... 0, n-1, ...
10     for (int i = dest.length; (i -= this.n) > 0;) {
11       System.arraycopy(dest, 0, dest, i, this.n);
12     }
13 // now randomly shuffle the array: create a random sequence
14     RandomUtils.shuffle(random, dest, 0, dest.length);
15   }
16 }
```

By calling the `apply` method of our implemented operator, it will create one random point in the search space. We can then pass this point through the representation mapping that we already implemented in Listing 2.7 and get a Gantt diagram. Easily we then obtain the quality, i.e., makespan, of this candidate solution as the right-most edge of any an job assignment in the diagram, as defined in Section 2.4.2.

3.2.2 Single Random Sample

3.2.2.1 The Algorithm

Now that we have all ingredients ready, we can test the idea. In Listing 3.7, we implement this algorithm (here called `1rs`) which creates exactly one random point x in the search space. It then takes this point and passes it to the evaluation function of our black-box `process`, which will perform the representation mapping $y = \gamma(x)$ and compute the objective value $f(y)$. Internally, we implemented this function in such a way that it automatically remembers the best candidate solution it ever has evaluated. Thus, we do not need to take care of this in our algorithm, which makes the implementation so short.

Listing 3.7 An excerpt of the implementation of an algorithm which creates a single random candidate solution. (src)

```
public class SingleRandomSample<X, Y>
    implements IMetaheuristic<X, Y> {
  public void solve(IBlackBoxProcess<X, Y> process) {
// allocate data structure for holding 1 point from search space
    X x = process.getSearchSpace().create();

// apply nullary operator: fill data structure with a random but
// valid point from the search space
    process.getNullarySearchOperator().apply(x,
        process.getRandom());

// evaluate the point: process.evaluate automatically applies
// representation mapping and calls objective function. the
// objective value is ignored here (not stored anywhere), but
// "process" will remember the best solution, so whoever called
// this "solve" function can obtain the result.
    process.evaluate(x);
  }
}
```

3.2.2.2 Results on the JSSP

Of course, since the algorithm is *randomized*, it may give us a different result every time we run it. In order to understand what kind of solution qualities we can expect, we hence have to run it a couple of times and compute result statistics. We therefore execute our program 101 times and the results are summarized in Table 3.1, which describes them using simple statistics whose meanings are discussed in-depth in Section 4.4.

Table 3.1: The results of the single random sample algorithm `1rs` for each instance $\mathcal{I}$ in comparison to the lower bound $\mathrm{lb}(f)$ of the makespan f over 101 runs: the *best*, *mean*, and median (*med*) result quality, the standard deviation *sd* of the result quality, as well as the median time *med(t)* and FEs *med(FEs)* until a run was finished.

$\mathcal{I}$	$\mathrm{lb}(f)$	best	mean	med	sd	med(t)	med(FEs)
abz7	656	1131	1334	1326	106	0s	1
la24	935	1487	1842	1814	165	0s	1

$\mathcal{I}$	$\mathrm{lb}(f)$	best	mean	med	sd	med(t)	med(FEs)
swv15	2885	5935	6600	6563	346	0s	1
yn4	929	1754	2036	2039	125	0s	1

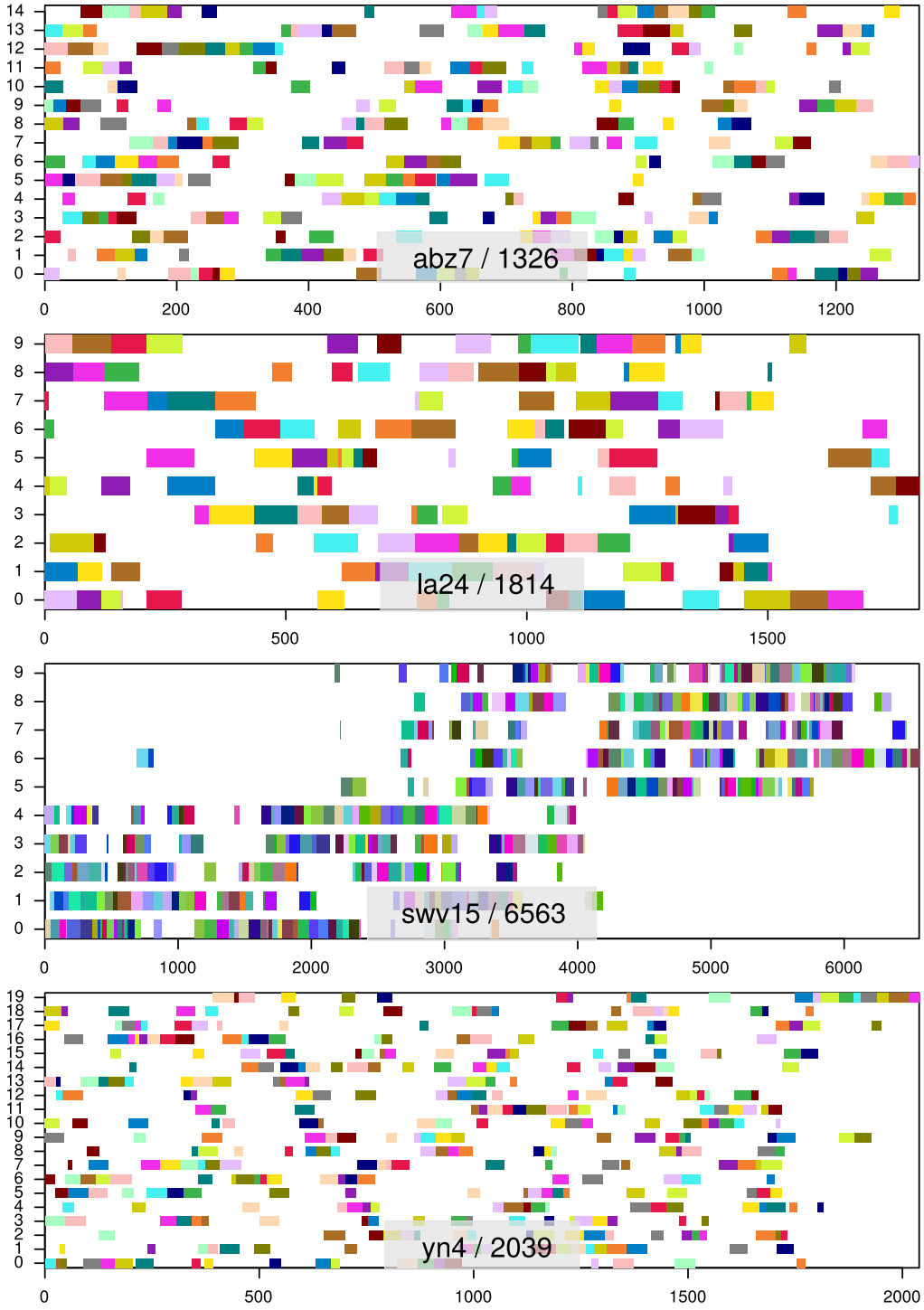

Figure 3.2: The Gantt charts of the median solutions obtained by the `1rs` algorithm. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

What we can find in Table 3.1 is that the makespan of the best solution that any of the 101 runs has delivered for each of the four JSSP instances is between 60% and 100% longer than the lower bound. The arithmetic mean and median of the solution qualities are even between 10% and 20% worse. In the Gantt charts of the median solutions depicted in Figure 3.2, we can find big gaps between the sub-jobs. This is completely reasonable. After all, we just create a single random solution. We can hardly assume that doing all jobs of a JSSP in a random order would even be good idea.

But we also notice more. The median time *t(med)* until the runs stop improving is approximately 0s. The reason is that we only perform one single objective function evaluation per run, i.e., 1 FE. Creating, mapping, and evaluating a solution can be very fast, it can happen within milliseconds. However, we had originally planned to use up to three minutes for optimization. Hence, almost all of our time budget remains unused. At the same time, we already know that that there is a 10-20% difference between the best and the median solution quality among the 101 random solutions we created. The standard deviation sd of the solution quality also is always above 100 time units of makespan. So why don't we try to make use of this variance and the high speed of solution creation?

3.2.3 Random Sampling Algorithm

3.2.3.1 The Algorithm

Random sampling algorithm, also called random search, repeats creating random solutions until the computational budget is exhausted [151]. In our corresponding Java implementation given in Listing 3.8, we therefore only needed to add a loop around the code from the single random sampling algorithm from Listing 3.7.

The algorithm can be described as follows:

1. Set best-so-far objective value to infinity.
2. Create random point x in search space $\mathbb{X}$ (using the nullary search operator).
3. Map the point x to a candidate solution y by applying the representation mapping $y = \gamma(x)$.
4. Compute objective value by invoking the objective function $z = f(y)$.
5. If z is better than best-so-far-objective value, then
   a. Set best-so-far objective value to z.
   b. Store y in a special variable and remember it.
6. If termination criterion is not met, return to point 1.
7. Return best-so-far objective value and best solution to the user.

In actual program code, points 3 to 5 can be encapsulate by a wrapper around the objective function. This reduces a lot of potential programming mistakes and makes the code much shorter. This is what

Listing 3.8 An excerpt of the implementation of the random sampling algorithm which keeps createing random candidate solutions and remembering the best encountered on until the computational budget is exhausted. (src)

```
1  public class RandomSampling<X, Y>
2      implements IMetaheuristic<X, Y> {
3    public void solve(IBlackBoxProcess<X, Y> process) {
4  // allocate data structure for holding 1 point from search space
5      X x = process.getSearchSpace().create();
6  // get nullary search operation for creating random point of X
7      INullarySearchOperator<X> nullary =
8          process.getNullarySearchOperator();
9      Random random = process.getRandom();// get random gen
10
11     do {// repeat until budget exhausted
12       nullary.apply(x, random); // create random point in X
13 // evaluate the point: process.evaluate applies representation
14 // mapping and calls objective function. "process" will remember
15 // the best solution, so whoever called "solve" can obtain it.
16       process.evaluate(x);
17     } while (!process.shouldTerminate()); // do until time is up
18   }
19 }
```

we did with the implementations of the black-box process interface `IBlackBoxProcess` given in Listing 3.1.

3.2.3.2 Results on the JSSP

Let us now compare the performance of this iterated random sampling with our initial method. Table 3.2 shows us that the iterated random sampling algorithm is better in virtually all relevant aspects than the single random sampling method. Its best, mean, and median result quality are significantly better. Since creating random points in the search space is so fast that we can sample many more than 101 candidate solutions, even the median and mean result quality of the `rs` algorithm are better than the best quality obtainable with `1rs`. Matter of fact, each run of our `rs` algorithm can create and test several million candidate solutions within the three minute time window, i.e., perform several million FEs. By remembering the best of millions of created solutions, what we effectively do is we exploit the variance of the quality of the random solutions. As a result, the standard deviation of the results becomes lower. This means that this algorithm has a more reliable performance, we are more likely to get results close to the mean or median performance when we use `rs` compared to `1rs`. Actually, if we

would have infinite time for each run (instead of three minutes), then each run would eventually guess the optimal solutions. The variance would become zero and the mean, median, and best solution would all converge to this global optimum. Alas, we only have three minutes, so we are still far from this goal.

Table 3.2: The results of the single random sample algorithm `1rs` and the random sampling algorithm `rs`. The columns present the problem instance, lower bound, the algorithm, the best, mean, and median result quality, the standard deviation *sd* of the result quality, as well as the median time *med(t)* and FEs *med(FEs)* until the best solution of a run was discovered. The better values are **emphasized**.

$\mathcal{I}$	$\mathrm{lb}(f)$	setup	best	mean	med	sd	med(t)	med(FEs)
abz7	656	1rs	1131	1334	1326	106	**0**s	**1**
		rs	**895**	**945**	**948**	**12**	77s	8’246’019
la24	935	1rs	1487	1842	1814	165	**0**s	**1**
		rs	**1154**	**1206**	**1207**	**15**	81s	17’287’329
swv15	2885	1rs	5935	6600	6563	346	**0**s	**1**
		rs	**4988**	**5165**	**5174**	**49**	85s	5’525’082
yn4	929	1rs	1754	2036	2039	125	**0**s	**1**
		rs	**1459**	**1496**	**1498**	**15**	83s	6’549’694

In Figure 3.3, we now again plot the solutions of median quality, i.e., those which are “in the middle” of the results, quality-wise. The improved performance becomes visible when comparing Figure 3.3 with Figure 3.2. The spacing between the jobs on the machines has significantly reduced. Still, the schedules clearly have a lot of unused time, visible as white space between the sub-jobs on the machines. We are also still relatively far away from the lower bounds of the objective function, so there is lots of room for improvement.

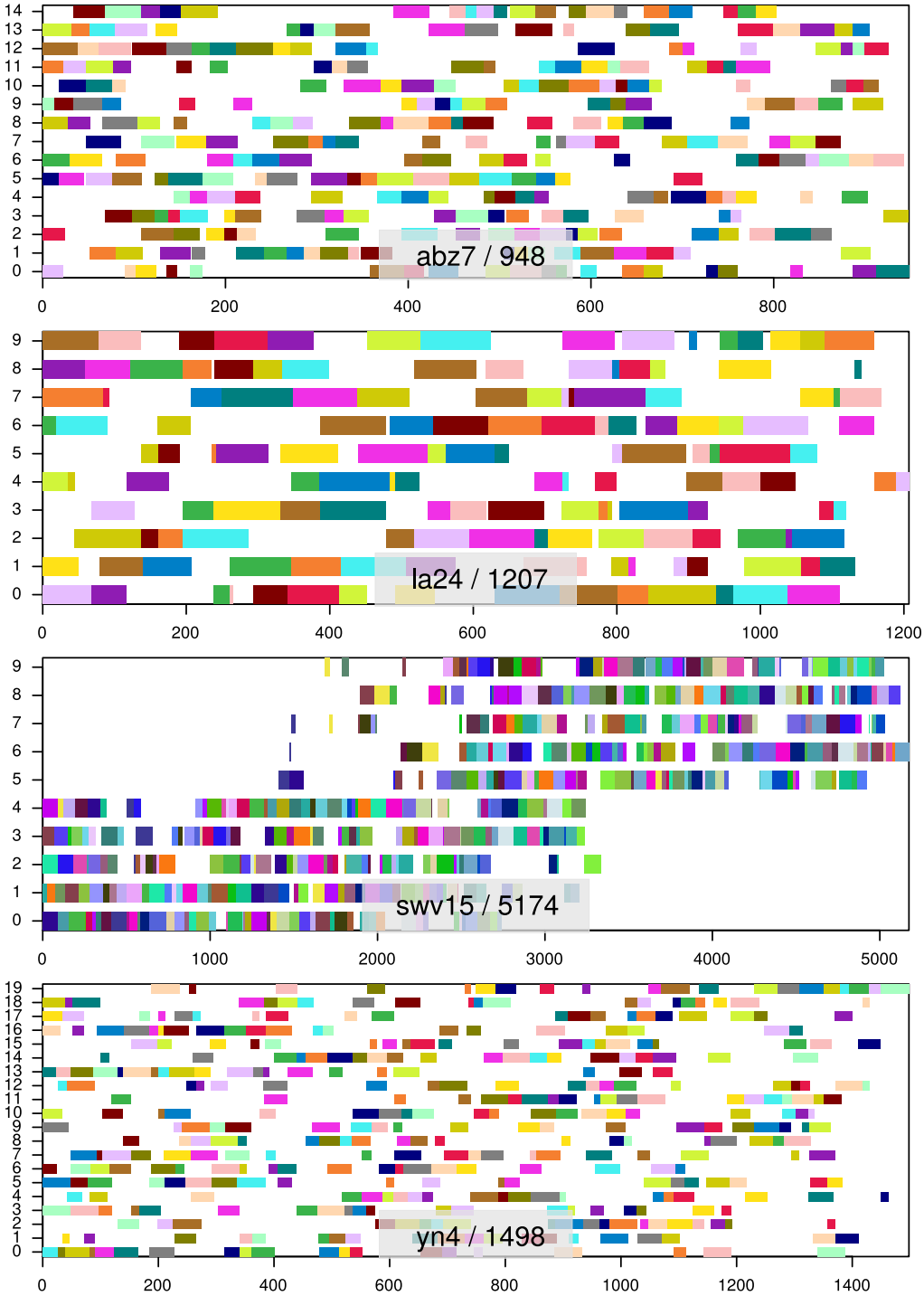

Figure 3.3: The Gantt charts of the median solutions obtained by the `rs` algorithm. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

3.2.3.3 Progress over Time and the Law of Diminishing Returns

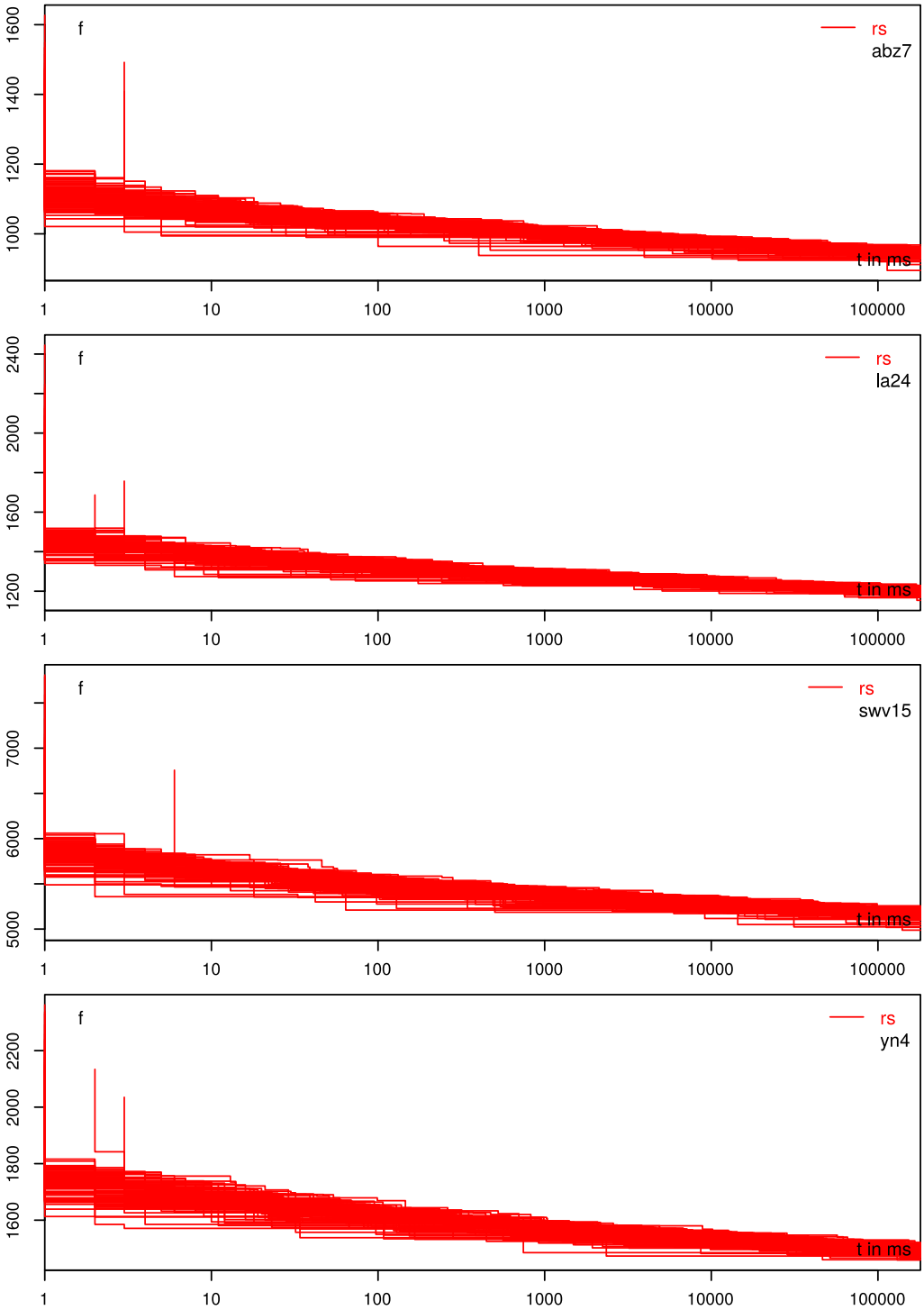

Figure 3.4: The progress of the `rs` algorithm over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis).

Another new feature of our `rs` algorithm is that it truly is an Anytime Algorithm (Section 3.1.1). It begins with an entirely random solution and tries to find better solutions as time goes by. Let us take a look at Figure 3.4, which illustrates how the solution quality of the runs improves over time. At first glance, this figure looks quite nice. For each of the four problem instances we investigate, our algorithms steadily and nicely improve the solution quality. Each single line (one per run) keeps slowly going down, which means that the makespan (objective value) of its best-so-far solution decreases steadily.

However, upon closer inspection, we notice that the time axes in the plots are logarithmically scaled. The first of the equally-spaces axis tick marks is at 1s, the second one at 10s, the third one at 100s, and so on. The progress curves plotted over these logarithmically scaled axes seem to progress more or less like straight linear lines, maybe even slower. A linear progress over a logarithmic time scale could mean, for instance, that we may make the same improvements in the time intervals $1s \ldots 9s$, $10s \ldots 99s$, $100s \ldots 999s$, and so on. In other words: We are getting slower and slower.

This is the first time we witness a manifestation of a very basic law in optimization. When trying to solve a problem, we need to invest resources, be it software development effort, research effort, computational budget, or expenditure for hardware, etc. If you invest a certain amount a of one of these resources, you may be lucky to improve the solution quality that you can get by, say, b units. Investing $2a$ of the resources, however, will rarely lead to an improvement by $2b$ units. Instead, the improvements will become smaller and smaller the more you invest. This is exactly the *Law of Diminishing Returns* [142] known from the field of economics.

And this makes a lot of sense here. On one hand, the maximal possible improvement of the solution quality is bounded by the global optimum – once we have obtained it, we cannot improve the quality further, even if we invest infinitely much of an resource. On the other hand, in most practical problems, the amount of solutions that have a certain quality gets smaller and smaller, the closer said quality is to the optimal one. This is actually what we see in Figure 3.4: The chance of randomly guessing a solution of quality F becomes the smaller the better (smaller) F is.

From the diagrams we can also see that random sampling is not a good method to solve the JSSP. It will not matter very much if we have three minutes, six minutes, or one hour. In the end, the improvements we would get by investing more time would probably become smaller and smaller.

3.3 Hill Climbing

Our first algorithm, random sampling, was not very efficient. It does not make any use of the information it "sees" during the optimization process. Each search step consists of creating an entirely new, entirely random candidate solution. Each search step is thus independent of all prior steps.

Local search algorithms [92,163] offer an alternative. They remember the current best point x_b in the search space $\mathbb{X}$. In every step, a local search algorithm investigates a point x similar to x_b. If it is better, it is accepted as the new best-so-far solution. Otherwise, it is discarded.

Definition 21. Causality means that small changes in the features of an object (or candidate solution) also lead to small changes in its behavior (or objective value).

Local search exploits a property of many optimization problems called *causality* [135,136,166,174]. The idea is that if we have a good candidate solution, then there may exist similar solutions which are better. We hope to find one of them and then continue trying to do the same from there.

3.3.1 Ingredient: Unary Search Operation for the JSSP

So the question arises how we can create a candidate solution which is similar to, but also slightly different from, one we already have? Our search algorithms are working in the search space $\mathbb{X}$. So we need one operation which accepts an existing point $x \in \mathbb{X}$ and produces a slightly modified copy of it as result. In other words, we need to implement a unary search operator!

On a JSSP with m machines and n jobs, our representation $\mathbb{X}$ encodes a schedule as an integer array of length $m * n$ containing each of the job IDs (from $0 \ldots (n-1)$) exactly m times. The sequence in which these job IDs occur then defines the order in which the jobs are assigned to the machines, which is realized by the representation mapping γ (see Listing 2.7).

One idea to create a slightly modified copy of such a point x in the search space would be to simply swap two of the jobs in it. Such a `1swap` operator can be implemented as follows:

1. Make a copy x' of the input point x from the search space.
2. Pick a random index i from $0 \ldots (m * n - 1)$.
3. Pick a random index j from $0 \ldots (m * n - 1)$.
4. If the values at indexes i and j in x' are the same, then go back to point 3. (Swapping the same values makes no sense, since then the value of x' and x would be the same at the end, so also their mappings $\gamma(x)$ and $\gamma(x')$ would be the same, i.e., we would actually not make a "move".)
5. Swap the values at indexes i and j in x'.
6. Return the now modified copy x' of x.

We implemented this operator in Listing 3.9. Notice that the operator is randomized, i.e., applying it twice to the same point in the search space will likely yield different results.

Listing 3.9 An excerpt of the 1swap operator for the JSSP, an implementation of the unary search operation interface Listing 2.9. 1swap swaps two jobs in our encoding of Gantt diagrams. (src)

```
1  public class JSSPUnaryOperator1Swap
2      implements IUnarySearchOperator<int[]> {
3    public void apply(int[] x, int[] dest,
4        Random random) {
5  // copy the source point in search space to the dest
6      System.arraycopy(x, 0, dest, 0, x.length);
7
8  // choose the index of the first sub-job to swap
9      int i = random.nextInt(dest.length);
10     int job_i = dest[i]; // remember job id
11
12     for (;;) { // try to find a location j with a different job
13       int j = random.nextInt(dest.length);
14       int job_j = dest[j];
15       if (job_i != job_j) { // we found two locations with two
16         dest[i] = job_j; // different values
17         dest[j] = job_i; // then we swap the values
18         return; // and are done
19       }
20     }
21   }
22 }
```

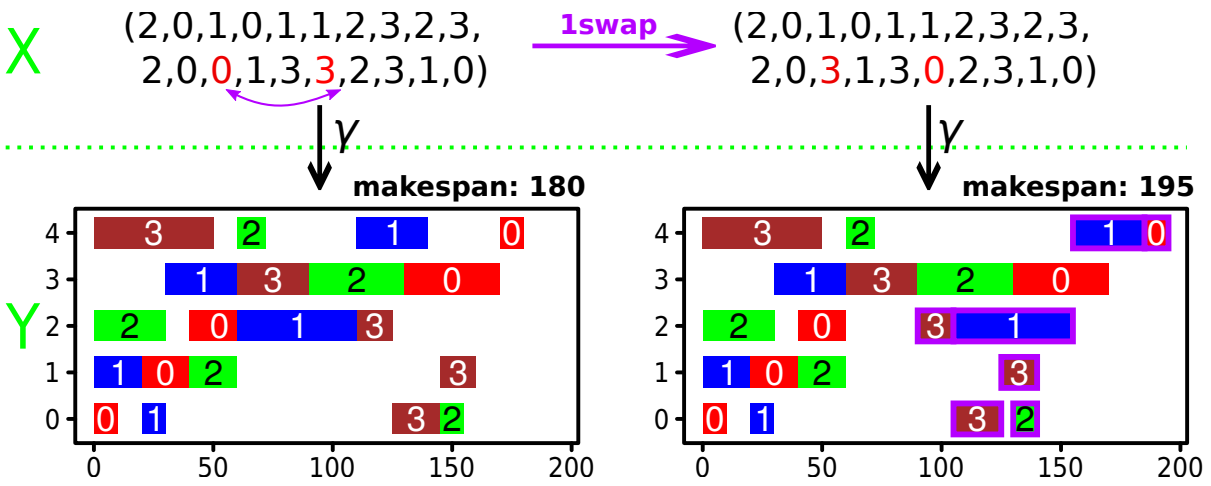

Figure 3.5: An example for the application of 1swap to an existing point in the search space (top-left) for the demo JSSP instance. It yields a slightly modified copy (top-right) with two jobs swapped. If we map these to the solution space (bottom) using the representation mapping γ, the changes marked with violet frames occur (bottom-right).

In Figure 3.5, we illustrate the application of this operator to one point x in the search space for our demo JSSP instance. It swaps the two jobs at index $i = 10$ and $j = 15$ of x. In the new, modified copy x', the jobs 3 and 0 at these indices have thus traded places. The impact of this modification becomes visible when we map both x and x' to the solution space using the representation mapping γ. The 3 which has been moved forward now means that job 3 will be scheduled before job 1 on machine 2. As a result, the last two sub-jobs of job 3 can now finish earlier on machines 0 and 1, respectively. However, time is wasted on machine 2, as we first need to wait for the first two sub-jobs of job 3 to finish before we can execute it there. Also, job 1 finishes now later on that machine, which also delays its last sub-job to be executed on machine 4. This pushes back the last sub-job of job 0 (on machine 4) as well. The new candidate solution $\gamma(x')$ thus has a longer makespan of $f(\gamma(x')) = 195$ compared to the original solution with $f(\gamma(x)) = 180$.

In other words, our application of `1swap` in Figure 3.5 has led us to a worse solution. This will happen most of the time. As soon as we have a good solution, the solutions similar to it tend to be worse. However, if we would have been at x' instead, an application of `1swap` could well have resulted in x. Often, the chance to find a really good solution by iteratively sampling the neighborhoods of good solutions is higher than trying to randomly guessing them (as `rs` does) even if most of our samples are worse.

3.3.2 Stochastic Hill Climbing Algorithm

3.3.2.1 The Algorithm

Stochastic Hill Climbing](http://en.wikipedia.org/wiki/Hill_climbing) [140,150,163] is the simplest implementation of local search. It is also sometimes called localized random search [151]. It proceeds as follows:

1. Create random point x in search space $\mathbb{X}$ (using the nullary search operator).
2. Map the point x to a candidate solution y by applying the representation mapping $y = \gamma(x)$.
3. Compute the objective value by invoking the objective function $z = f(y)$.
4. Store x in the variable x_b and z in z_b.
5. Repeat until the termination criterion is met:
   a. Apply the unary search operator to x_b to get the slightly modified copy x' of it.
   b. Map the point x' to a candidate solution y' by applying the representation mapping $y' = \gamma(x')$.
   c. Compute the objective value z' by invoking the objective function $z' = f(y')$.
   d. If $z' < z_b$, then store x' in the variable x_b and z' in z_b.
6. Return best-so-far objective value and best solution to the user.

This algorithm is implemented in Listing 3.10 and we will refer to it as hc.

Listing 3.10 An excerpt of the implementation of the Hill Climbing algorithm, which remembers the best-so-far solution and tries to find better solutions in its neighborhood. (src)

```
1  public class HillClimber<X, Y>
2      implements IMetaheuristic<X, Y> {
3    public void solve(IBlackBoxProcess<X, Y> process) {
4  // init local variables x_cur, x_best, nullary, unary, random
5  // create starting point: a random point in the search space
6      nullary.apply(x_best, random); // put random point in x_best
7      double f_best = process.evaluate(x_best); // map & evaluate
8
9      while (!process.shouldTerminate()) {// repeat until budget
10                                         // exhausted
11 // create a slightly modified copy of x_best and store in x_cur
12       unary.apply(x_best, x_cur, random);
13 // map x_cur from X to Y and evaluate candidate solution
14       double f_cur = process.evaluate(x_cur);
15       if (f_cur < f_best) { // we found a better solution
16 // remember best objective value and copy x_cur to x_best
17         f_best = f_cur;
18         process.getSearchSpace().copy(x_cur, x_best);
19       } // otherwise, i.e., f_cur >= f_best: just forget x_cur
20     } // until time is up
21   } // process will have remembered the best candidate solution
22 }
```

3.3.2.2 Results on the JSSP

We now apply our hc algorithm together with the 1swap to the JSSP. We will refer to this setup as hc_1swap and present its results with those of rs in Table 3.3.

Table 3.3: The results of the hill climber `hc_1swap` in comparison with those of random sampling algorithm `rs`. The columns present the problem instance, lower bound, the algorithm, the best, mean, and median result quality, the standard deviation *sd* of the result quality, as well as the median time *med(t)* and FEs *med(FEs)* until the best solution of a run was discovered. The better values are **emphasized**.

$\mathcal{I}$	$\mathrm{lb}(f)$	setup	best	mean	med	sd	med(t)	med(FEs)
abz7	656	hc_1swap	**717**	**800**	**798**	28	**0**s	16’978
		rs	895	945	948	**12**	77s	8’246’019
la24	935	hc_1swap	**999**	**1095**	**1086**	56	**0**s	6612
		rs	1154	1206	1207	**15**	81s	17’287’329
swv15	2885	hc_1swap	**3837**	**4108**	**4108**	137	**1**s	104’598
		rs	4988	5165	5174	**49**	85s	5’525’082
yn4	929	hc_1swap	**1109**	**1222**	**1220**	48	**0**s	31’789
		rs	1459	1496	1498	**15**	83s	6’549’694

The hill climber outperforms random sampling in almost all aspects. It produces better mean, median, and best solutions. Actually, its median and mean solutions are better than the best solutions discovered by `rs`. Furthermore, it finds its solutions much much faster. The time consumed until convergence is not more than one seconds and the number of consumed FEs to find the best solutions per run is between 7000 and 105’000, i.e., between one 50$^{\text{th}}$ and one 2500$^{\text{th}}$ of the number of FEs needed by `rs`.

It may be interesting to know that this simple `hc_1swap` algorithm can already achieve some remotely acceptable performance. For instance, on instance `abz7`, it delivers better best and mean results than all four Genetic Algorithms (GAs) presented in [100] and on `la24`, only one of the four (GA-PR) has a better best result and all lose in terms of mean result. On this instance, `hc_1swap` finds a better best solution than all six GAs in [2] and better mean results than four of them. In Section 3.4, we will later introduce Evolutionary Algorithms, to which GAs belong.

The Gantt charts of the median solutions of `hc_1swap` are illustrated in Figure 3.6 are also more compact than those in Figure 3.3.

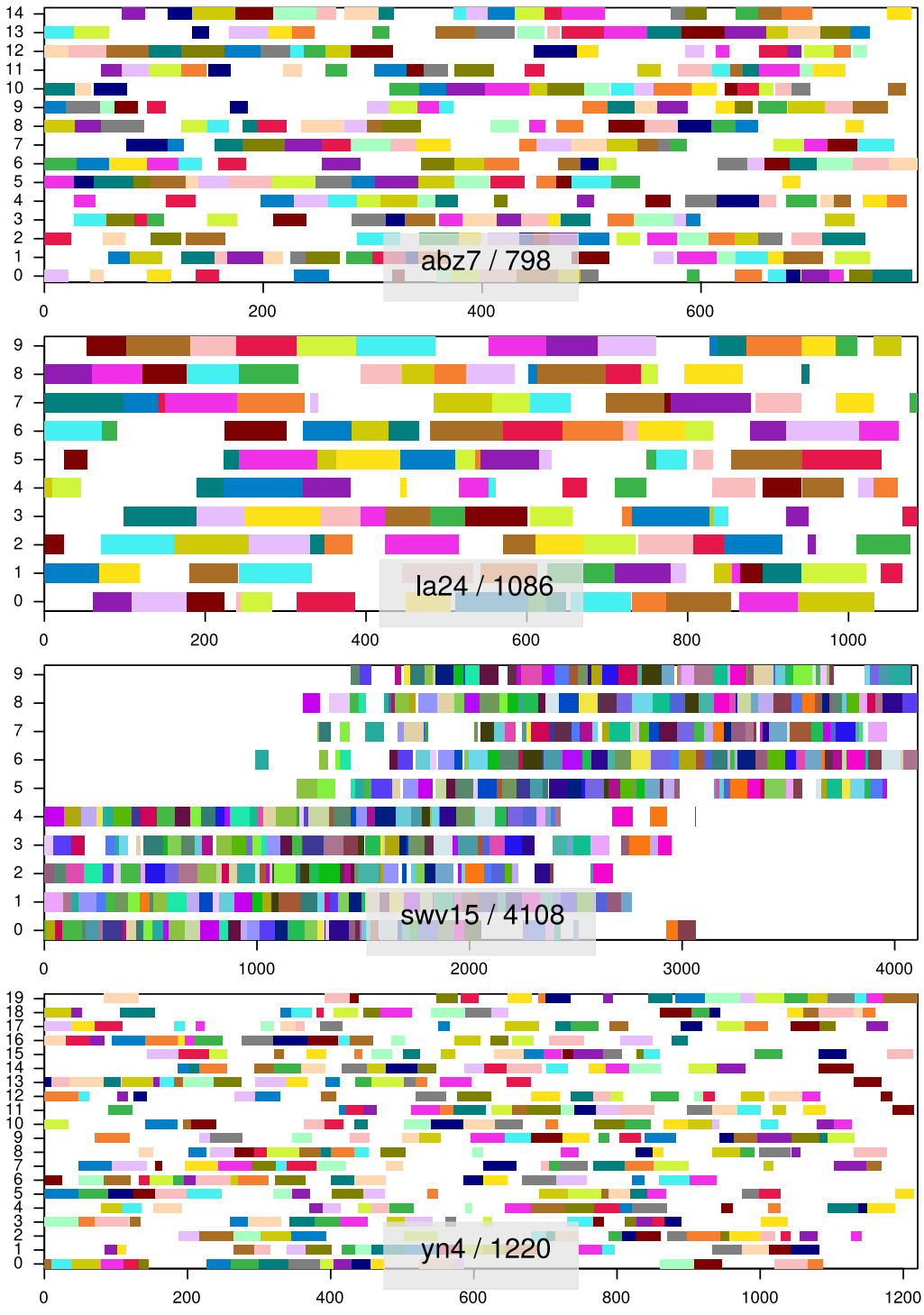

Figure 3.6: The Gantt charts of the median solutions obtained by the `hc_1swap` algorithm. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

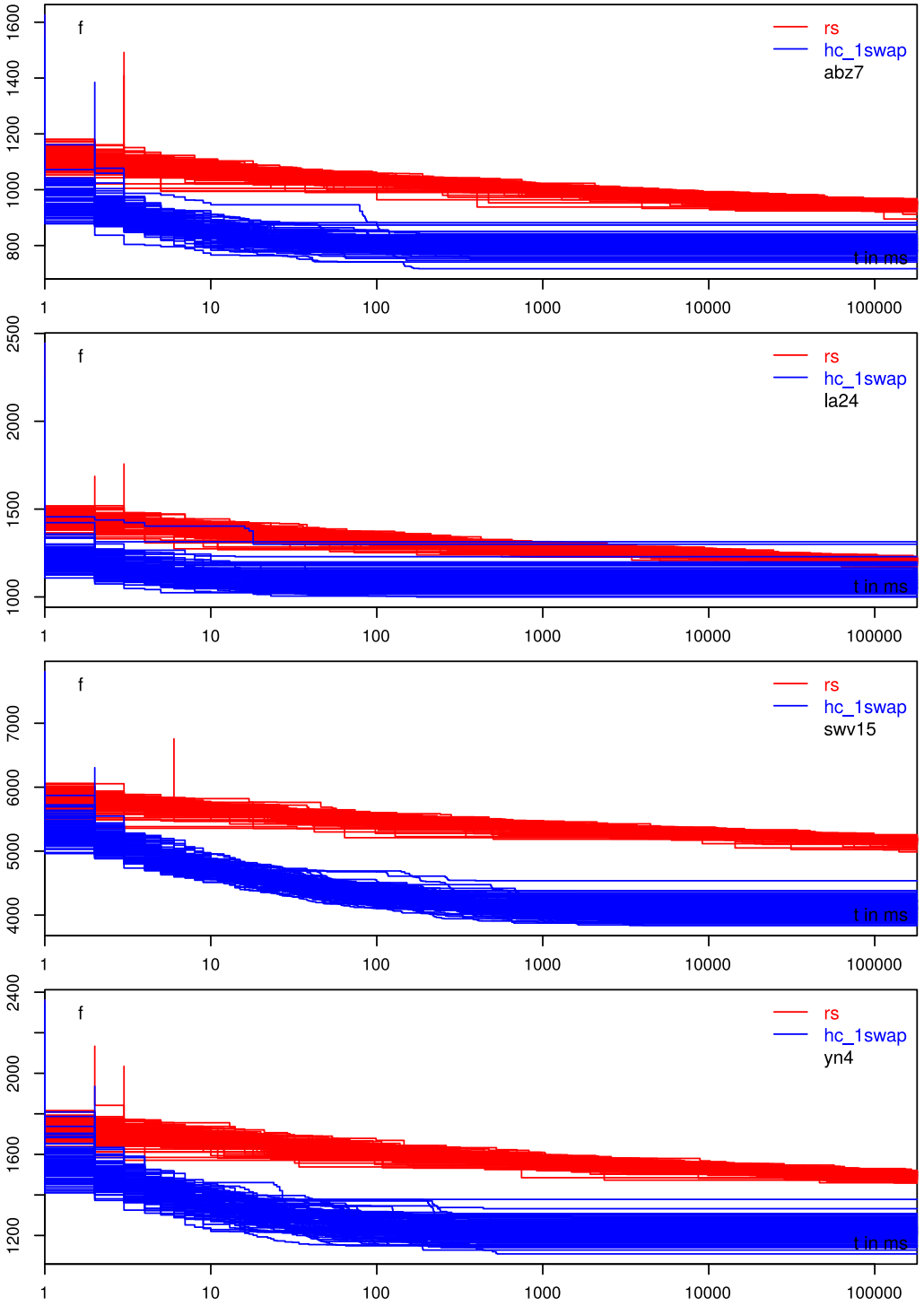

Figure 3.7: The progress of the `hc_1swap` and `rs` algorithm over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis).

Figure 3.7 shows how both hc_1swap and rs progress over time. It should be noted that I designed the experiments in such a way that there were 101 different initial solutions and the runs of the hill climber and random sampling started *at the same points*. On the logarithmically scaled plots, this is almost invisible. The runs of the two different algorithms separate almost immediately. We already knew from Table 3.3 that hc_1swap converges very quickly. After initial phases with quick progress, it stops making any further progress. With the exception of instance la24, there is much space between the runs of rs and hc_1swap. We can also see again that there is more variance in the end results of hc_1swap compared to those of rs, as they are spread wider in the vertical direction.

3.3.3 Stochastic Hill Climbing with Restarts

We now are in the same situation as with the 1rs algorithm: There is some variance between the results and most of the "action" takes place in a short time compared to our total computational budget (1s vs. 3min). Back in Section 3.2.3 we made use of this situation by simply repeating 1rs until the computational budget was exhausted, which we called the rs algorithm. Now the situation is a bit different, however. 1rs creates exactly one solution and is finished, whereas our hill climber does not actually finish. It keeps creating modified copies of the current best solution, only that these happen to not be better. The algorithm has converged into a *local optimum*.

Definition 22. A *local optimum* is a point $x^\times$ in the search space which maps to a better candidate solution than any other points in its neighborhood (see Definition 17).

Definition 23. An optimization process has prematurely converged if it has not yet discovered the global optimum but can no longer improve its approximation quality. [166,174]

Of course, our hill climber does not really know that it is trapped in a local optimum, that it has *prematurely converged*. However, we can try to guess it: If there has not been any improvement for many steps, then the current-best candidate solution is probably a local optimum. If that happens, we just restart at a new random point in the search space. Of course, we will remember the **best ever encountered** candidate solution over all restarts and return it to the user in the end.

3.3.3.1 The Algorithm

1. Set counter C of unsuccessful search steps to 0, initialize limit L for the maximally allowed unsuccessful search steps.
2. Set the overall-best objective value z_B to infinity and the overall-best candidate solution y_B to NULL.
3. Create random point x in search space $\mathbb{X}$ (using the nullary search operator).
4. Map the point x to a candidate solution y by applying the representation mapping $y = \gamma(x)$.

5. Compute the objective value by invoking the objective function $z = f(y)$.
6. Store x in the variable x_b and z in z_b.
7. If $z_b < z_B$, then set z_B to z_b and store $y_B = \gamma x$.
8. Repeat until the termination criterion is met:
   a. Apply the unary search operator to x_b to get the slightly modified copy x' of it.
   b. Map the point x' to a candidate solution y' by applying the representation mapping $y' = \gamma(x')$.
   c. Compute the objective value z' by invoking the objective function $z' = f(y')$.
   d. If $z' < z_b$, then
      i. store x' in the variable x_b,
      ii. z' in z_b, and
      iii. set C to 0.
      iv. If $z' < z_B$, then set z_B to z' and store $y_B = \gamma x'$.

      otherwise
      i. increment C by 1
      ii. if $C \geq L$ then
         (1) Maybe: increase L (see later).
         (2) Go back to step 3.
9. Return **best ever encountered** objective value z_B and solution y_B to the user.

Now this algorithm – implemented in Listing 3.11 – is a bit more elaborate. Basically, we embedd the original hill climber into a loop. This hill climber will stop after a certain number of unsuccessful search steps, which then leads to a new round in the outer loop. The problem that we have is that we do not know which "certain number" is right. If we pick it too low, then the algorithm will restart before it actually converges to a local optimum. If we pick it too much, we waste runtime and do fewer restarts than what we could do. To deal with this dilemma, we can slowly increase the number of allowed unsuccessful search moves.

3.3.3.2 Results on the JSSP

In Table 3.4 we present the performance indicators of the two versions of our hill climber with restarts in comparison with the plain hill climber. We implement `hcr_256_1swap`, which begins at a new random point in the search space after $L = 256$ applications of the unary operator to the same current-best solution did not yield any improvement. `hcr_256+5%_1swap` does the same, but increases L by 5% after each restart, i.e., initially waits 256 steps, then $round(1.05 * 256) = 267$ steps, then 280, and so on. Of course, the actual search procedure of both algorithms is still the same as the one of the

plain hill climber hc_1swap. What we can expect is therefore mainly an utilization of the variance in the end results and the time "wasted" after hc_1swap has converged.

Table 3.4: The results of the hill climber hc_1swap with restarts. hcr_256_1swap restarts after 256 unsuccessful search moves, hcr_256+5%_1swap does the same but increases the allowed number of unsuccessful moves by 5% after each restart. The columns present the problem instance, lower bound, the algorithm, the best, mean, and median result quality, the standard deviation *sd* of the result quality, as well as the median time *med(t)* and FEs *med(FEs)* until the best solution of a run was discovered. The better values are **emphasized**.

$\mathcal{I}$	$\mathrm{lb}(f)$	setup	best	mean	med	sd	med(t)	med(FEs)
abz7	656	hc_1swap	**717**	800	798	28	**0**s	16'978
		hcr_256_1swap	738	765	766	**7**	82s	22'881'557
		hcr_256+5%_1swap	723	**742**	**743**	7	21s	5'681'591
la24	935	hc_1swap	999	1095	1086	56	**0**s	6612
		hcr_256_1swap	975	1001	1002	**6**	91s	49'588'742
		hcr_256+5%_1swap	**970**	**997**	**998**	9	6s	3'470'368
swv15	2885	hc_1swap	3837	4108	4108	137	**1**s	104'598
		hcr_256_1swap	4069	4173	4177	**32**	92s	15'351'798
		hcr_256+5%_1swap	**3701**	**3850**	**3857**	40	60s	9'874'102
yn4	929	hc_1swap	1109	1222	1220	48	**0**s	31'789
		hcr_256_1swap	1153	1182	1184	**12**	90s	18'843'991
		hcr_256+5%_1swap	**1095**	**1129**	**1130**	14	22s	4'676'669

Table 3.4 shows us that the restarted algorithms offer improved median and mean results. The standard deviation of their end results is also reduced, so they have become more reliable. Also, their median time until they converge is now higher, which means that we make better use of our computational budget. The best solution from all 101 runs they discover does not necessarily improve, which makes sense because they are still essentially the same algorithms. Slowly increasing the time until restart turns out to be a good idea: hcr_256+5%_1swap outperforms hcr_256_1swap in almost all aspects.

This could also mean that waiting 256 steps until a restart is not enough, of course. If this was an actual, practical application scenario we should experiment with more settings. For the sake of demonstrating

the basic ideas in this book, however, we will not do that.

The best result of our still quite basic `hcr_256_1swap` and `hcr_256+5%_1swap` for instance `la24` can both surpass the best result (982) delivered by the Gray Wolf Optimization algorithm in [98].

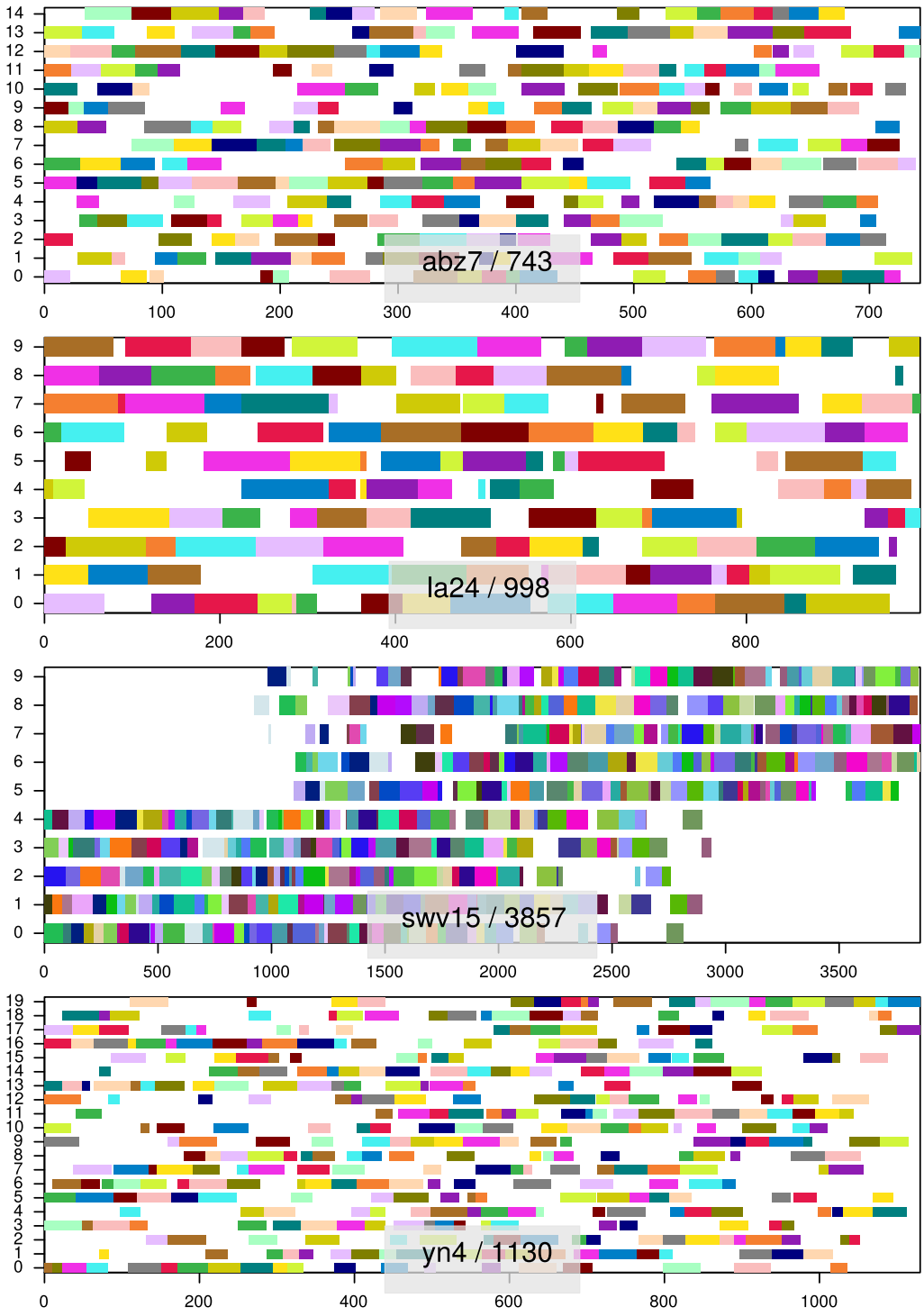

Figure 3.8: The Gantt charts of the median solutions obtained by the `hcr_256+5%_1swap` algorithm. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

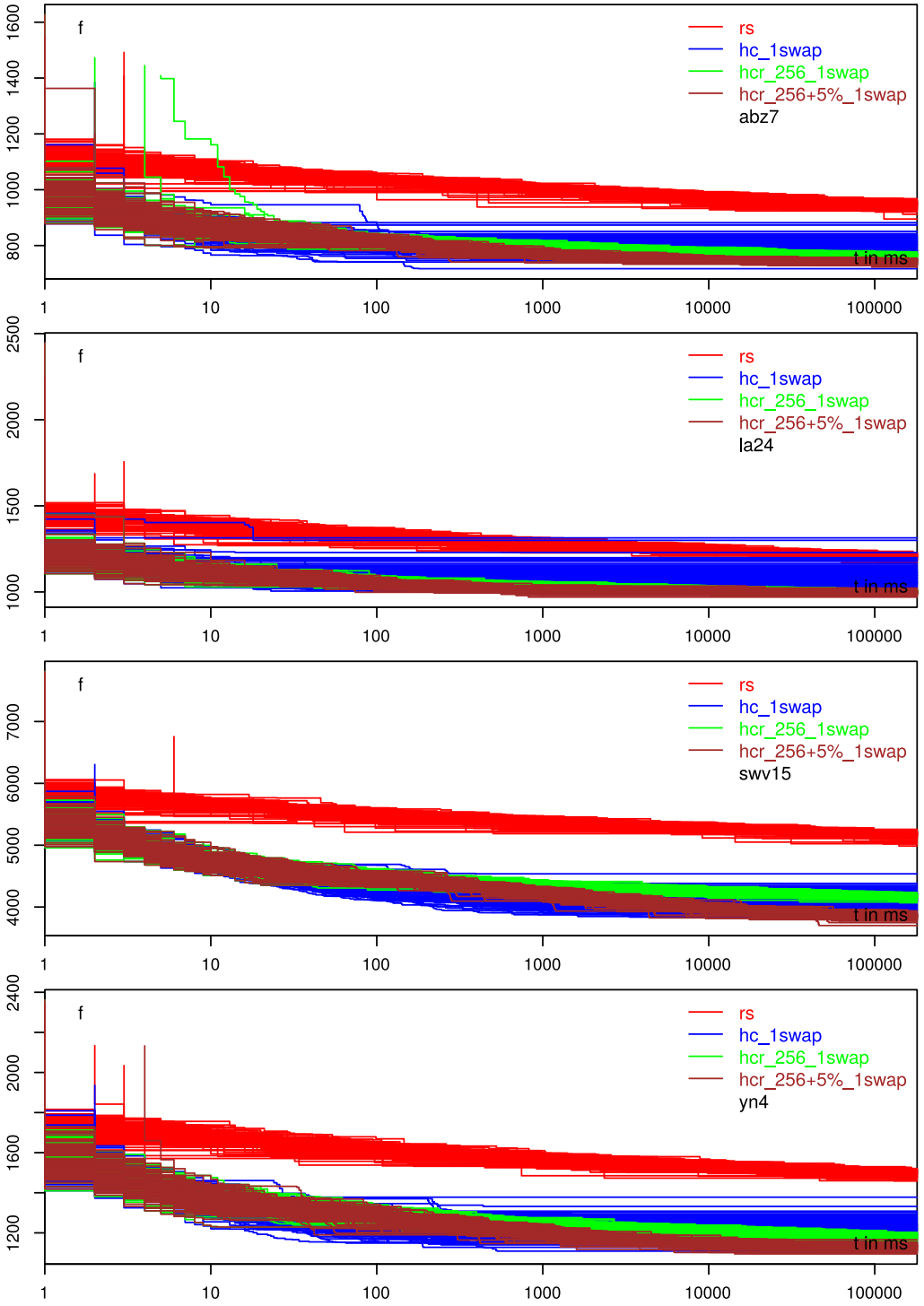

Figure 3.9: The progress of the algorithms `rs`, `hc_1swap`, `hcr_256_1swap`, and `hcr_256+5%_1swap` over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis).

The average solutions discovered by `hcr_256+5%_1swap`, illustrated in Figure 3.8, again show less wasted time. The scheduled jobs again move a bit closer together.

From the progress diagrams plotted in Figure 3.8, we can see that the algorithm versions with restart initially behave exactly the same as the "normal" hill climber. They should do that, because until they do their first restart, the are identical to `hc_1swap`. However, when `hc_1swap` has converged and stops making improvements, `hcr_256_1swap` and `hcr_256+5%_1swap` still continue to make progress. On all problem instances except `la24`, `hcr_256+5%_1swap` provides visible better end results compared to `hcr_256_1swap` as well, confirming the findings from Table 3.4.

Listing 3.11 An excerpt of the implementation of the Hill Climbing algorithm with restarts, which remembers the best-so-far solution and tries to find better solutions in its neighborhood but restarts if it seems to be trapped in a local optimum. (src)

```
1  public class HillClimberWithRestarts<X, Y>
2      implements IMetaheuristic<X, Y> {
3    public void solve(IBlackBoxProcess<X, Y> process) {
4  // omitted: initialize local variables x_cur, x_best, nullary,
5  // unary,random, failsBeforeRestart, and failCounter=0
6      while (!(process.shouldTerminate())) { // outer loop: restart
7        nullary.apply(x_best, random); // sample random solution
8        double f_best = process.evaluate(x_best); // evaluate it
9
10       innerHC: do {// repeat until budget exhausted or got stock
11         unary.apply(x_best, x_cur, random); // try to improve
12         ++failCounter;// increase step counter
13         double f_cur = process.evaluate(x_cur); // evaluate
14         if (f_cur < f_best) { // we found a better solution
15           f_best = f_cur; // remember best quality
16           process.getSearchSpace().copy(x_cur, x_best); // copy
17           failCounter = 0L; // reset number of unsuccessful steps
18         } else { // ok, we did not find an improvement
19           if (failCounter >= failsBeforeRestart) {
20             // increase steps before restart
21             failsBeforeRestart = Math.max(failsBeforeRestart,
22                 Math.round(failsBeforeRestart
23                     * (1d + this.increaseFactor)));
24             failCounter = 0L;
25             break innerHC; // jump back to outer loop for restart
26           }
27         }
28       } while (!process.shouldTerminate()); // until time is up
29     }
30   }
31 }
```

3.3.4 Hill Climbing with a Different Unary Operator

With our restart method could significantly improve the results of the hill climber. It directly addressed the problem of premature convergence, but it tried to find a remedy for its symptoms, not its cause.

One cause for this problem in our hill climber is the design of unary operator. `1swap` will swap two jobs in an encoded solution. Since the solutions are encoded as integer arrays of length $m * n$, there are $m * n$ choices to pick the index of the first job to be swapped. Since we swap only with *different* jobs and each job appears m times in the encoding, this leaves $m * (n - 1)$ choices for the second swap index. We can also ignore equivalent swaps, e.g., exchanging the jobs at indexes $(10, 5)$ and $(5, 10)$ would result in the same outcome. In total, from any given point in the search space, `1swap` may reach $0.5 * m * n * m * (n - 1) = 0.5 * (m^2n^2 - n)$ different other points (some of which may still actually encode the same candidate solutions). These are only tiny fractions of the big search space (remember Table 2.3?).

This has two implications:

1. The chance of premature convergence for a hill climber applying this operator is relatively high, since the neighborhoods are relatively small. If the neighborhood spanned by the operator was larger, it would contain more, potentially better solutions. Hence, it would take longer for the optimization process to reach a point where no improving move can be discovered anymore.
2. Assume that there is no better solution in the `1swap` neighborhood of the current best point in the search space. There might still be a much better, similar solution which could, for instance, require swapping three or four jobs – but the algorithm will never find it, because it can only swap two jobs. If the search operator would permit such moves, the hill climber may discover this better solution.

Now we need to think about how we could define a new unary operator which can access a larger neighborhood. Here we first should consider the extreme cases. On the one hand, we have `1swap` which samples from a relatively small neighborhood. The other extreme could be to use our nullary operator as unary operator: It would return an entirely random point from the search space $\mathbb{X}$ and ignore its input. It would span $\mathbb{X}$ as its neighborhood and uniformly sample from it, effectively turning the hill climber into random sampling. From this thought experiment we know that unary operators which indiscriminately sample from very large neighborhoods are not very good ideas, as they are "too random." They also make less use of the causality of the search space, as they make large steps and their produced outputs are very different from their inputs. What we would like is an operator that often creates outputs very similar to its input (like `1swap`), but also from time to time samples points a bit farther away in the search space.

3.3.4.1 Second Unary Search Operator for the JSSP

We define the nswap operator for the JSSP as follows and implement it in Listing 3.12:

1. Make a copy x' of the input point x from the search space.
2. Pick a random index i from $0 \ldots (m * n - 1)$.
3. Store the job-id at index i in the variable f for holding the very first job, i.e., set $f = x'_i$.
4. Set the job-id variable l for holding the last-swapped-job to x'_i as well.
5. Repeat
   a. Decide whether we should continue the loop *after* the current iteration (TRUE) or not (FALSE) with equal probability and remember this decision in variable n.
   b. Pick a random index j from $0 \ldots (m * n - 1)$.
   c. If $l = x'_j$, go back to point b.
   d. If $f = x_j$ *and* we will *not* do another iteration ($n = FALSE$), go back to point b.
   e. Store the job-id at index j in the variable l.
   f. Copy the job-id at index j to index i, i.e., set $x'_i = x'_j$.
   g. Set $i = j$.
6. If we should do another iteration ($n = TRUE$), go back to point 5.
7. Store the first-swapped job-id f in x'_i.
8. Return the now modified copy x' of x.

The idea of this operator is that we will perform at least one iteration of the loop (*point 5*). If we would do exactly one iteration, then we would pick two indices i and j, then we will pick two indices where different job-ids are stored, as l must be different from f (*point c* and *d*). We would then would swap the jobs at these indices (*points f*, *g*, and *7*). So in the case of exactly one iteration of the main loop, this operator behaves exactly the same as 1swap. This takes place with a probability of 0.5 (*point a*).

If we do two iterations, i.e., pick TRUE the first time we arrive at *point a* and FALSE the second time, then we swap three job ids-instead. Let us say we picked indices α at *point 2*, β at *point b*, and γ when arriving the second time at *b*. We will store the job-id originally stored at index β at index α, the job originally stored at index γ at index β, and the job-id from index γ to index α. *Condition c* prevents index β from referencing the same job-id as index α and index γ from referencing the same job-id as what was originally stored at index β. *Condition d* only applies in the last iteration and prevents γ from referencing the original job-id at α.

This three-job swap will take place with probability $0.5 * 0.5 = 0.25$. Similarly, a four-job-swap will happen with half of that probability, and so on. In other words, we have something like a Bernoulli process, where we decide whether or not to do another iteration by flipping a fair coin, where each choice has probability 0.5. The number of iterations will therefore be geometrically distributed with an

expectation of two job swaps. Of course, we only have m different job-ids in a finite-length array x', so this is only an approximation. Generally, this operator will most often apply small changes and sometimes bigger steps. The bigger the search step, the less likely will it be produced. The operator therefore can make use of the *causality* while – at least theoreticaly – being able to escape from any local optimum.

3.3.4.2 Results on the JSSP

Let us now compare the end results that our hill climbers can achieve using either the `1swap` or the new `nswap` operator after three minutes of runtime on my little laptop computer in Table 3.5.

Table 3.5: The results of the hill climbers `hc_1swap` and `hc_nswap` with and without restarts. The columns present the problem instance, lower bound, the algorithm, the best, mean, and median result quality, the standard deviation *sd* of the result quality, as well as the median time *med(t)* and FEs *med(FEs)* until the best solution of a run was discovered. The better values are **emphasized**.

$\mathcal{I}$	$\mathrm{lb}(f)$	setup	best	mean	med	sd	med(t)	med(FEs)
abz7	656	hc_1swap	717	800	798	28	**0**s	16978
		hc_nswap	724	757	757	17	30s	8145596
		hcr_256_1swap	738	765	766	7	82s	22881557
		hcr_256_nswap	756	774	774	**6**	101s	27375920
		hcr_256+5%_1swap	723	742	743	7	21s	5681591
		hcr_256+5%_nswap	**707**	**733**	**734**	7	64s	17293038
la24	935	hc_1swap	999	1095	1086	56	**0**s	6612
		hc_nswap	**945**	1017	1015	29	21s	11123744
		hcr_256_1swap	975	1001	1002	**6**	91s	49588742
		hcr_256_nswap	986	1008	1008	7	100s	52711888
		hcr_256+5%_1swap	970	997	998	9	6s	3470368
		hcr_256+5%_nswap	**945**	**981**	**984**	9	57s	29246097
swv15	2885	hc_1swap	3837	4108	4108	137	**1**s	104598
		hc_nswap	**3599**	3867	3859	113	70s	11559667
		hcr_256_1swap	4069	4173	4177	32	92s	15351798

$\mathcal{I}$	$\mathrm{lb}(f)$	setup	best	mean	med	sd	med(t)	med(FEs)
		hcr_256_nswap	4118	4208	4214	**29**	95s	15746919
		hcr_256+5%_1swap	3701	3850	3857	40	60s	9874102
		hcr_256+5%_nswap	3645	**3804**	**3811**	44	91s	14907737
yn4	929	hc_1swap	1109	1222	1220	48	**0**s	31789
		hc_nswap	1087	1160	1156	33	63s	13111115
		hcr_256_1swap	1153	1182	1184	12	90s	18843991
		hcr_256_nswap	1163	1198	1199	**11**	91s	18700214
		hcr_256+5%_1swap	1095	1129	1130	14	22s	4676669
		hcr_256+5%_nswap	**1081**	**1117**	**1119**	14	55s	11299461

When comparing two setups which only differ in the unary operator, we find that in most cases, nswap performs better when applied without restarts (hc_*) or with restarts after increasing periods of time (hcr_256+5%_*). Indeed, all the best results we have obtained so far stem from nswap setups and the setups with best mean and median performance use nswap as well. When being restarted, the standard deviations of their results are similar to those with 1swap, meaning that these setups are similarly reliable. Interestingly, for instance la24, the makespan of the best discovered solution is now only 1% longer than the lower bound (945 vs. 935). For instance swv15, however, there is still a 20% gap.

As can be seen when comparing the hill climbers without restart, the nswap operator needs longer to converge because half of its steps are bigger than those of 1swap. It utilizes the causality in the search space a bit less. This may be the reason why hcr_256_1swap tends to be better than hcr_256_nswap while hcr_256+5%_nswap outperforms hcr_256+5%_1swap the restarts happen too early for nswap. The setups with nswap tend to converge later, both in terms of runtime med(t) and med(FEs).

Figure 3.10 illustrates the progress of the hill climbers with the 1swap and nswap operators. While there is quite an improvement when comparing the non-restarting algorithms, the difference between hcr_256+5%_1swap and hcr_256+5%_nswap does not look that big. From Table 3.5 we know that the nswap operator here can squeeze out around 1% of solution quality. The Gantt charts of the median solutions obtained with hcr_256+5%_nswap setup, illustrated in Figure 3.11, do thus look similar to those obtained with hcr_256+5%_1swap in Figure 3.8, although there are some slight differences. Although 1% savings in makespan does not look much, but in a practical application, even a small improvement can mean a lot of benefit.

Both restarts and the idea of allowing bigger search steps with small probability are intended to decrease the chance of premature convergence, while the latter one also can investigate more solutions similar to the current best one. We have seen that both measures work separately and in this case, we were lucky that they also work hand-in-hand. This is not necessarily always the case, in optimization sometimes two helpful measures combined may lead to worse results, as we can see when comparing `hcr_256_1swap` with `hcr_256_nswap`.

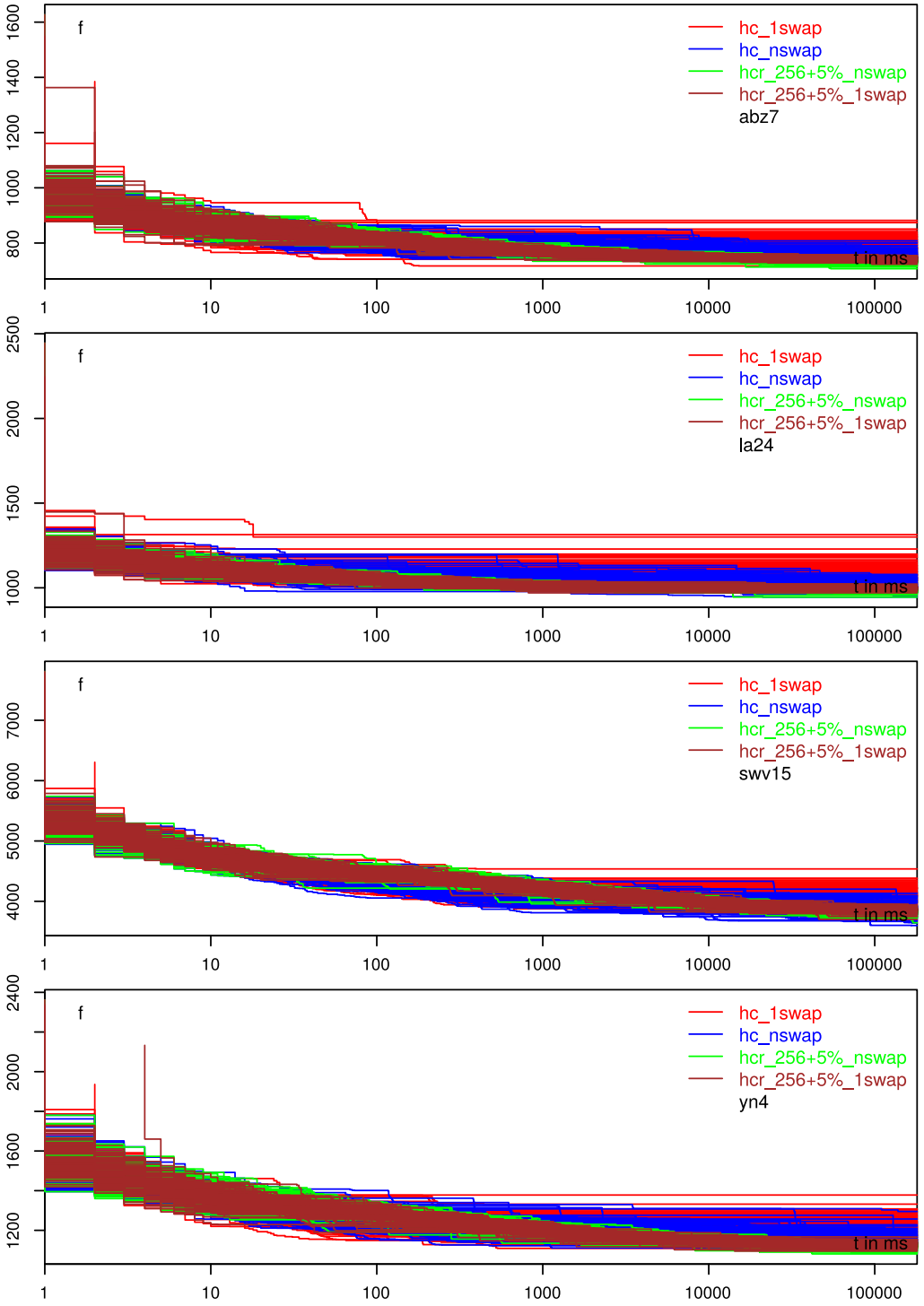

Figure 3.10: The progress of the hill climbers (without and with restarts) with the `1swap` and `nswap` operators over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis).

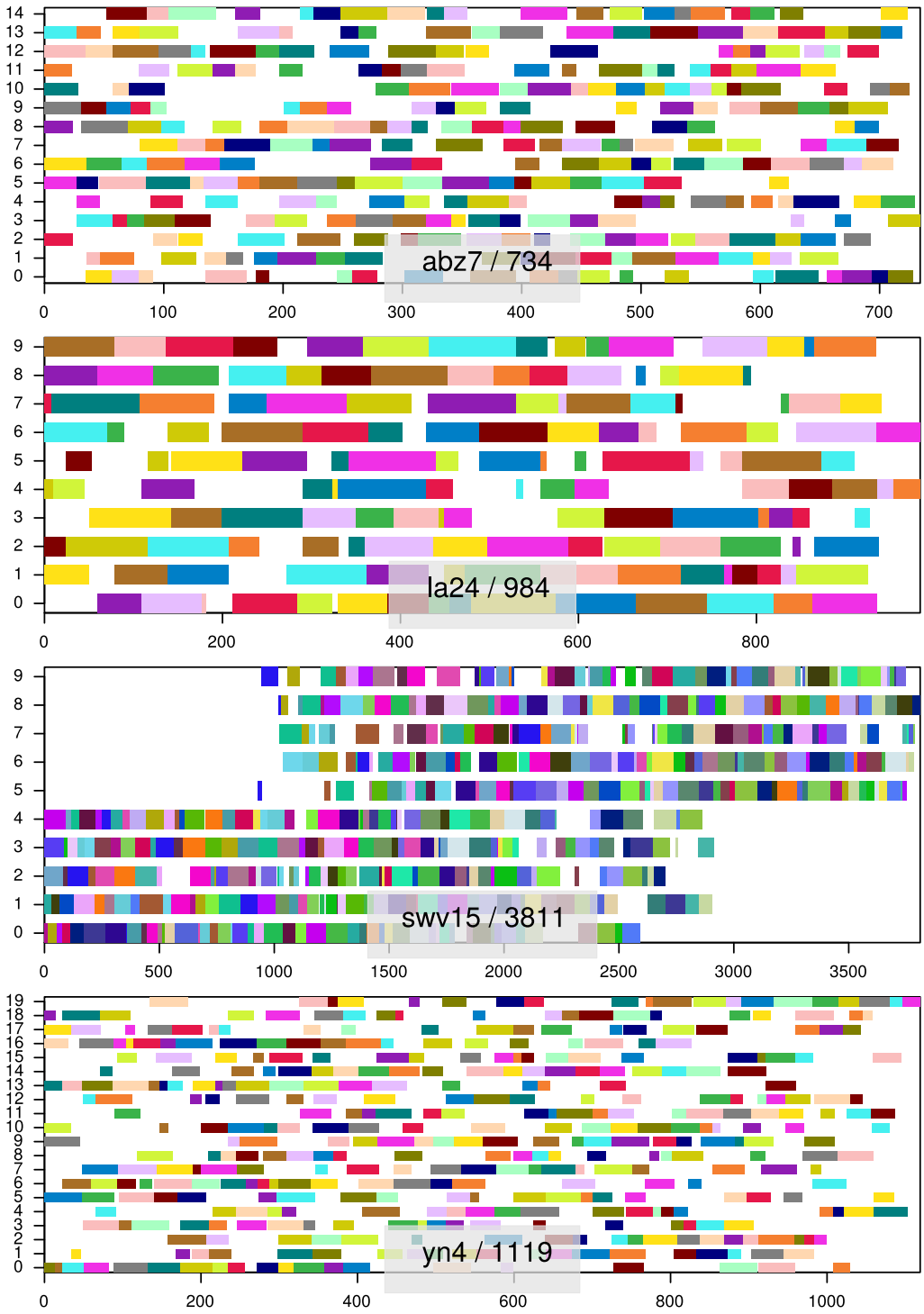

Figure 3.11: The Gantt charts of the median solutions obtained by the `hcr_256+5%_nswap` algorithm. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

Listing 3.12 An excerpt of the nswap operator for the JSSP, an implementation of the unary search operation interface Listing 2.9. nswap can swap an arbitrary number of jobs in our encoding, while favoring small search steps. (src)

```
1  public class JSSPUnaryOperatorNSwap
2      implements IUnarySearchOperator<int[]> {
3    public void apply(int[] x, int[] dest,
4        Random random) {
5  // copy the source point in search space to the dest
6      System.arraycopy(x, 0, dest, 0, x.length);
7
8  // choose the index of the first sub-job to swap
9      int i = random.nextInt(dest.length);
10     int first = dest[i];
11     int last = first; // last stores the last job id to swap with
12
13     boolean hasNext;
14     do { // we repeat a geometrically distributed number of times
15       hasNext = random.nextBoolean();
16       inner: for (;;) {// find a location with a different job
17         int j = random.nextInt(dest.length);
18         int job_j = dest[j];
19         if ((last != job_j) && // don't swap job with itself
20             (hasNext || (first != job_j))) { // also not at end
21           dest[i] = job_j; // overwrite job at index i with job_j
22           i = j; // remember index j: we will overwrite it next
23           last = job_j; // but not with the same value job_j...
24           break inner;
25         }
26       }
27     } while (hasNext); // Bernoulli process
28
29     dest[i] = first; // write back first id to last copied index
30   }
31 }
```

3.4 Evolutionary Algorithm

We now already have one more or less functional, basic optimization method – the hill climber. Different from the random sampling approach, it makes use of some knowledge gathered during the optimization process, namely the best-so-far point in the search space. However, only using this point led to the danger of premature convergence, which we tried to battle with two approaches, namely restarts and the search operator nswap spanning a larger neighborhood from which we sampled in a non-uniform way. These concepts can be transfered rather easily to may different kinds of optimization problems. Now we will look at a third concept to prevent premature convergence: Instead of just remembering and utilizing one single point from the search space during our search, we will work on an array of points!

3.4.1 Evolutionary Algorithm without Recombination

Today, there exists a wide variant of Evolutionary Algorithms (EAs) [15,36,48,70,119,120,151,163]. We will begin with a very simple, yet efficient variant: the $(\mu + \lambda)$ EA without recombination.[1] This algorithm always remembers the best $\mu \in \mathbb{N}_1$ points in the search space found so far. In each step, it derives $\lambda \in \mathbb{N}_1$ new points from them by applying the unary search operator.

3.4.1.1 The Algorithm (without Recombination)

The basic $(\mu + \lambda)$ Evolutionary Algorithm works as follows:

1. $I \in \mathbb{X} \times \mathbb{R}$ be a data structure that can store one point x in the search space and one objective value z.
2. Allocate an array P of length $\mu + \lambda$ instances of I.
3. For index i ranging from 0 to $\mu + \lambda - 1$ do
   a. Store a randomly chosen point from the search space in $P_i.x$.
   b. Apply the representation mapping $y = \gamma(P_i.x)$ to get the corresponding candidate solution y.
   c. Compute the objective objective value of y and store it at index i as well, i.e., $P_i.z = f(y)$.
4. Repeat until the termination criterion is met:
   d. Sort the array P according to the objective values such that the records with better associated objective value z are located at smaller indices. For minimization problems, this means elements with smaller objective values come first.

[1] For now, we will discuss EAs in a form without recombination. Wait for the binary recombination operator until Section 3.4.3.

e. Shuffle the first μ elements of P randomly.
f. Set the first source index $p = -1$.
g. For index i ranging from μ to $\mu + \lambda - 1$ do
   i. Set the source index p to $p = (p + 1) \bmod \mu$, i.e., make sure that every one of the μ selected points is used approximately the same number of times.
   ii. Set $P_i.x = \mathrm{searchOp}_1(P_p.x)$, i.e., derive a new point in the search space for the record at index i by applying the unary search operator to the point stored at index p.
   iii. Apply the representation mapping $y = \gamma(P_i.x)$ to get the corresponding candidate solution y.
   iv. Compute the objective objective value of y and store it at index i as well, i.e., $P_i.z = f(y)$.

5. Return the candidate solution corresponding to the best record in P to the user.

This algorithm is implemented in Listing 3.13. Basically, it starts out by creating and evaluating $\mu + \lambda$ random candidate solutions (*point 3*).

Definition 24. Each iteration of the main loop of an Evolutionary Algorithm is called a *generation*.

Definition 25. The array of solutions under investigation in an EA is called *population*.

In each generation, the μ best points in the population P are retained and the other λ solutions are overwritten.

Definition 26. The *selection* step in an Evolutionary Algorithm picks the set of points in the search space from which new points should be derived. This usually involves choosing a smaller number $\mu \in \mathbb{N}_1$ of points from a larger array P. [15,23,31,120,163]

Selection can be done by sorting the array P (*point d*). This way, the best μ solutions end up at the front of the array on the indices from 0 to $\mu - 1$. The worse λ solutions are at index μ to $\mu + \lambda - 1$. These are overwritten by sampling points from the neighborhood of the μ selected solutions by applying the unary search operator (which, in the context of EAs, is often called *mutation* operator).

Definition 27. The *reproduction* step in an Evolutionary Algorithm uses the selected $\mu \in \mathbb{N}_1$ points from the search space to derive $\lambda \in \mathbb{N}_1$ new points.

For each new point to be created during the reproduction step, we apply a search operator to one of the selected μ points. Therefore, the index p identifies the point to be used as source for sampling the next new solution. By incrementing p before each application of the search operator, we try to make sure that each of the selected points is used approximately equally often to create new solutions. Of course, μ and λ can be different (often $\lambda > \mu$), so if we would just keep increasing p for λ times, it could exceed μ. We thus performing a modulo division with μ, i.e., set p to the remainder of the division with μ, which makes sure that p will be in $0 \ldots (\mu - 1)$.

If $\mu \neq \lambda$, then the best solutions in P tend to be used more often, since they may "survive" selection several times and often be at the front of P. This means that, in our algorithm, they would be used more often as input to the search operator. To make our algorithm more fair, we randomly shuffle the selected μ points (*point f*) – their actual order does not matter, as they have already been selected.

3.4.1.2 Results on the JSSP

After implementing the $(\mu + \lambda)$ EA as discussed above, we already have all the ingredients ready to apply to the JSSP. We need to decide which values for μ and λ we want to use. The configuration of EAs is a whole research area itself. Here, let us just set $\mu = \lambda$ and test the values 16, 32, 64, 512, 2048, and 4096. We find that the two fairly large values 2048 and 4096 give the best results, so we will focus on them. We will call the corresponding setups `ea2048` and `ea4096`, respectively. As unary search operators, we test again `1swap` and `nswap`. The results are given in Table 3.6, together with those of our best hill climber with restarts `hcr_256+5%_nswap`.

Table 3.6: The results of the Evolutionary Algorithms without crossover in comparison to the best hill climber with restarts setup `hcr_256+5%_nswap`. The columns present the problem instance, lower bound, the algorithm, the best, mean, and median result quality, the standard deviation *sd* of the result quality, as well as the median time *med(t)* and FEs *med(FEs)* until the best solution of a run was discovered. The better values are **emphasized**.

$\mathcal{I}$	$\mathrm{lb}(f)$	setup	best	mean	med	sd	med(t)	med(FEs)
abz7	656	hcr_256+5%_nswap	707	733	734	**7**	64s	17'293'038
		ea2048_1swap	695	719	718	13	**11**s	2'581'614
		ea2048_nswap	694	714	714	12	18s	4'271'587
		ea4096_1swap	**688**	716	716	12	19s	4'416'129
		ea4096_nswap	692	**711**	**710**	10	34s	7'888'233
la24	935	hcr_256+5%_nswap	945	981	984	**9**	57s	29'246'097
		ea2048_1swap	945	983	983	16	**2**s	927'000
		ea2048_nswap	943	980	984	15	3s	1'329'883
		ea4096_1swap	941	980	978	14	5s	1'897'387
		ea4096_nswap	**938**	**976**	**975**	13	6s	2'512'530
swv15	2885	hcr_256+5%_nswap	3645	3804	3811	**44**	**91**s	14'907'737
		ea2048_1swap	3395	3535	3530	78	128s	19'290'521

$\mathcal{I}$	$\mathrm{lb}(f)$	setup	best	mean	med	sd	med(t)	med(FEs)
		ea2048_nswap	**3374**	**3521**	**3517**	70	157s	22'976'339
		ea4096_1swap	3397	3533	3533	54	171s	25'073'630
		ea4096_nswap	3421	3543	3539	46	178s	25'678'144
yn4	929	hcr_256+5%_nswap	1081	1117	1119	**14**	55s	11'299'461
		ea2048_1swap	1032	1082	1082	22	**26**s	4'792'622
		ea2048_nswap	1034	1074	1073	19	41s	7'514'890
		ea4096_1swap	**1020**	1076	1074	21	39s	6'907'692
		ea4096_nswap	1034	**1068**	**1067**	18	56s	9'976'531

Table 3.6 shows us that we can improve the best, mean, and median solution quality that we can get within three minutes of runtime by at least three percent when using our EA setups instead of the hill climber. The exception is case la24, where the hill climber already came close to the lower bound of the makespan. Here, the best solution encountered now has a makespan which is only 0.3% longer than what is theoretically possible. Nevertheless, we find quite a tangible improvement in case swv15.

The bigger setting 4096 for μ and λ tends to work better, except for instance swv15, where 2048 gives us better results. It is quite common in optimization that different problem instances may require different setups to achieve the best performance. Thenswap operator again works better than 1swap.

The best solution quality for abz7 delivered by ea4096_1swap is better than the best result found by the old Fast Simulated Annealing algorithm which was improved in [4], and both ea4096_1swap and ea4096_nswap find better best solutions on la24 as well (but are slower and have worse mean results and we also did more runs). Later, in Section 3.5, we will introduce Simulated Annealing.

The Gantt charts of the median solutions of ea4096_nswap are illustrated in Figure 3.12. More interesting are the progress diagrams of ea4096_nswap, ea2048_nswap, and hcr_256+5%_nswap in Figure 3.13. Here we find big visual differences between the way the EAs and hill climbers proceed. The EAs spend the first 100ms to discover some basins of attraction of local optima before speeding up. The larger the population, the longer it takes them until this happens. It is interesting to notice that the two problems where the EAs visually outperform the hill climber the most, swv15 and yn4, are also those with the largest search spaces (see Table 2.3). la24, however, which already can "almost be solved" by the hill climber and where there are the smallest differences in perfomance, is the smallest instance. The population used by the EA seemingly guards against premature convergence and allows it to keep progressing for a longer time.

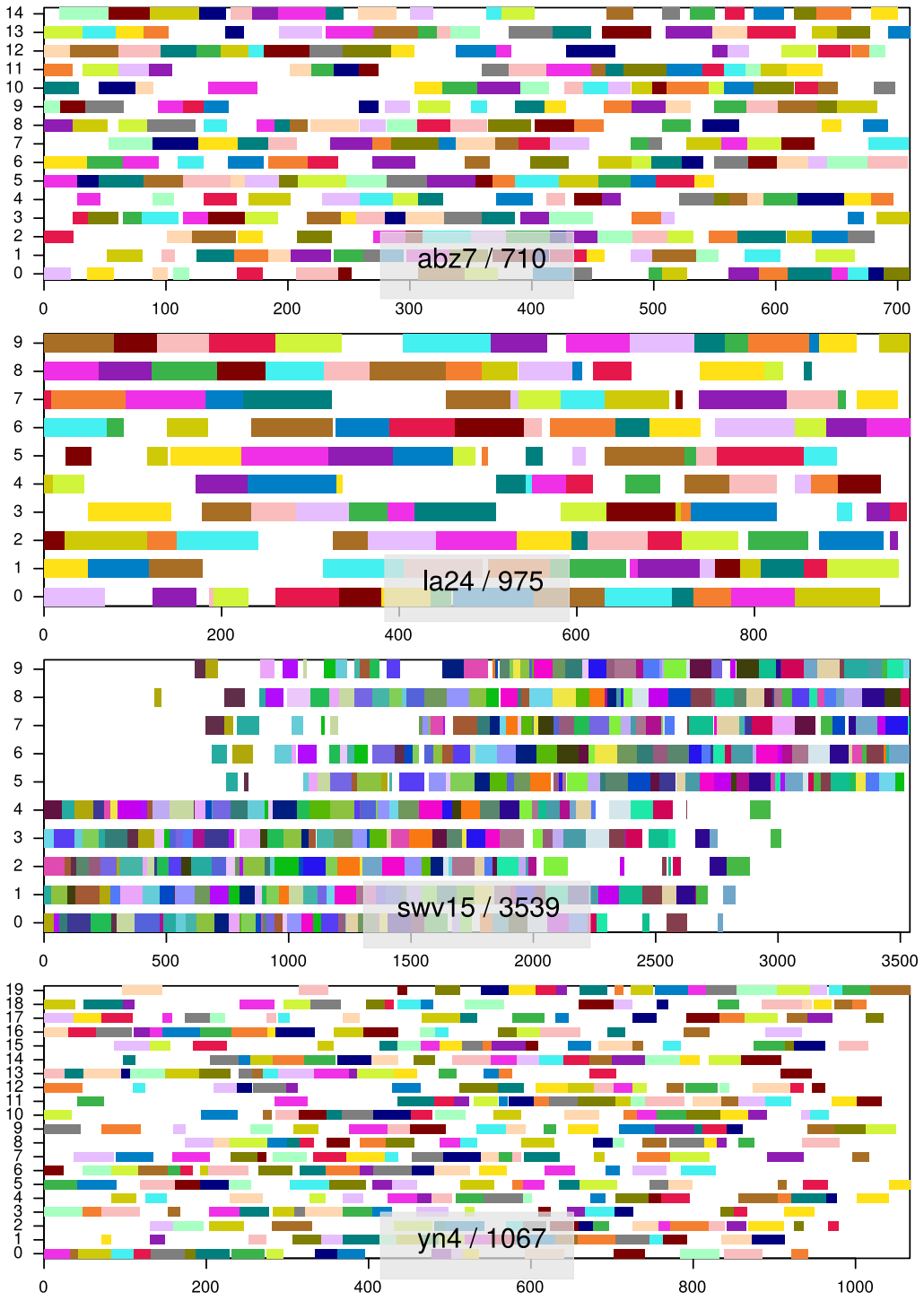

Figure 3.12: The Gantt charts of the median solutions obtained by the `ea4096_nswap` setup. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

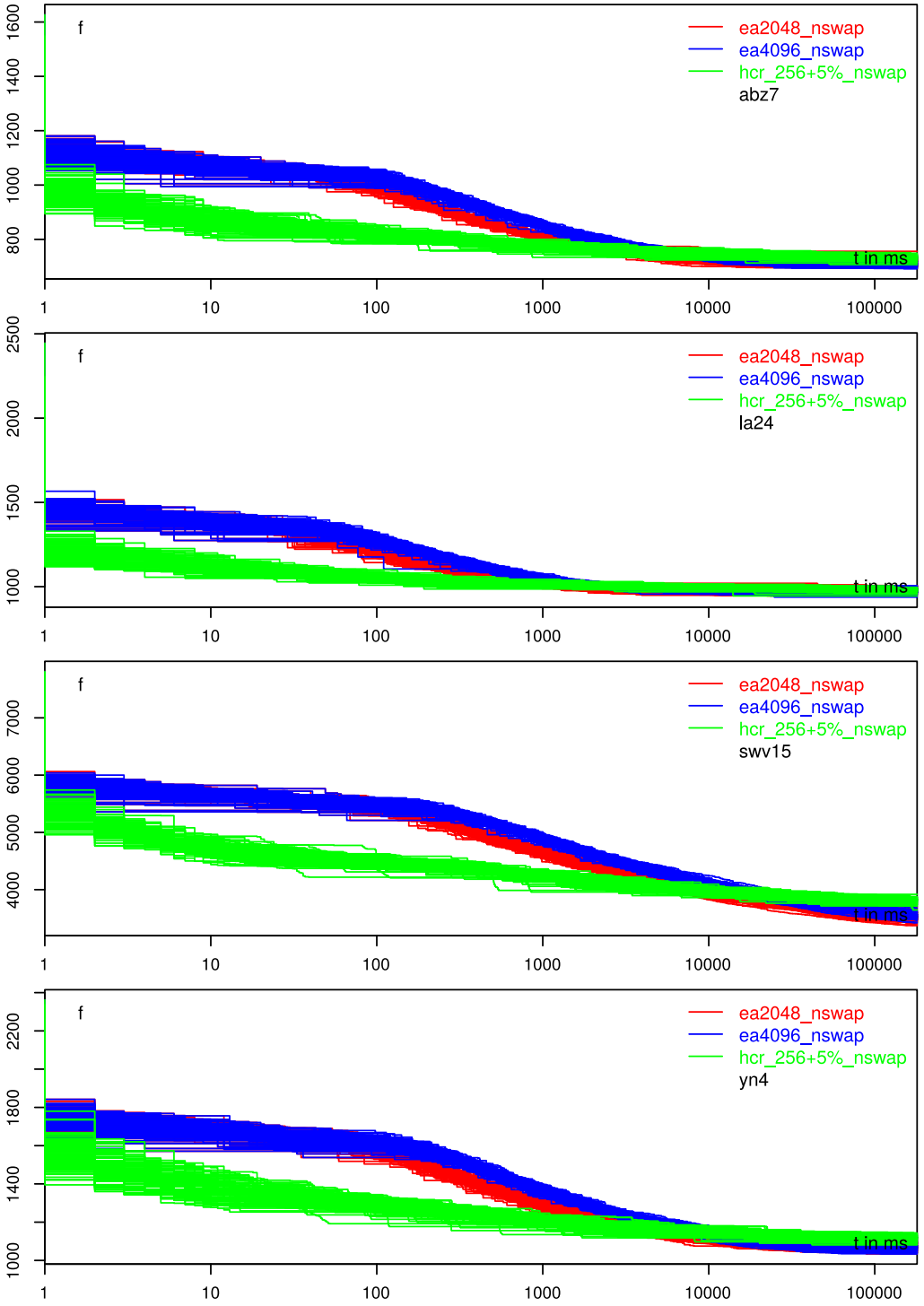

Figure 3.13: The progress of the `ea4096_nswap`, `ea2048_nswap`, and `hcr_256+5%_nswap` algorithms over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis).

3.4.1.3 Exploration versus Exploitation

Naturally, we may ask why the population is helpful in the search. First of all, we can consider it as a "generalized" version of the Hill Climber. If we would set $\mu = 1$ and $\lambda = 1$, then we would always remember the best solution we had so far and, in each generation, derive one new solution from it. This is the hill climber.

Now imagine what would happen if we would set μ to infinity. We then would remember each and every point in the search space we would have ever visited during the search. We would not perform any actual selection, as we would always select all points. Our search would not be steered in any direction, there would not be any *bias* or preference for better solutions. Due to the fairness of our algorithm when it comes to selecting "parent" points for sampling, each of the past solutions would have the same chance to be the input to the unary search operator to produce the next point to visit. In other words, the EA would be some wierd version of random sampling.

The parameter μ basically allows us to "tune" between these two behaviors [172]! If we pick it small, our algorithm becomes more "greedy". It will investigate (*exploit*) the neighborhood current best solutions more eagerly, which means that it will trace down local optima faster but be trapped more easily in local optima as well. If we set μ to a larger value, we will keep more not-that-great solutions in its population. The algorithm spends more time *exploring* the neighborhoods of solutions which do not look that good, but from which we might eventually reach better results. The convergence is slower, but we are less likely to get trapped in a local optimum.

This is dilemma of "Exploration versus Exploitation" [58,163,166,174].

Listing 3.13 An excerpt of the implementation of the Evolutionary Algorithm algorithm **without** crossover. (src)

```
public class EA<X, Y> implements IMetaheuristic<X, Y> {
  public void solve(IBlackBoxProcess<X, Y> process) {
// omitted: initialize local variables random, searchSpace,
// nullary, unary and the array P of length mu+lambda
// first generation: fill population with random individuals
    for (int i = P.length; (--i) >= 0;) {
      X x = searchSpace.create();
      nullary.apply(x, random);
      P[i] = new Individual<>(x, process.evaluate(x));
    }

    for (;;) { // main loop: one iteration = one generation
// sort the population: mu best individuals at front are selected
      Arrays.sort(P);
// shuffle the first mu solutions to ensure fairness
      RandomUtils.shuffle(random, P, 0, this.mu);
      int p1 = -1; // index to iterate over first parent

// override the worse lambda solutions with new offsprings
      for (int index = P.length; (--index) >= this.mu;) {
        if (process.shouldTerminate()) { // we return
          return; // best solution is stored in process
        }

        Individual<X> dest = P[index];
        Individual<X> sel = P[(++p1) % this.mu];
// create modified copy of parent using unary operator
          unary.apply(sel.x, dest.x, random);
// map to solution/schedule and evaluate quality
        dest.quality = process.evaluate(dest.x);
      } // the end of the offspring generation
    } // the end of the main loop
  }
}
```

3.4.2 Ingredient: Binary Search Operator

On one hand, keeping a population of the $\mu > 1$ best solutions as starting points for further exploration helps us to avoid premature convergence. On the other hand, it also represents more *information*. The hill climber only used the information in current-best solution as guide for the search (and the hill climber with restarts used, additionally, the number of steps performed since the last improvement). Now we have a set of μ selected points from the search space. These points have, well, been selected. At least after some time has passed in our optimization process, "being selected" means "being good". If you compare the Gantt charts of the median solutions of `ea4096_nswap` (Figure 3.12) and `hcr_256+5%_nswap` (Figure 3.8), you can see some good solutions, which, however, do differ in some details. Wouldn't it be nice if we could take two good solutions and derive a solution "in between," a new solution which is similar to both of its "parents"?

This is the idea of the binary search operator (also often referred to as *recombination* or *crossover* operator). By defining such an operator, we hope that we can merge the "good characteristics" of two selected solutions to obtain one new (ideally better) solution [47,90]. If we are lucky and that works, then ideally such good characteristics could aggregate over time [70,121].

How can we define a binary search operator for our JSSP representation? *One possible* idea would be to create a new encoded solution x' by processing both input points $x1$ and $x2$ from front to back and "schedule" their not-yet scheduled job IDs into x' similar to what we do in our representation mapping.

1. Allocate a data structure x' to hold the new point in the search space that we want to sample.
2. Set the index i where the next sub-job should be stored in x' to $i = 0$.
3. Repeat
    a. Randomly choose of the input points $x1$ or $x2$ with equal probability as source x.
    b. Select the first (at the lowest index) sub-job in x that is not marked yet and store it in variable J.
    c. Set $x'_i = J$.
    d. Increase i by one ($i = i + 1$).
    e. If $i = n * m$, then all sub-jobs have been assigned. We exit and returning x'.
    f. Mark the first unmarked occurrence of J as "already assigned" in $x1$.
    g. Mark the first unmarked occurrence of J as "already assigned" in $x2$.

This can be implemented efficiently keeping indices of the first unmarked element for both $x1$ and $x2$, which we do in Listing 3.14.

As we discussed in Section 2.6.2, our representation mapping processes the elements $x \in \mathbb{X}$ from the front to the back and assigns the job to machines according to the order in which their IDs appear. Our

binary operator works in a similar way, but it processes *two* points from the search space $x1$ and $x2$ from their beginning to the end. At each step randomly picks one of them to extract the next sub-job, which is is then stored in the output x' and marked as "done" in both $x1$ and $x2$.

If it would, by chance, always choose $x1$ as source, then it would produce exactly $x1$ as output. If it would always pick $x2$ as source, then it would also return $x2$. If it would pick $x1$ for the first half of the times and then always pick $x2$, it would basically copy the first half of $x1$ and then assign the rest of the sub-jobs in exactly the order in which they appear in $x2$.

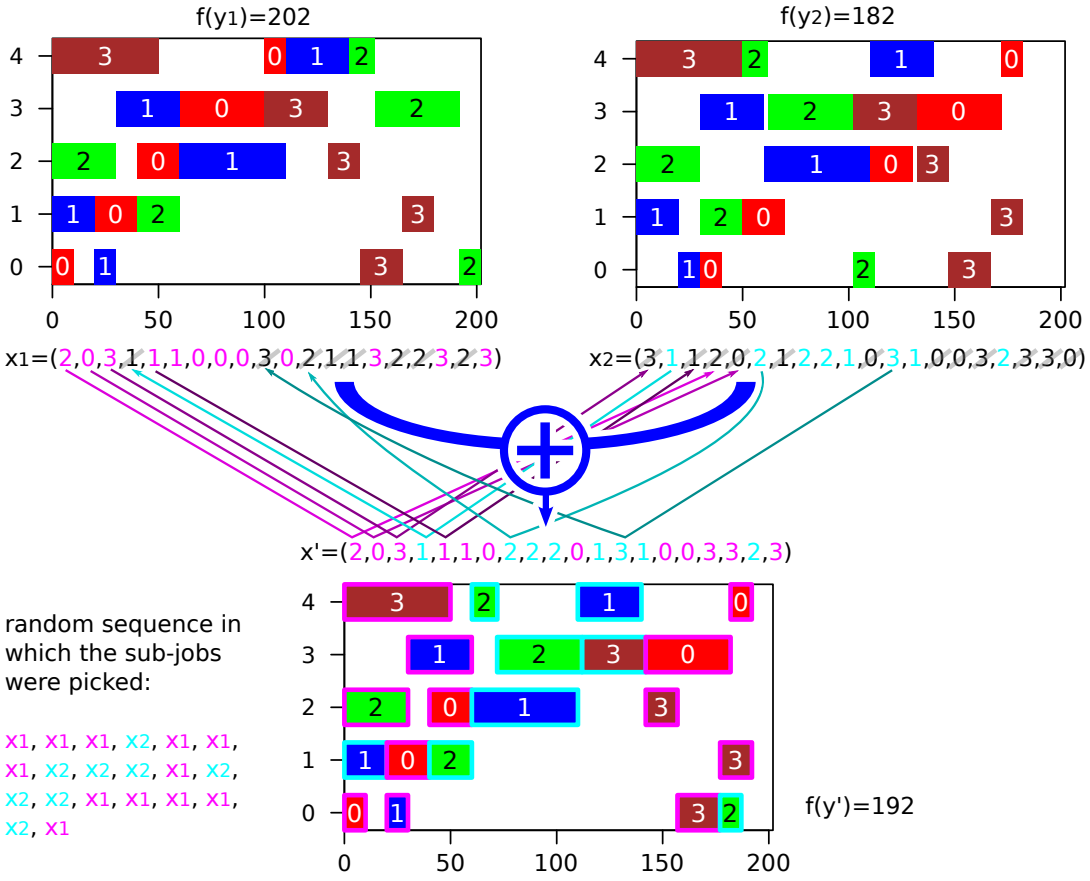

Figure 3.14: An example application of our sequence recombination operator to two points $x1$ and $x2$ in the search space of the demo instance, resulting in a new point x'. We mark the selected job IDs with pink and cyan color, while crossing out those IDs which were not chosen because of their received marks in the source points. The corresponding candidate solutions $y1$, $y2$, and y' are illustrated as well.

For illustration purposes, one example application of this operator is sketched in Figure 3.14. As input, we chose to points $x1$ and $x2$ from the search space for our demo instance. They encode two different corresponding Gantt charts, $y1$ and $y2$, with makespans of 202 and 182 time units, respectively.

Our operator begins by randomly choosing $x1$ as the source of the first sub-job for the new point x'. The first job ID in $x1$ is 2, which is placed as first sub-job into x'. We also mark the first occurrence of 2 in $x2$, which happens to be at position 4, as "already scheduled." Then, the operator again randomly picks $x1$ as source for the next sub-job. The first not-yet marked element in $x1$ is now at the second 0, so it is placed into x' and marked as scheduled in $x2$, where the fifth element is thus crossed out. As next source, the operator, again, chooses&bsnp;$x1$. The first unmarked sub-job in $x1$ is 3 at position 3, which is added to x' and leads to the first element of $x2$ being marked. Finally, for picking the next sub-job, $x2$ is chosen. The first unmarked sub-job there has ID 1 and is located at index 2. It is inserted at index 4 into x'. It also occurs at index 4 in $x1$, which is thus marked. This process is repeated again and again, until x' is constructed completely, at which point all the elements of $x1$ and $x2$ are marked.

The application of our binary operator yields a new point x' which corresponds to the Gantt chart y' with makespan 192. This new candidate solution clearly "inherits" some characteristics from either of its parents.

Listing 3.14 An excerpt of the `sequence` recombination operator for the JSSP, an implementation of the binary search operation interface Listing 2.10. (src)

```
1  public class JSSPBinaryOperatorSequence
2      implements IBinarySearchOperator<int[]> {
3    public void apply(int[] x0, int[] x1,
4        int[] dest, Random random) {
5  // omitted: initialization of arrays done_x0 and done_x1 (that
6  // remember the already-assigned sub-jobs from x0 and x1) of
7  // length=m*n to all false; and indices desti, x0i, x10 to 0
8      for (;;) { // repeat until dest is filled, i.e., desti=length
9  // randomly chose a source point and pick next sub-job from it
10       int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
11       dest[desti++] = add; // we picked a sub-job and added it
12       if (desti >= length) { // if desti==length, we are finished
13         return; // in this case, desti is filled and we can exit
14       }
15
16       for (int i = x0i;; i++) { // mark the sub-job as done in x0
17         if ((x0[i] == add) && (!done_x0[i])) { // find added job
18           done_x0[i] = true;// found it and marked it
19           break; // quit sub-job finding loop
20         }
21       }
22       while (done_x0[x0i]) { // now we move the index x0i to the
23         x0i++; // next, not-yet completed sub-job in x0
24       }
25
26       for (int i = x1i;; i++) { // mark the sub-job as done in x1
27         if ((x1[i] == add) && (!done_x1[i])) { // find added job
28           done_x1[i] = true; // found it and marked it
29           break; // quit sub-job finding loop
30         }
31       }
32       while (done_x1[x1i]) { // now we move the index x1i to the
33         x1i++; // next, not-yet completed sub-job in x0
34       }
35     } // loop back to main loop and to add next sub-job
36   } // end of function
37 }
```

3.4.3 Evolutionary Algorithm with Recombination

We can now utilize this new operator in our EA, which therefore needs to be modified a bit.

3.4.3.1 The Algorithm (with Recombination)

We introduce a new paramter $cr \in [0, 1]$, the so-called "crossover rate". It is used whenever we want to derive a new points in the search space from existing ones. It denotes the probability that we apply the binary operator (while we will apply the unary operator with probability $1 - cr$). The basic $(\mu + \lambda)$ Evolutionary Algorithm with recombination works as follows:

1. $I \in \mathbb{X} \times \mathbb{R}$ be a data structure that can store one point x in the search space and one objective value z.
2. Allocate an array P of length $\mu + \lambda$ instances of I.
3. For index i ranging from 0 to $\mu + \lambda - 1$ do
   a. Store a randomly chosen point from the search space in $P_i.x$.
   b. Apply the representation mapping $y = \gamma(P_i.x)$ to get the corresponding candidate solution y.
   c. Compute the objective objective value of y and store it at index i as well, i.e., $P_i.z = f(y)$.
4. Repeat until the termination criterion is met:
   d. Sort the array P according to the objective values such that the records with better associated objective value z are located at smaller indices. For minimization problems, this means elements with smaller objective values come first.
   e. Shuffle the first μ elements of P randomly.
   f. Set the first source index $p = -1$.
   g. For index i ranging from μ to $\mu + \lambda - 1$ do
      i. Set the source index p to $p = (p + 1) \bmod \mu$, i.e., make sure that every one of the μ selected points is used approximately the same number of times.
      ii. Draw a random number c uniformly distributed in $[0, 1)$.
      iii. If c is less than the crossover rate cr, then we apply the binary operator: A. Randomly choose another index $p2$ from $0 \ldots (\mu - 1)$ such that $p2 \neq p$. B. Set $P_i.x = \mathrm{searchOp}_2(P_p.x, P_{p2}.x)$, i.e., derive a new point in the search space for the record at index i by applying the binary search operator to the points stored at index p and $p2$.
      iv. else, i.e., $c \geq cr$, then we apply the unary operator: C. Set $P_i.x = \mathrm{searchOp}_1(P_p.x)$, i.e., derive a new point in the search space for the record at index i by applying the unary search operator to the point stored at index p.

v. Apply the representation mapping $y = \gamma(P_i.x)$ to get the corresponding candidate solution y.
vi. Compute the objective objective value of y and store it at index i as well, i.e., $P_i.z = f(y)$.

5. Return the candidate solution corresponding to the best record in P to the user.

This algorithm, implemented in Listing 3.15 only differs from the version in Section 3.4.1.1 by choosing whether to use the unary or binary operator to sample new points from the search space (*steps A*, *B*, and *C*). If cr is the probability to apply the binary operator and we draw a random number c which is uniformly distributed in $[0, 1)$, then the probability that $c < cr$ is exactly cr (see *point iii*).

3.4.3.2 Results on the JSSP

We now apply the new algorithm with our binary `sequence` operator to the JSSP. As unary operator, we only apply `nswap` and for μ and λ, we again provide results for the values 2048 and 4096. As crossover rates cr, we use 0, 0.05, and 0.3. A crossover rate of 0 is exactly equivalent to not applying the binary operator at all, that is, to our EAs from Section 3.4.1. For the non-zero crossover rates, we append $cr * 100$ to the setup name, i.e., `ea2048_nswap_30` stands for an $(2048 + 2048)$ EA with the `nswap` unary operator which applies the binary `sequence` operator at a crossover rate (=probability) of 0.3.

Table 3.7: The results of the Evolutionary Algorithms with crossover rates 0, 0.05, and 0.3. The columns present the problem instance, lower bound, the algorithm, the best, mean, and median result quality, the standard deviation *sd* of the result quality, as well as the median time *med(t)* and FEs *med(FEs)* until the best solution of a run was discovered. The better values are **emphasized**.

$\mathcal{I}$	$\mathrm{lb}(f)$	setup	best	mean	med	sd	med(t)	med(FEs)
abz7	656	ea2048_nswap	694	714	714	12	**18**s	4'271'587
		ea2048_nswap_5	691	710	709	9	19s	4'105'841
		ea2048_nswap_30	689	710	710	9	24s	3'228'294
		ea4096_nswap	692	711	710	10	34s	7'888'233
		ea4096_nswap_5	**685**	**706**	**706**	10	29s	5'933'332
		ea4096_nswap_30	691	708	**706**	**8**	29s	3'675'335
la24	935	ea2048_nswap	943	980	984	15	**3**s	1'329'883
		ea2048_nswap_5	941	975	975	15	4s	1'638'907

$\mathcal{I}$	$\mathrm{lb}(f)$	setup	best	mean	med	sd	med(t)	med(FEs)
		ea2048_nswap_30	946	978	979	**11**	5s	1'214'869
		ea4096_nswap	**938**	976	975	13	6s	2'512'530
		ea4096_nswap_5	941	**974**	**971**	13	6s	2'277'833
		ea4096_nswap_30	947	975	975	12	14s	3'308'665
swv15	2885	ea2048_nswap	3374	**3521**	**3517**	70	157s	22'976'339
		ea2048_nswap_5	**3372**	3531	3527	70	142s	18'919'277
		ea2048_nswap_30	3454	3595	3589	69	**119**s	11'980'325
		ea4096_nswap	3421	3543	3539	**46**	178s	2'567'8144
		ea4096_nswap_5	3440	3543	3537	51	177s	22'603'785
		ea4096_nswap_30	3458	3595	3599	63	176s	16'530'328
yn4	929	ea2048_nswap	1034	1074	1073	19	41s	7'514'890
		ea2048_nswap_5	1027	1067	1066	19	34s	5'523'450
		ea2048_nswap_30	1035	1070	1069	18	**31**s	3'116'408
		ea4096_nswap	1034	1068	1067	18	56s	9'976'531
		ea4096_nswap_5	**1017**	**1058**	**1058**	18	52s	8'248'627
		ea4096_nswap_30	1030	1061	1060	**17**	50s	4'828'673

ea4096_nswap_5 outperforms all the Genetic Algorithms in [2] and [100] and the Grey Wolf Algorithm in [98] in terms of both best and mean result quality on la24.

The results in Table 3.7 show that a moderate crossover rate of 0.05 can indeed improve our algorithm's performance – a little bit. Only for the JSSP instance swv15, setup ea2048_nswap without crossover remains best. Here, the reason is probably hidden in the late median last improvement times, which are already at 157s and 178s for the two algorithm variants with $cr = 0$. Since the total budget is only 180s, there might just not be enough time for any potential benefits of the binary operator to kick in. This could also be a valuable lesson: it does not help if the algorithm gives better results if it needs too much time. Any statement about an achieved result quality is only valid if it also contains a statement about the required computational budget. If we would have let the algorithms longer, maybe the setups using the binary operator would have given more saliently better results … but these would then be useless in our real-world scenario, since we only have 3 minutes of runtime.

By the way: It is very important to *always* test the $cr = 0$ rate! Only by doing this, we can find whether

our binary operator is designed properly. It is a common fallacy to assume that an operator which we have designed to combine good characteristics from different solutions *will actually do that*. If the algorithm setups with $cr = 0$ would be better than those that use the binary operator, it is a clear indication that we are doing something wrong. So we need to carefully analyze whether the small improvements that our binary operator can provide are actually *significant*.

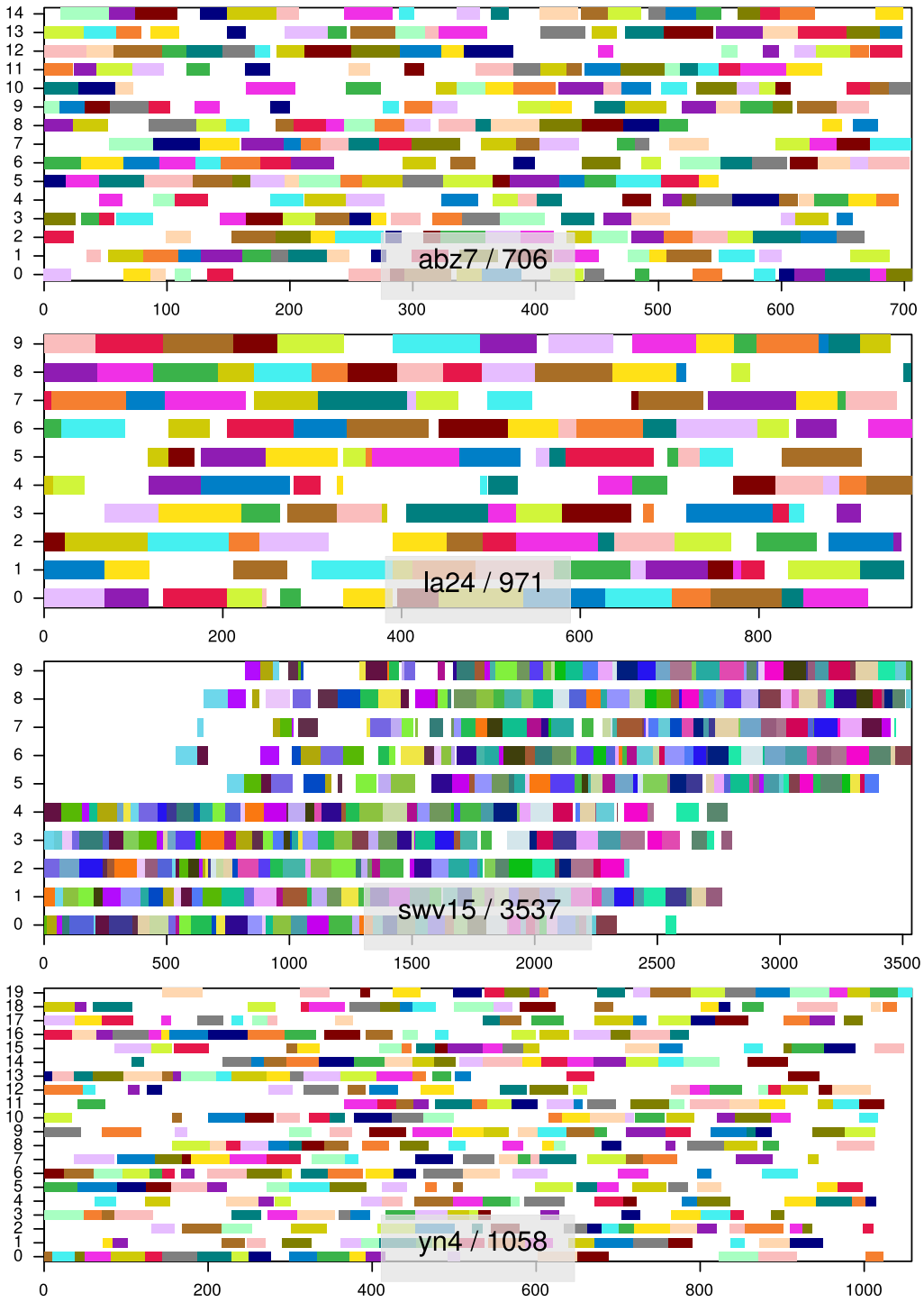

Figure 3.15: The Gantt charts of the median solutions obtained by the `ea4096_nswap_5` setup. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

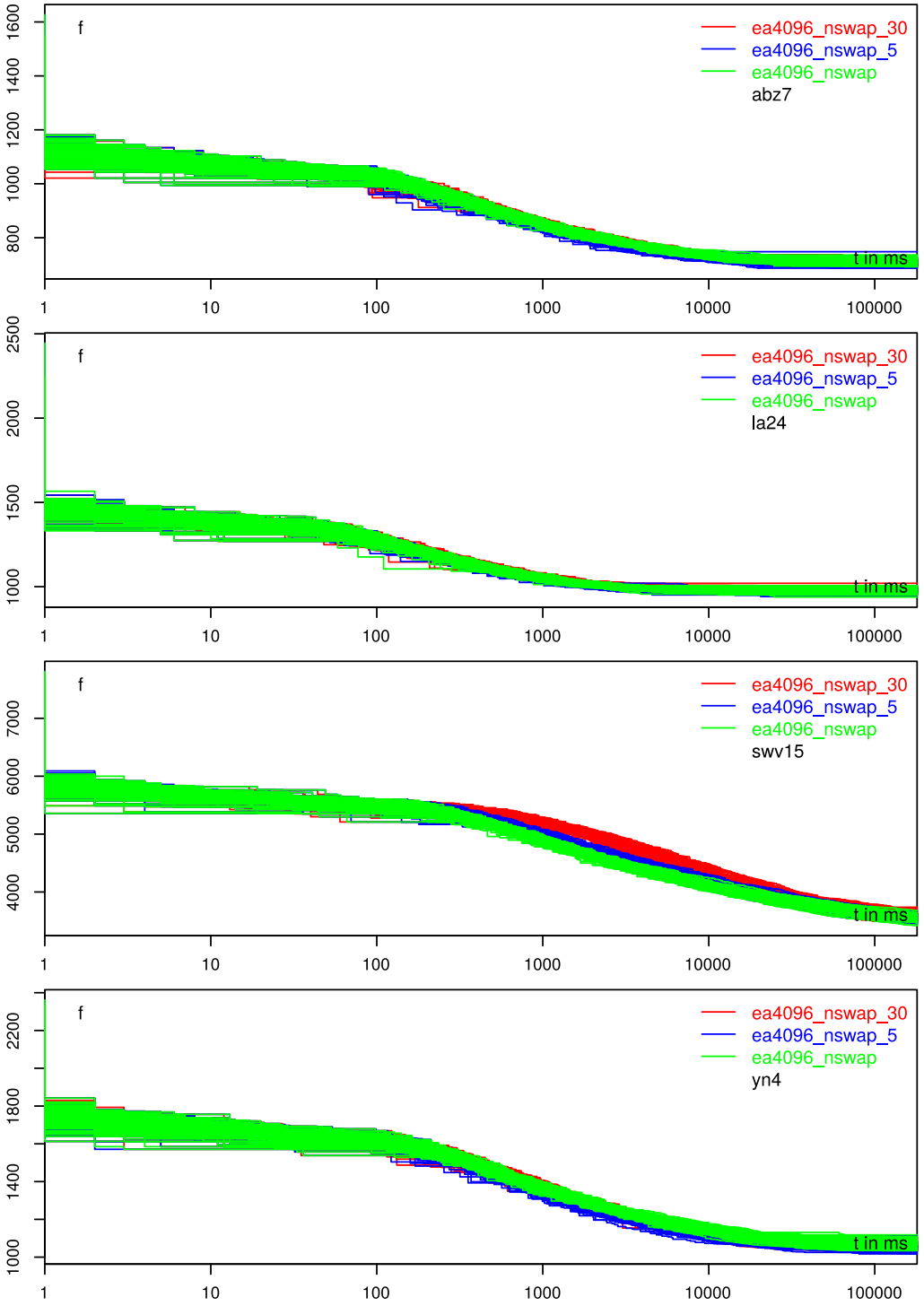

Figure 3.16: The progress of the `ea4096_nswap` setup without binary operator compared to those of `ea4096_nswap_5` and `ea4096_nswap_30`, which apply the binary operator in 5% and 30% of the reproduction steps, over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis).

Indeed, if we look at the progress of the setups `ea4096_nswap`, `ea4096_nswap_5`, and `ea4096_nswap_30` over time (illustrated in Figure 3.16), we find that they look quite similar. Also the schedules of median quality obtained by `ea4096_nswap_5` and plotted in Figure 3.15 do not look very different from those of `ea4096_nswap` shown in Figure 3.12. Of course, applying an operator only 5% of the time, which here seems to be the better choice, will probably not change the algorithm behavior very much. Furthermore, in instance `la24`, we are already very close to lower bound defining the best possible solution quality that can theoretically be reached.

Listing 3.15 An excerpt of the implementation of the Evolutionary Algorithm algorithm **with** crossover. (src)

```java
public class EA<X, Y> implements IMetaheuristic<X, Y> {
  public void solve(IBlackBoxProcess<X, Y> process) {
// omitted: initialize local variables random, searchSpace,
// nullary, unary, binary, and array P of length mu+lambda
// first generation: fill population with random individuals
    for (int i = P.length; (--i) >= 0;) {
      X x = searchSpace.create();
      nullary.apply(x, random);
      P[i] = new Individual<>(x, process.evaluate(x));
    }

    for (;;) { // main loop: one iteration = one generation
// sort the population: mu best individuals at front are selected
      Arrays.sort(P);
// shuffle the first mu solutions to ensure fairness
      RandomUtils.shuffle(random, P, 0, this.mu);
      int p1 = -1; // index to iterate over first parent

// override the worse lambda solutions with new offsprings
      for (int index = P.length; (--index) >= this.mu;) {
        if (process.shouldTerminate()) { // we return
          return; // best solution is stored in process
        }

        Individual<X> dest = P[index];
        Individual<X> sel = P[(++p1) % this.mu];
        if (random.nextDouble() <= this.cr) { // crossover!
          int p2; // to hold index of second selected record
          do { // find a second, different record
            p2 = random.nextInt(this.mu);
          } while (p2 == p1);
// perform recombination of the two selected individuals
          binary.apply(sel.x, P[p2].x, dest.x, random);
        } else {
// create modified copy of parent using unary operator
          unary.apply(sel.x, dest.x, random);
        }
// map to solution/schedule and evaluate quality
        dest.quality = process.evaluate(dest.x);
      } // the end of the offspring generation
    } // the end of the main loop
  }
}
```

3.4.4 Testing for Significance

All in all, the changes in both Table 3.7 and Figure 3.16 achieved by introducing recombination in the EA seem to not be very big. This could either mean that they are an artifact of the randomness in the algorithm *or*, well, that there are improvements but they are small.

In order to understand the first situation, consider the following thought experiment. Assume you have a completely unbiased, uniform source of true random real numbers from the interval $[0, 1)$. You draw 500 such numbers, i.e., have a list A containing 500 numbers, each from $[0, 1)$. Now you repeat the experiment and get a list B. Since the numbers stem from a random source, we can expect that $A \neq B$. If we compute the medians A and B, they are likely to be different as well. Actually, I just did exactly this in the R programming language and got `median(A)=0.5101432` and `median(B)=0.5329007`. Does this mean that the generator producing the numbers in A creates somehow smaller numbers than the generator from which the numbers in B stem? Obviously not, because we sampled the numbers from the same source. Also, every time I would repeat this experiment, I would get different results.

Now, our EAs are randomized as well. On `yn4`, setup `ea4096_nswap_5` has a median end result quality of 1058, while `ea4096_nswap` (without binary operator) achieves 1067, a difference of 0.8%. If our binary operator would have no impact whatsoever, we could theoretically still this results or any other from Table 3.7, just because of the randomness in the algorithms. It is simply not possible to decide, without further investigation, whether results and algorithm behaviors that overlap as much as those in Figure 3.16 are actually different or not. The "further investigation" which allows us to make this decision is called significance test and it is discussed in-depth in Section 4.5 as part of our investigation on how to compare algorithms.

In order to see whether two different setups also behave differently, we compare their two sets of 101 end results on each of the problem instances. For this purpose, we use the Mann-Whitney U test), as prescribed in Section 4.5.4 and compare the end results of the two setups `ea4096_nswap` and `ea4096_nswap_5`:

On instance `abz7`, we obtain 0.0016 as p-value.
On instance `la24`, we obtain 0.3275 as p-value.
On instance `swv15`, we obtain 0.8757 as p-value.
On instance `yn4`, we obtain 0.0002 as p-value.

The p-value can roughly be interpreted as the probability of observing the differences that we saw if the two algorithms would produce similar results. We obtain two very small p-values on `abz7` and `yn4`. There, it would thus be unlikely to see the different outcomes that we saw under the assumption that the binary operator is not useful. This means we can instead conclude that our binary operator

sequence instead leads to real, significant improvements on these instances. The p-values bigger than 0.3 on the other two instances indicate that it does probably not make an actual difference there, so while our operator does certain not improve the results on swv15, from a statistical point of view, it also does not make them significantly worse.

In summary, although it was not as beneficial as one would have hoped, using the binary operator can be considered as helpful in our case. Of course, we just tested *one* binary operator on only *four* problem instances – in any application scenario, we would do more experiments with more settings.

3.5 Simulated Annealing

So far, we have only discussed one variant of local search: the hill climbing algorithm. A hill climbing algorithm is likely to get stuck at local optima, which may vary in quality. We found that we can utilize this variance of the result quality by restarting the optimization process when it could not improve any more in Section 3.3.3. Such a restart is costly, as it forces the local search to start completely from scratch (while we, of course, remember the best-ever solution in a variable hidden from the algorithm).

Another way to look at this is the following: A schedule which is a local optimum probably is somewhat similar to what the globally optimal schedule would look like. It must, obviously, also be somewhat different. This difference is shaped such that it cannot be conquered by the unary search operator that we use, because otherwise, the basic hill climber could already move from the local to the global optimum. If we do a restart, we also dispose of the similarities to the global optimum that we have already discovered. We will subsequently spend time to re-discover them in the hope that this will happen in a way that allows us to eventually reach the global optimum itself. But maybe there is a less-costly way? Maybe we can escape from a local optimum without discarding the entirety good solution characteristics we already have discovered?

3.5.1 Idea: Accepting Worse Solutions with Decreasing Probability

Simulated Annealing (SA) [42,96,103,132] is a local search which provides another approach to escape local optima [151,163]. The algorithm is inspired by the idea of simulating the thermodynamic process of *annealing* using statistical mechanics, hence the naming [118]. Instead of restarting the algorithm when reaching a local optima, it tries to preserve the parts of the current best solution by permitting search steps towards worsening objective values. This algorithm therefore introduces three principles:

1. Worse candidate solutions are sometimes accepted, too.
2. The probability P of accepting them is decreases with increasing differences ΔE of the objective values to the current best solution.

3. The probability also decreases with the number of performed search steps.

This basic idea is realized as follows. First, ΔE be the difference between the objective value of the freshly sampled point x' from the search space and the "current" best point x, where γ is the representation mapping and f the objective function, i.e.

$$\Delta E = f(\gamma(x')) - f(\gamma(x)) \tag{3.1}$$

Clearly, if we try to minimize the objective function f, then $\Delta E < 0$ means that x' is better than x since $f(\gamma(x')) < f(\gamma(x))$. If $\Delta E > 0$, on the other hand, the new solution is worse. The probability P to overwrite x with x' then be

$$P = \begin{cases} 1 & \text{if } \Delta E \leq 0 \\ e^{-\frac{\Delta E}{T}} & \text{if } \Delta E > 0 \wedge T > 0 \\ 0 & \text{otherwise } (\Delta E > 0 \wedge T = 0) \end{cases} \tag{3.2}$$

In other words, if the new candidate solution is actually better than the current best one, i.e., $\Delta E < 0$, then we will definitely accept it. If the new solution is worse ($\Delta E > 0$), the acceptance probability then

1. gets smaller the larger ΔE is and
2. gets smaller the smaller the so-called "temperature" $T \geq 0$ is.

Both the temperature $T > 0$ and the objective value difference $\Delta E > 0$ enter Equation (3.2) in an exponential term and the two above points follow from $e^{-a} < e^{-b} \forall a > b$ and $e^{-a} \in [0, 1] \forall a > 0$.

The temperature decreases and approaches zero with the algorithm iteration τ, i.e., the performed objective function evaluations. The optimization process is initially "hot." Then, the search progresses wildly any may accept even significantly worse solutions. As the process "cools" down, the search tends to accept fewer and fewer worse solutions and more likely such which are only a bit worse. Eventually, at temperature $T = 0$, the algorithm only accepts better solutions. In other words, T is actually a monotonously decreasing function $T(\tau)$ called the "temperature schedule" and it holds that $\lim_{\tau \to \infty} T(\tau) = 0$.

3.5.2 Ingredient: Temperature Schedule

The temperature schedule $T(\tau)$ determines how the temperature changes over time (where time is measured in algorithm steps τ). It begins with an start temperature T_s at $\tau = 1$. Then, the temperature is the highest, which means that the algorithm is more likely to accept worse solutions. It will then

behave a bit similar to a random walk and put more emphasis on exploring the search space than on improving the objective value. As time goes by and τ increases, $T(\tau)$ decreases and may even reach 0 eventually. Once T gets small enough, then Simulated Annealing will behave exactly like a hill climber and only accepts a new solution if it is better than the best-so-far solution. This means the algorithm tunes itself from an initial exploration phase to strict exploitation.

Consider the following perspective: An Evolutionary Algorithm allows us to pick a behavior in between a hill climber and a random sampling algorithm by choosing a small or large population size. The Simulated Annealing algorithm allows for a smooth transition of a random search behavior towards a hill climbing behavior over time.

Listing 3.16 An excerpt of the abstract base class for temperature schedules. (src)

```
1 public abstract class TemperatureSchedule {
2   public double startTemperature;
3
4   public abstract double temperature(long tau);
5 }
```

The ingredient needed for this tuning, the temperature schedule, can be expressed as a class implementing exactly one simple function that translates an iteration index τ to a temperature $T(\tau)$, as defined in Listing 3.16.

If we want to apply Simulated Annealing to a given problem, we would like that the probability to accept worse solutions declines smoothly during the optimization process. It should not go down close to 0 too quickly, because then we essentially have a hill climber. It should also not stay too high for too long, because then we waste too much time investigating worse solutions. This means that the right temperature schedule to select will depend on the problem (namely, the range of the objective values) and the computational budget at hand.

Our SA is basically an improved hill climber and, here, we want to solve the JSSP with it. We therefore consider how many iterations the `hc_1swap` from Section 3.3.2.2 performed within the three minutes of runtime. The median total steps range from about 30 million on `swv15` to 97 million on `abz7`. We also know that our objective function is discrete and reasonable values for ΔE are maybe somewhere in the range of 1 to 10. This rough estimate of scale can be can be seen if we look at the differences between the best or median solutions of different algorithm settings in our previous experiments. Hence, we should select temperature schedules that tune the probability of accepting such slightly worse solutions gracefully from relatively high to close-to-zero within 30 million algorithms steps. But how can we do that?

Two common ways to decrease the temperature over time are the *exponential* and the *logarithmic*

temperature schedules, examples for both of which with the desired properties are illustrated in Figure 3.17.

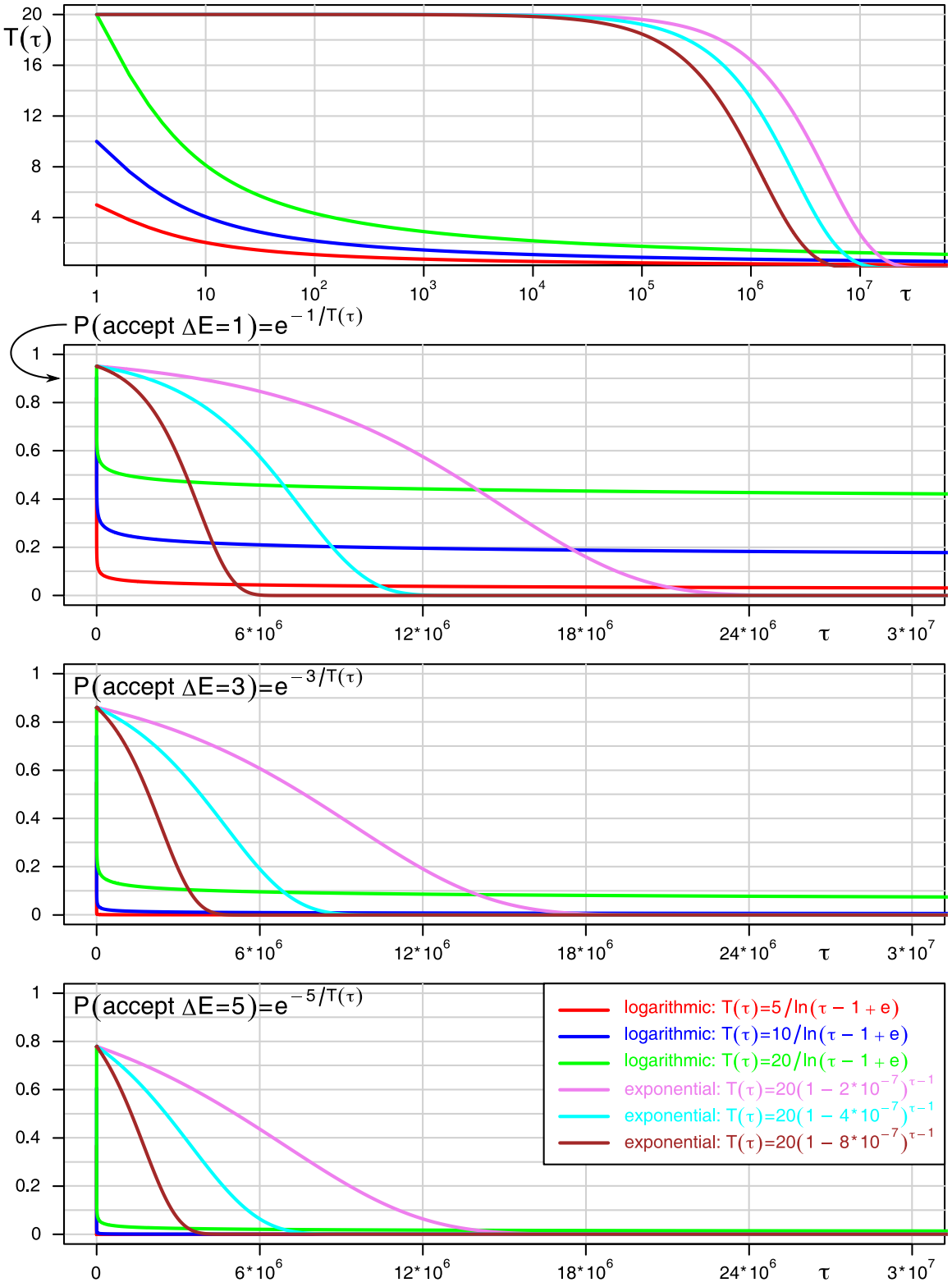

Figure 3.17: The temperature progress of six example temperature schedules (top) plus their probabilities to accept solutions with objective values worse by 1, 3, or 5 than the current solution.

3.5.2.1 Exponential Temperature Schedule

In an exponential temperature schedule, the temperature decreases exponentially with time (as the name implies). It follows Equation (3.3) and is implemented in Listing 3.17. Besides the start temperature T_s, it has a parameter $\epsilon \in (0, 1)$ which tunes the speed of the temperature decrease.

$$T(\tau) = T_s * (1 - \epsilon)^{\tau - 1} \tag{3.3}$$

Listing 3.17 An excerpt of the exponential temperature schedules. (src)

```
public static class Exponential
    extends TemperatureSchedule {
  public double epsilon;

  public double temperature(long tau) {
    return (this.startTemperature
        * Math.pow((1d - this.epsilon), (tau - 1L)));
  }
}
```

Higher values of ϵ lead to a faster temperature decline. In Figure 3.17, we choose the values $\epsilon \in \{2 * 10^{-7}, 4^{-7}, 8 * 10^{-7}\}$ and a starting temperature of $T_s = 20$. As can be seen, they yield a nice and smooth decline of the probabilities to accept solutions slightly worse than the current solution. The probability curves corresponding to the exponential schedules eventually effectively become 0 after about half of the predicted 30 million steps. Notice that we chose the starting temperature T_s and the parameter ϵ in such a way that solutions which are "reasonably worse" in our JSSP scenario are acceptable for a reasonable time in our optimization process, based on our knowledge of the range of objective values that may occur and the number of algorithm steps that will probably be performed. The values of T_s and ϵ are not chosen arbitrarily! They play an important role in the algorithm.

3.5.2.2 Logarithmic Temperature Schedule

The logarithmic temperature schedule will prevent the temperature from becoming very small for a longer time. Compared to the exponential schedule, it will thus longer retain a higher probability to accept worse solutions. It obeys Equation (3.4) and is implemented in Listing 3.18.. It, too, has the parameters $\epsilon \in (0, \infty)$ and T_s.

$$T(\tau) = \frac{T_s}{\ln\left(\epsilon(\tau - 1) + e\right)} \tag{3.4}$$

Listing 3.18 An excerpt of the logarithmic temperature schedules. (src)

```
1  public static class Logarithmic
2      extends TemperatureSchedule {
3    public double epsilon;
4    public double temperature(long tau) {
5      if (tau >= Long.MAX_VALUE) {
6        return 0d;
7      }
8      return (this.startTemperature
9          / Math.log(((tau - 1L) * this.epsilon) + Math.E));
10   }
11 }
```

Larger values of ϵ again lead to a faster temperature decline and we investigated logarithmic schedules with $\epsilon = 1$ for three starting temperatures $T_s \in \{5, 10, 20\}$ in Figure 3.17. Compared to our selected exponential schedules, the temperatures decline earlier but then remain at a higher value. This means that the probability to accept worse candidates in logarithmic schedules remains almost constant (and above 0) after some time. Notice again that the settings of T_s and ϵ are not arbitrary, they are selected so that the probability curve gives a not-too-high and not-too-low acceptance probability to solutions which are not too much worse than the current best solution.

3.5.3 The Algorithm

Now that we have temperature schedules, we can completely define our SA algorithm and implement it in Listing 3.19.

1. Create random point x in search space $\mathbb{X}$ (using the nullary search operator).
2. Map the point x to a candidate solution y by applying the representation mapping $y = \gamma(x)$.
3. Compute the objective value by invoking the objective function $z = f(y)$.
4. Store y in y_b and z in z_b.
5. Set the iteration counter τ to $\tau = 1$.
6. Repeat until the termination criterion is met:
   a. Set $\tau = \tau + 1$.
   b. Apply the unary search operator to x to get the slightly modified copy x' of it.
   c. Map the point x' to a candidate solution y' by applying the representation mapping $y' = \gamma(x')$.
   d. Compute the objective value z' by invoking the objective function $z' = f(y')$.
   e. If $z' \leq z_b$, then

i. Store x' in the variable x and z' in z.
ii. If $z' \leq z_b$, then store y' in the variable y_b and z' in z_b.
iii. Perform next iteration by going to step 6.

f. Compute the temperature T according to the temperature schedule, i.e., set $T = T(\tau)$.
g. If $T \leq 0$ the perform next iteration by goind to step 6.
h. Set $\Delta E = f(\gamma(x)) - f(\gamma(x_b))$ according to Equation (3.1).
i. Compute $P = e^{-\frac{\Delta E}{T}}$ according to Equation (3.2).
j. Draw a random number r uniformly distributed in $[0, 1)$.
k. If $k \leq P$, then store x' in the variable x_b and z' in z_b and perform next iteration by goind to step 6.

7. Return best-so-far objective value z_b and best solution z_b to the user.

There exist a several proofs [74,127] showing that, with a slow-enough cooling schedule, the probability that Simulated Annealing will find the globally optimal solution approaches 1. However, the runtime one would need to invest to actually "cash in" on this promise exceeds the time needed to enumerate all possible solutions [127]. In Section 1.2.1 we discussed that we are using metaheuristics because for many problems, we can only guarantee to find the global optimum if we invest a runtime growing exponentially with the problem scale (i.e., proportional to the size of the solution space). So while we have a proof that SA will eventually find a globally optimal solution, this proof is not applicable in any practical scenario and we instead use SA as what it is: a metaheuristic that will hopefully give us good *approximate* solutions in *reasonable* time.

Listing 3.19 An excerpt of the implementation of the Simulated Annealing algorithm. (src)

```
public class SimulatedAnnealing<X, Y>
    implements IMetaheuristic<X, Y> {
  public TemperatureSchedule schedule;

  public void solve(IBlackBoxProcess<X, Y> process) {
// init local variables x_new, x_cur, nullary, unary, random
// create starting point: a random point in the search space
    nullary.apply(x_cur, random); // put random point in x_cur
    double f_cur = process.evaluate(x_cur); // map & evaluate
    long tau = 1L; // initialize step counter to 1

    do {// repeat until budget exhausted
// create a slightly modified copy of x_cur and store in x_new
      unary.apply(x_cur, x_new, random);
      ++tau; // increase step counter
// map x_new from X to Y and evaluate candidate solution
      double f_new = process.evaluate(x_new);
      if ((f_new <= f_cur) || // accept if better solution OR
          (random.nextDouble() < // probability is e^(-dE/T)
              Math.exp((f_cur - f_new) / // -dE == -(f_new-f_cur)
                  this.schedule.temperature(tau)))) {
// accepted: remember objective value and copy x_new to x_cur
        f_cur = f_new;
        process.getSearchSpace().copy(x_new, x_cur);
      } // otherwise, i.e., f_new >= f_cur: just forget x_new
    } while (!process.shouldTerminate()); // until time is up
  } // process will have remembered the best candidate solution
}
```

3.5.4 Results on the JSSP

Table 3.8: The results of different Simulated Annealing setups compared to the best plain hill climber with restarts and the best basic EA. The columns present the problem instance, lower bound, the algorithm, the best, mean, and median result quality, the standard deviation *sd* of the result quality, as well as the median time *med(t)* and FEs *med(FEs)* until the best solution of a run was discovered. The better values are **emphasized**.

$\mathcal{I}$	$\mathrm{lb}f$	setup	best	mean	med	sd	med(t)	med(FEs)
abz7	656	hcr_256+5%_nswap	707	733	734	7	64s	17’293’038

$\mathcal{I}$	$\mathrm{lb} f$	setup	best	mean	med	sd	med(t)	med(FEs)
		ea4096_nswap_5	685	706	706	10	**29**s	**5'933'332**
		sa_e_20_2e-7_1swap	663	673	672	5	92s	22'456'822
		sa_e_20_4e-7_1swap	**658**	674	675	5	55s	13'388'301
		sa_e_20_8e-7_1swap	663	675	675	6	36s	8'625'161
		sa_l_5_1swap	**658**	675	675	6	63s	15'745'842
		sa_l_10_1swap	659	**672**	**671**	4	86s	21'271'077
		sa_l_20_1swap	675	682	682	**3**	125s	30'740'378
la24	935	hcr_256+5%_nswap	945	981	984	9	57s	29'246'097
		ea4096_nswap_5	941	974	971	13	**6**s	**22'77'833**
		sa_e_20_2e-7_1swap	938	949	946	**8**	27s	12'358'941
		sa_e_20_4e-7_1swap	**935**	949	946	9	16s	7'135'423
		sa_e_20_8e-7_1swap	**935**	951	950	8	9s	4'044'217
		sa_l_5_1swap	940	956	950	13	6s	2'873'837
		sa_l_10_1swap	938	953	950	11	7s	3'210'824
		sa_l_20_1swap	938	**946**	**941**	10	19s	9'097'608
swv15	2885	hcr_256+5%_nswap	3645	3804	3811	44	**91**s	**14'907'737**
		ea4096_nswap_5	3440	3543	3537	51	177s	22'603'785
		sa_e_20_2e-7_1swap	2937	**2990**	**2988**	28	148s	21'949'073
		sa_e_20_4e-7_1swap	2941	2993	2993	28	128s	18'244'751
		sa_e_20_8e-7_1swap	**2936**	3000	3002	28	111s	16'029'528
		sa_l_5_1swap	2963	3032	3029	33	135s	20'087'431
		sa_l_10_1swap	2964	3021	3018	30	141s	21'252'052
		sa_l_20_1swap	2985	3017	3016	**12**	153s	22'596'946
yn4	929	hcr_256+5%_nswap	1081	1117	1119	14	55s	11'299'461
		ea4096_nswap_5	1017	1058	1058	18	**52**s	**8'248'627**
		sa_e_20_2e-7_1swap	973	**985**	**985**	5	113s	20'676'041
		sa_e_20_4e-7_1swap	**971**	987	986	7	68s	12'193'934

$\mathcal{I}$	$\mathrm{lb}f$	setup	best	mean	med	sd	med(t)	med(FEs)
		sa_e_20_8e-7_1swap	972	988	988	7	58s	10'178'219
		sa_l_5_1swap	980	1005	1006	13	75s	13'732'297
		sa_l_10_1swap	975	997	996	11	108s	19'850'143
		sa_l_20_1swap	979	990	990	**4**	116s	21'108'153

In Table 3.8, we now present the results of different setups of our Simulated Annealing algorithm in comparison with the hill climbers with restarts and the best pure EA setup, ea4096_nswap_5. The setups are named after the pattern sa_e_TS_EP_unary have an exponential temperature schedule with the start temperature $T_s = TS$ and $\epsilon = EP$. sa_e_20_8e-7_1swap, for instance, is SA with an exponential temperature schedule with $T_s = 20$ and $\epsilon = 8 * 10^{-7}$ and the 1swap unary operator. The setups named after the pattern sa_l_TS_unary use logarithmic schedules with $\epsilon = 1$, the start temperature $T_s = TS$, and the named unary operator.

What we find from the table is that Simulated Annealing here consistently and significantly outperforms the hill climbers and the best plain EA. On abz7, swv15, and yn4, its mean and median solutions are better than the best solutions offered by these algorithms. Over all, instance la24 could even be solved to optimality and on abz7, we are only 0.3% worse than the lower bound of the objective function. The median solutions of sa_e_20_4e-7_1swap are illustrated in Figure 3.18. For abz7, they are only 3% longer than the theoretical lower bound (656), 1.1% for la24, 4% for swv15, and 6% for yan4. We also tested the Simulated Annealing setups with the unary nswap operator, but this did not yield further improvements.

If we compare our sa_e_20_4e-7_1swap with the related work, we find its best and mean solution quality on abz7 surpass those of the four Genetic Algorithms in [100] as well as those of the original Fast Simulated Annealing algorithm and its improved version HFSAQ from [4]. The best result is better than the one of the TGA in [8]. Its mean and best results of sa_e_20_4e-7_1swap on la24 outperform all algorithms from [2,98,100] and the mean results are also better than the results of the aLSGA in [11]. On yn4, it outperforms all four AntGenSA algorithms (complex hybrids of three algorithms including SA and EAs) in [88] in mean and best result quality. So while we are not shooting for solving the JSSP outstandingly well using very complicated algorithms, our simple take on the problem seems to work.

In Figure 3.19, we compare the progress over time of our Simulated Annealing setups with those of the best hill climber with restarts. We find a very significant difference on three of the four problem instancee. The higher similarity of the end result distribution on la24 results from the fact that even

`hcr_256+5%_nswap` produces schedules which are less than 5% longer than the best possible one (objective function lower bound) in median.

We also find that the two SA approaches have qualitatively different behavior. The setup with the logarithmic schedule improves the solution quality a bit similar to the hill climber but eventually yields better results, as it can escape from local optima. The setup with the exponential schedule progresses initially more slowly, but at some point suddenly speeds up. These two behaviors fit exactly the temperature schedule and acceptance probability illustrations in Figure 3.17: While the temperature and acceptance probability of the logarithmic schedule slowly decrease and remain at a slightly higher level, there is a clear phase transition in the exponential schedule. Both the temperature and acceptance probability remain higher for some time until they suddenly drop. Interestingly, the objective value of the best-so-far solution in SA seems to follow that pattern.

This also means: It is very important to have the right temperature schedule. We obtained the right temperature schedule because we know *a)* the reasonable range of good objective values and *b)* roughly how many algorithm steps we can perform within our computational budget.

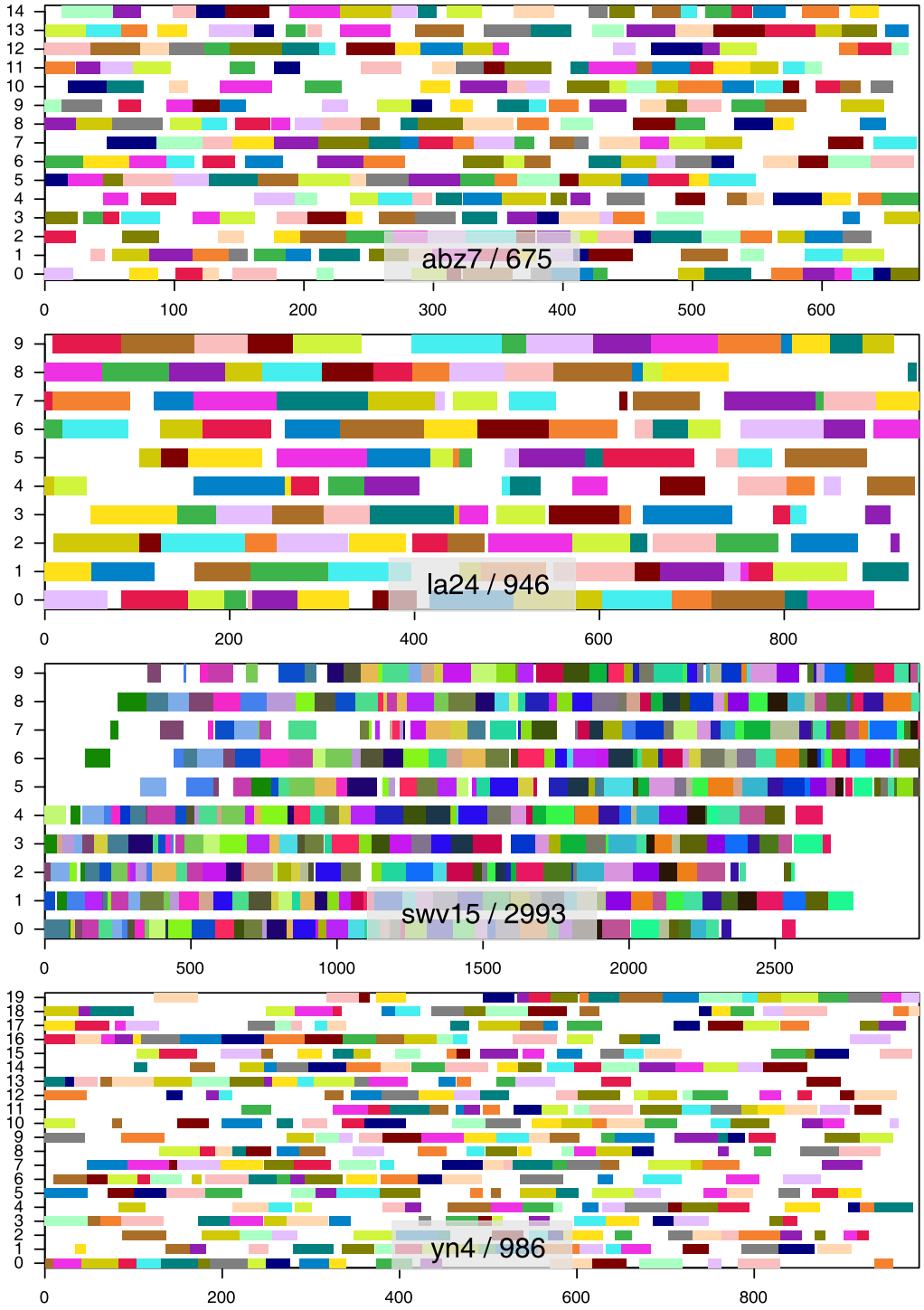

Figure 3.18: The Gantt charts of the median solutions obtained by the `sa_e_20_4e-7_1swap` setup. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

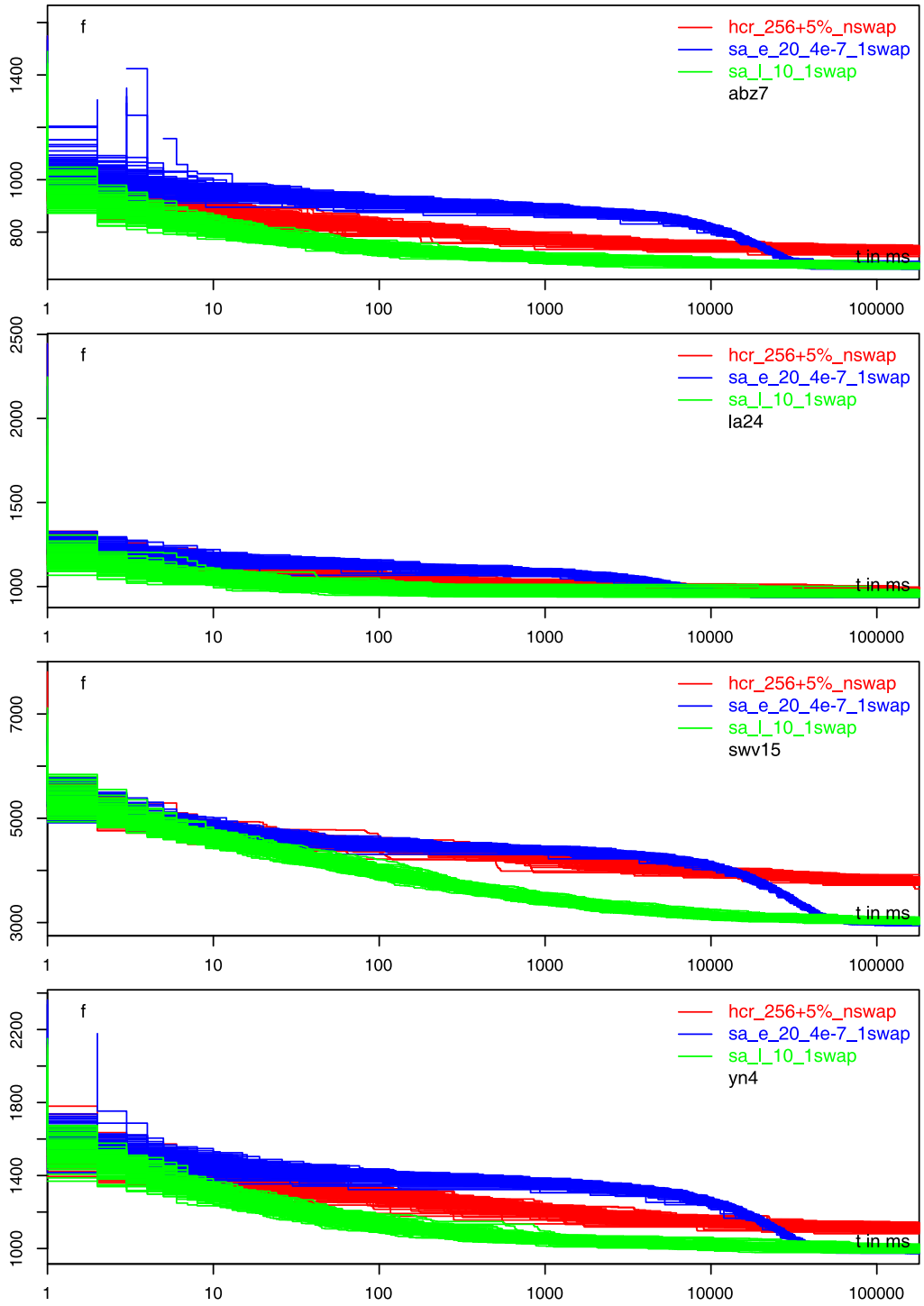

Figure 3.19: The progress of the two Simulated Annealing setups `sa_e_20_4e-7_1swap` and `sa_l_10_1swap` compared with the best basic hill climber with restarts `hcr_256+5%_nswap`, over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis).

3.6 Hill Climbing Revisited

Until now, we have entirely relied on randomness to produce new points in the search space. The results of our nullary, unary, and binary operators are all random. In case of the unary and binary operator, they of course depend on the input points in the search space fed to the operators, but still, the results are unpredictable and random. This is, in general, not a bad property. In the absence of knowledge about what is best, doing an arbitrary thing might have a better expected outcome than doing a fixed, pre-determined thing.

However, it also has some drawbacks. For example, there is no guarantee to not test the same `1swap` move several times in the `hc_1swap` algorithm. Also, since we do not know when we have tested the complete neighborhood of a point x in the search space, we also do not know whether x is a (local) optimum or not. We instead need to guess this and in Section 3.3.3 we therefore design an algorithm that restarts if it did not encounter an improvement for a certain time. This might be too early, as there may still be undiscovered solutions in the neighborhood of x – or it might be too late and we may have already investigated the complete neighborhood several times.

Let us take one step back, to the simple hill climber and the original unary search operator `1swap` for the JSSP from Section 3.3.1. This operator tries to perform a single swap, i.e., exchange the order of two job IDs in a point from the search space. We already discussed in Section 3.3.4 that the size of this neighborhood is $0.5 * m^2 * n * (n-1)$ for each point in the search space.

3.6.1 Idea: Enumerating Neighborhoods

Instead of randomly sampling elements from this neighborhood, we could simple iteratively and exhaustively enumerate over them. As soon as we encounter an improvement, we can stop and accept the better point. If we have finished enumerating all possible `1swap` neighbors and none of them yields a candidate solution with better objective value (e.g., a Gantt chart with shorter makespan), we know that we have arrived in a local optimum. This way, we do no longer need to "guess" if we have converged or not, we know it directly. Also, as detailed in Section 6.1.2, we should be able to find an improving move faster in average, because we will never redundantly sample the same point in the search space again when investigating the neighborhood of the current best solution.

Implementing this concept is a little bit more complicated than creating the simple unary operator that just returns one single new point in the search space as a result. Instead, such an enumerating unary operator for a black-box metaheuristic may create any number of points. Moreover, if one of the new points already maps to a candidate solutions which can improve upon the current best solution, then maybe we wish to terminate the enumeration process at that point.

Such behavior can be realized by following a visitor design pattern. An enumerating unary operator will receive a point x in the search space and a call-back function from the optimization process. Every time it creates a neighbor x' of x, it will invoke the call-back function and pass x' to it. If the function returns, say `true`, then the enumeration will be terminated, while it is continued for `false`. The optimization process, in the call-back function, could apply the representation mapping γ to x' and compute the objective value of the resulting candidate solution. If that solution is better than x, it could store it and return `true`. Otherwise, it would return `false` and be fed with the next neighbor, until the neighborhood was exhaustively enumerated.

This idea can be implemented by extending our original interface `IUnarySearchOperator` for unary search operations given in Listing 2.9.

Listing 3.20 A the generic interface for unary search operators, now able to enumerate neighborhoods. (src)

```
1  public interface IUnarySearchOperator<X> {
2    public abstract void apply(X x, X dest,
3        Random random);
4    public default boolean enumerate(Random random,
5        X x, X dest, Predicate<X> visitor) {
6      throw new UnsupportedOperationException("The operator " +
7          this.getClass().getName() +
8          " does not support exhaustive enumeration of neighborhoods.");
9    }
10 }
```

The extension, presented in Listing 3.20, is a single new function, `enumerate`, which should realize the neighborhood enumeration. This function receives an existing point `x` in the search space as input, as well as a destination data structure `dest` where, iteratively, the neighboring points of `x` should be stored. Additionally, a call-back function `visitor` is provided as implementation Java 8-interface `Predicate`. The `test` function function of this interface will, upon each call, receive the next neighbor of `x` (stored in `dest`). It returns `true` when the enumeration should be stopped (maybe because a better solution was discovered) and `false` to continue. `enumerate` itself will return `true` if and only if `test` ever returned `true` and `false` otherwise.

Of course, we cannot implement a neighborhood enumeration for all possible unary operators: In the case of the `nswap`, operator, for instance, all other points in the search space could potentially be reached from the current one. Enumerating this neighborhood would include the complete search space and would take way too long. Hence, the `default` implementation of the new method should just create an error. It will only be overwritten by operators with a neighborhood sufficiently small for efficient enumeration. A usual limit is neighborhood whose size grows quadratically with the problem

scale, as is the case here, or at most with the third power of the problem scale.

3.6.2 Ingredient: Neighborhood Enumerating 1swap Operation for the JSSP

Let us now consider how such an exhaustive enumeration of the neighborhood spanned by the 1swap operator can be implemented.

1. Make a copy x' of the input point x from the search space.
2. For index i from 1 to $m * n - 1$ do:
   a. Store the job at index i in x' in variable job_i.
   b. For index j from 0 to $i - 1$ do:
      i. Store the job at index j in x' in variable job_j.
      ii. If $job_i \neq job_j$ then:
         1. Store job_i at index j in x'.
         2. Store job_j at index i in x'.
         3. Pass x' to a call-back function of the optimization process. If the function indicates that it wishes to terminate the enumeration, then quit. Otherwise continue with the next step.
         4. Store job_i at index i in x'.
         5. Store job_j at index j in x'.

This simple algorithm is implemented in Listing 3.21, which only shows the new function that was added to our class `JSSPUnaryOperator1Swap` that we had already back in Section 3.3.1.

Listing 3.21 An excerpt of the `1swap` operator for the JSSP, namely the implementation of the `enumerate` function from the interface `IUnarySearchOpertor` (Listing 3.20). (src)

```
public boolean enumerate(Random random,
    int[] x, int[] dest,
    Predicate<int[]> visitor) {
  int i = x.length; // get the length
  System.arraycopy(x, 0, dest, 0, i); // copy x to dest
  for (; (--i) > 0;) { // iterate over all indices 1..(n-1)
    int job_i = dest[i]; // remember job id at index i
    for (int j = i; (--j) >= 0;) { // iterate over 0..(i-1)
      int job_j = dest[j]; // remember job at index j
      if (job_i != job_j) { // both jobs are different
        dest[i] = job_j; // then we swap the values
        dest[j] = job_i; // and will then call the visitor
        if (visitor.test(dest)) {
          return true; // visitor says: stop -> return true
        } // visitor did not say stop, so we need to
        dest[i] = job_i; // revert the change
        dest[j] = job_j; // and continue
      } // end of creation of different neighbor
    } // end of iteration via index j
  } // end of iteration via index i
  return false; // we have enumerated the complete neighborhood
}
```

3.6.3 Hill Climbing Algorithm based on Neighborhood Enumeration

3.6.3.1 The Algorithm

The new variant of the hill climber would then be able to step-by-step enumerating the neighborhood of the current best point z_b from the search space spanned by a unary operator. As soon as it discovers an improvement with respect to the objective function, the new, better point replaces z_b. The neighborhood enumeration then starts again from there, until the termination criterion is met. The general pattern of this algorithm is given below:

1. Create random point x in search space $\mathbb{X}$ (using the nullary search operator).
2. Map the point x to a candidate solution y by applying the representation mapping $y = \gamma(x)$.
3. Compute the objective value by invoking the objective function $z = f(y)$.
4. Store x in the variable x_b and z in z_b.
5. Repeat until the termination criterion is met:

   a. For each point x' in the search space neighboring to the current best point x_b according to the unary search operator do:
      i. Map the point x' to a candidate solution y' by applying the representation mapping $y' = \gamma(x')$.
      ii. Compute the objective value z' by invoking the objective function $z' = f(y')$.
      iii. If $z' < z_b$, then store x' in the variable x_b, z' in z_b, and stop the enumeration (go back to step *5*).
   b. If we arrive here, the neighborhood of z_b did not contain any better solution. So we can stop the algorithm by going to step 6.
6. Return best-so-far objective value and best solution to the user.

If we want to implement this algorithm for black-box optimization, we face the situation that the algorithm does not know the nature of the search space nor the neighborhood spanned by the operator. Therefore, we rely on the design introduced in Section 3.6.1, which allows us to realize this implicitly unknown looping behavior (point *a* above) in form of the visiting pattern. The idea is that, while our hill climber does not know how to enumerate the neighborhood, the unary operator does, since it defines the neighborhood. The resulting code is given in Listing 3.22.

We find that this algorithm can quite its main loop early: If the complete enumeration of the neighborhood of the current best solution x_b yields no improvement, we can stop. It makes no sense to enumerate the same neighborhood of the same solution again. What would make sense here would be to *restart* the search at a different, random point in the search space.

Listing 3.22 An excerpt of the implementation of the neighborhood-enumerating Hill Climbing algorithm, which remembers the best-so-far solution and tries to find better solutions by iteratively investigating the solutions in its neighborhood until it finds an improvement. (src)

```
1  public class HillClimber2<X, Y>
2      implements IMetaheuristic<X, Y> {
3
4    public void solve(IBlackBoxProcess<X, Y> process) {
5  // init local variables x_cur, x_best, nullary, unary, random,
6  // f_best, improved: omitted here for brevity
7  // create starting point: a random point in the search space
8      nullary.apply(x_best, random); // put random point in x_best
9      double[] f_best = { process.evaluate(x_best) }; // evaluate
10
11     while (improved && !process.shouldTerminate()) {
12 // repeat until budget exhausted or no improving move
13 // enumerate all neighboring solutions of x_best and receive them
14 // one-by-one in parameter x (for which x_cur is used)
15       improved = unary.enumerate(random, x_best, x_cur, (x) -> {
16 // map x from X to Y and evaluate candidate solution
17         double f_cur = process.evaluate(x);
18         if (f_cur < f_best[0]) { // we found a better solution
19 // remember best objective value and copy x to x_best
20           f_best[0] = f_cur;
21           process.getSearchSpace().copy(x, x_best);
22           return true; // quit enumerating neighborhood
23         }
24 // no improvement: continue enumeration unless time is up
25         return process.shouldTerminate();
26       });
27 // repeat until time is up or no further improvement possible
28     }
29
30   } // process will have remembered the best candidate solution
31 }
```

3.6.4 Hill Climbing Algorithm based on Neighborhood Enumeration with Restarts

The very idea of using an enumerate-able neighborhood was to be able to do restarts more efficiently compared to our original hill climber. As planned, we are now freed from the need to "guess" when we have to restart. Instead, we should restart exactly when we have finished enumerating the neighborhood of the current best solution without discovering an improvement.

3.6.4.1 The Algorithm

1. Set the overall-best objective value z_B to infinity and the overall-best candidate solution y_B to NULL.
2. Create random point x in search space $\mathbb{X}$ (using the nullary search operator).
3. Map the point x to a candidate solution y by applying the representation mapping $y = \gamma(x)$.
4. Compute the objective value by invoking the objective function $z = f(y)$.
5. Store x in the variable x_b and z in z_b.
6. If $z_b < z_B$, then set z_B to z_b and store $y_B = \gamma x$.
7. Repeat until the termination criterion is met:
   a. For each point x' in the search space neighboring to the current best point x_b according to the unary search operator do:
      i. Map the point x' to a candidate solution y' by applying the representation mapping $y' = \gamma(x')$.
      ii. Compute the objective value z' by invoking the objective function $z' = f(y')$.
      iii. If $z' < z_b$, then store x' in the variable x_b, z' in z_b, and stop the enumeration (go back to step *6*).
   b. If we arrive here, the neighborhood of z_b did not contain any better solution. Hence, we perform a restart by going back to point 2.
8. Return **best ever encountered** objective value z_B and solution y_B to the user.

Different from Section 3.3.3.1, this new algorithm does not need to count steps or even manage a parameter regarding how often to restart. Its implementation in Listing 3.23 is therefore also shorter and simpler than the implementation of the original algorithm variant in Listing 3.11. It should be noted that both new hill climbers can only be applied in scenarios where we actually can enumerate the neighborhoods of the current best solutions efficiently. In other words, we pay for a potential gain of search efficiency by a reduction of the types of problems we can process.

3.6.4.2 Results on the JSSP

Table 3.9: The results of the neighborhood-enumerating hill climber with (`hc2r_1swap`) and without (`hc2_1swap`) restarts in comparison with the "original" hill climbers from Section 3.3. The columns present the problem instance, lower bound, the algorithm, the best, mean, and median result quality, the standard deviation *sd* of the result quality, as well as the median time *med(t)* and FEs *med(FEs)* until the best solution of a run was discovered. The better values are **emphasized**.

$\mathcal{I}$	$\mathrm{lb}f$	setup	best	mean	med	sd	med(t)	med(FEs)
abz7	656	hc_1swap	717	800	798	28	**0**s	**16978**
		hc2_1swap	723	789	786	30	3s	737235
		hcr_256+5%_1swap	723	742	743	**7**	21s	5681591
		hc2r_1swap	**705**	734	736	8	84s	23244617
		hcr_256+5%_nswap	707	**733**	**734**	7	64s	17293038
la24	935	hc_1swap	999	1095	1086	56	**0**s	**6612**
		hc2_1swap	1004	1102	1092	55	0s	99601
		hcr_256+5%_1swap	970	997	998	9	6s	3470368
		hc2r_1swap	959	**977**	**977**	**8**	78s	43179265
		hcr_256+5%_nswap	**945**	981	984	9	57s	29246097
swv15	2885	hc_1swap	3837	4108	4108	137	**1**s	**104598**
		hc2_1swap	3685	3982	3974	153	24s	3708826
		hcr_256+5%_1swap	3701	3850	3857	**40**	60s	9874102
		hc2r_1swap	**3628**	**3797**	**3799**	66	112s	17325313
		hcr_256+5%_nswap	3645	3804	3811	44	91s	14907737
yn4	929	hc_1swap	1109	1222	1220	48	**0**s	**31789**
		hc2_1swap	1121	1203	1198	50	9s	1905085
		hcr_256+5%_1swap	1095	1129	1130	14	22s	4676669
		hc2r_1swap	**1076**	1125	1124	17	89s	18869590
		hcr_256+5%_nswap	1081	**1117**	**1119**	**14**	55s	11299461

In Table 3.9, we compare this new neighborhood-enumerating hill climbers (prefix `hc2`) with the "original" hill climbers from Section 3.3.

We find that the non-restarting variant hc2_1swap from Section 3.6.3.1 outperforms hc_1swap except for instance la24. We can also see that it can require several seconds (3s for abz7 and 24s for swv15) until it arrives in a local optimum from which it can no longer escape with 1swap moves.

hc2r_1swap is the algorithm version with restarts, i.e., once it finds a point in the search space whose neighborhood, given in Section 3.6.4. It performs better than its non-enumerating counterpart hcr_256+5%_1swap and sometimes also than hcr_256+5%_nswap which was the best original hill climber due to its nswap operator with a larger neighborhood (see Section 3.3.4.1).

The improvements are small, but they are there.

Listing 3.23 An excerpt of the implementation of the Hill Climbing algorithm with restarts based on neighborhood enumeration. (src)

```
public class HillClimber2WithRestarts<X, Y>
    implements IMetaheuristic<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
// initialization of local variables x_cur, x_best, nullary,
// unary, random omitted for brevety
    while (!process.shouldTerminate()) { // main loop
// create starting point: a random point in the search space
// put random point in x_best
      nullary.apply(x_best, random);
      f_best[0] = process.evaluate(x_best); // evaluate

      do {// repeat until budget exhausted or no improving move
// enumerate all neighboring solutions of x_best and receive them
// one-by-one in parameter x (for which x_cur is used)
        improved = unary.enumerate(random, x_best, x_cur,
            (x) -> {
// map x from X to Y and evaluate candidate solution
              double f_cur = process.evaluate(x);
              if (f_cur < f_best[0]) { // found better solution
// remember best objective value and copy x to x_best
                f_best[0] = f_cur;
                process.getSearchSpace().copy(x, x_best);
                return true; // quit enumerating neighborhood
              }
// no improvement: continue enumeration unless time is up
              return process.shouldTerminate();
            });
// repeat until time is up or no further improvement possible
        if (process.shouldTerminate()) {
          return; // ok, we should exit
        } // otherwise: continue inner loop as long as we
      } while (improved); // can find improvements
    } // outer loop: if we get here, we need to restart
  } // process will have remembered the best candidate solution
}
```

3.7 Memetic Algorithms: Hybrid of Global and Local Search

We now have seen two types of efficient algorithms for solving optimization problems:

1. local search methods, like the hill climbers, that can refine and improve one solution quickly but may get stuck at local optima, and
2. global search methods, like evolutionary algorithms, which try to preserve a diverse set of solutions and are less likely to end up in local optima, but pay for it by slower optimization speed.

It is a natural idea to combine both types of algorithms, to obtain a hybrid algorithm which unites the best from both worlds. Such algorithms are today often called *Memetic Algorithms* (MAs) [85,122,126] (sometimes also Lamarkian Evolution [178]).

3.7.1 Idea: Combining Local Search and Global Search

The idea is as follows: In an Evolutionary Algorithm, the population guards against premature convergence to a local optimum. In each generation of the EA, new points in the search space are derived from the ones that have been selected in the previous step. This means that, from the perspective of a single point in the population, each generation of the EA is similar to one iteration of a hill climber. However, there are μ~points in the population, not just one. As a result, the progress made towards a good solution is much slower compared to the hill climber.

Another issue is that we introduced a binary search operator which combines traits from two points in the population to form a new, hopefully better solution. The idea is that the points that have survived selection should be good, hence they should include good components, and we hope to combine these. However, during the early stages of the search, the population contains first random and then slightly refined points (see above). For quite some time, these will not yet be good and thus neither contain good components.

Both of these issues can be mitigated by one simple idea: Let each new point, before it enters the population, become the starting point of a local search that runs until it converges and then enter the result of this local search into the population instead. This is already the concept of a Memetic Algorithm.

As a result, the first generation of the MA performs exactly the same as a Hill Climber with restarts Section 3.6.4. The inputs of the binary search operator will then not just be selected points, they will be local optima (with respect to the neighborhood spanned by the unary operator). Actually, we can omit the unary operator in the MA as it is already used in the local search and always apply the binary operator to generate new points. In the following generations, the local search will then refine the combinations of local optima.

3.7.2 Algorithm: EA Hybridized with Neighborhood-Enumerating Hill Climber

The basic $(\mu + \lambda)$ Memetic Algorithm is given below and implemented in Listing 3.24.

1. $I \in \mathbb{X} \times \mathbb{R}$ be a data structure that can store one point x in the search space and one objective value z.
2. Allocate an array P of length $\mu + \lambda$ instances of I.
3. For index i ranging from 0 to $\mu + \lambda - 1$ do
   a. Store a randomly chosen point from the search space in $P_i.x$.
4. Repeat until the termination criterion is met:
   b. For index i ranging from μ to $\mu + \lambda - 1$ do
      i. Apply the representation mapping $y = \gamma(P_i.x)$ to get the corresponding candidate solution y. ii Compute the objective objective value of y and store it at index i as well, i.e., $P_i.z = f(y)$.
      ii. **Local Search:** For each point x' in the search space neighboring to $P_i.x$ according to the unary search operator do:
         1. Map the point x' to a candidate solution y' by applying the representation mapping $y' = \gamma(x')$. 2 Compute the objective value z' by invoking the objective function $z' = f(y')$.
         2. If the termination criterion has been met, jump directly to step 5.
         3. If $z' < z_b$, then store x' in the variable $P_i.x$, z' in $P_i.z$, stop the enumeration, and go back to step *4b.iii*.
   c. Sort the array P according to the objective values such that the records with better associated objective value z are located at smaller indices. For minimization problems, this means elements with smaller objective values come first.
   d. Shuffle the first μ elements of P randomly.
   e. Set the first source index $p = -1$.
   f. For index i ranging from μ to $\mu + \lambda - 1$ do
      iv. Set the source index p to $p = (p + 1) \bmod \mu$, i.e., make sure that every one of the μ selected points is used approximately the same number of times.
      v. Randomly choose another index $p2$ from $0 \ldots (\mu - 1)$ such that $p2 \neq p$.
      vi. Set $P_i.x = \mathrm{searchOp}_2(P_p.x, P_{p2}.x)$, i.e., derive a new point in the search space for the record at index i by applying the binary search operator to the points stored at index p and $p2$.

5. Return the candidate solution corresponding to the best record in P to the user.

3.7.2.1 Results on the JSSP

Listing 3.24 An excerpt of the implementation of the Memetic Algorithm algorithm. (src)

```
public class MA<X, Y> implements IMetaheuristic<X, Y> {
  public void solve(IBlackBoxProcess<X, Y> process) {
// the initialization of local variables is omitted for brevity
// first generation: fill population with random individuals
    for (int i = P.length; (--i) >= 0;) {
// set P[i] = random individual (code omitted)
    }
    int localSearchStart = 0; // at first, apply ls to all

    while (!process.shouldTerminate()) { // main loop
      for (int i = P.length; (--i) >= localSearchStart;) {
        Individual<X> ind = P[i];
// refine P[i] with local search à la HillClimber2 (code omitted)
// for a given number of maximum steps
// sort the population: mu best individuals at front are selected
      Arrays.sort(P);
// shuffle the first mu solutions to ensure fairness
      RandomUtils.shuffle(random, P, 0, this.mu);
      int p1 = -1; // index to iterate over first parent

// override the worse lambda solutions with new offsprings
      for (int index = P.length; (--index) >= this.mu;) {
        Individual<X> dest = P[index];
        Individual<X> sel = P[(++p1) % this.mu];

        do { // find a second, different record
          p2 = random.nextInt(this.mu);
        } while (p2 == p1);
// perform recombination of the two selected individuals
        binary.apply(sel.x, P[p2].x, dest.x, random);
// map to solution/schedule and evaluate quality
        dest.quality = process.evaluate(dest.x);
      } // the end of the offspring generation

      localSearchStart = this.mu;
    } // the end of the main loop
  }
}
```

4 Evaluating and Comparing Optimization Algorithms

We have now learned quite a few different approaches for solving optimization problems. Whenever we have introduced a new algorithm, we have compared it with some of the methods we have discussed before.

Clearly, when approaching an optimization problem, our goal is to solve it in the best possible way. What the best possible way is will depend on the problem itself as well as the framework conditions applying to us, say, the computational budget we have available.

It is important that *performance* is almost always relative. If we have only a single method that can be applied to an optimization problem, then it is neither good nor bad, because we can either take it or leave it. Instead, we often start by first developing one idea and then try to improve it. Of course, we need to compare each new approach with the ones we already have. Alternatively, especially if we work in a research scenario, maybe we have a new idea which then needs to be compared to a set of existing state-of-the-art algorithms. Let us now discuss here how such comparisons can be conducted in a rigorous, reliable, and reproducible way.

4.1 Testing and Reproducibility as Important Elements of Software Development

The very first and maybe one of the most important issues when evaluating an optimization algorithms is that you *never* evaluate an optimization algorithm. You always evaluate an *implementation* of an optimization algorithm. You always compare *implementations* of different algorithms.

Before we even begin to think about running experiments, we need to be assert whether our algorithm implementations are correct. In almost all cases, it is not possible to proof whether a software is implemented correctly or not. However, we can apply several measures to find potential errors.

4.1.1 Unit Testing

A very important tool that should be applied when developing a new optimization method is unit testing. Here, the code is divided into units, each of which can be tested separately.

In this book, we try to approach optimization in a structured way and have defined several interfaces for the components of an optimization and the representation in chapter 2. An implementation of such an interface can be considered as a unit. The interfaces define methods with input and output values. We now can write additional code that tests whether the methods behave as expected, i.e., do not violate their contract. Such unit tests can be executed automatically. Whenever we compile our software after changing code, we can also run all the tests again. This way, we are very likely to spot a lot of errors before they mess up our experiments.

In the Java programming language, the software framework JUnit provides an infrastructure for such testing. In the example codes of our book, in the folder src/test/java, we provide JUnit tests for general implementations of our interfaces as well as for the classes we use in our JSSP experiment.

Here, the encapsulation of different aspects of black-box optimization comes in handy. If we can ensure that the implementations of all search operations, the representation mapping, and the objective function are correct, then our implemented black-box algorithms will – at least – not return any invalid candidate solutions. The reason is that they use exactly only these components (along with utility methods in the `ISpace` interface which we can also test) to produce solutions. A lot of pathological errors can therefore be detected early.

Always develop the tests either before or at least along with your algorithm implementation. Never say "I will do them later." Because you won't. And if you actually would, you will find errors and then repeat your experiments.

4.1.2 Reproducibility

A very important aspect of rigorous research is that experiments are reproducible. It is extremely important to consider reproduciblity *before* running the experiments. From personal experiments, I can say that sometimes, even just two or three years after running the experiments, I have looked at the collected data and did no longer know, e.g., the settings of the algorithms. Hence, the data became useless. The following measures can be taken to ensure that your experimental results are meaningful to yourself and others in the years to come:

1. Always use self-explaining formats like plain text files to store your results.
2. Create one file for each run of your experiment and *automatically* store at least the following information [164,167]:

i. the algorithm name and all parameter settings of the algorithm,
ii. the relevant measurements, i.e., the logged data,
iii. the seed of the pseudo-random number generator used,
iv. information about the problem instance on which the algorithm was applied,
v. short comments on how the above is to be interpreted,
vi. maybe information about the computer system your code runs on, maybe the Java version, etc., and
vii. maybe even your contact information. This way, you or someone else can, next year, or in ten years from now, read your results and get a clear understanding of "what is what." Ask yourself: If I put my data on my website and someone else downloads it, does every single file contain sufficient information to understand its content?

3. Store the files and the compiled binaries of your code in a self-explaining directory structure [164,167]. I prefer having a base folder with the binaries that also contains a folder `results`. `results` then contains one folder with a short descriptive name for each algorithm setup, which, in turn, contain one folder with the name of each problem instance. The problem instance folders then contain one text file per run. After you are done with all experiments and evaluation, such folders lend them self for compression, say in the `tar.xz` format, for long-term archiving.
4. Write your code such that you can specify the random seeds. This allows to easily repeat selected runs or whole experiments. All random decisions of an algorithm depend on the random number generator (RNG). The "seed" (see *point 2.iii* above) is an initialization value of the RNG. If I initialize the (same) RNG with the same seed, it will produce the same sequence of random numbers. If I know the random seed used for an experiment, then I can start the same algorithm again with the same initialization of the RNG. Even if my optimization method is randomized, it will then make the same "random" decisions. In other words, you should be able to repeat the experiments in this book and get more or less identical results. There might be differences if Java changes the implementation of their RNG or if your computer is significantly faster or slower than mine, though.
5. Ensure that all random seeds in your experiments are generated in a deterministic way in your code. This can be a proof that you did not perform cherry picking during your experiments, i.e., that you did not conduct 1000 runs and picked only the 101 where your newly-invented method works best. In other words, the seeds should come from a reproducible sequence, say the same random number generator, but seeded with the MD5 checksum of the instance name. This would also mean that two algorithms applied to the same instance have the same random seed and may therefore start at the same random point.
6. Clearly document and comment your code. In particular, comment the contracts of each method such that you can properly verify them in unit tests. Never say "I document the code when I am finished with my work." Because you won't.

7. Prepare your code from the very beginning as if you would like to put it on your website. Prepare it with the same care and diligence you want to see your name associated with.
8. If you are conducting research work, consider to publish both your code and data online:
    a. For code, several free platforms such as GitHub or bitbucket exist. These platforms often integrate with free continuous integration platforms, which can automatically compile your code and run your unit tests when you make a change.
    b. For results, there, too, are free platforms such as zenodo. Using such online repositories also protects us from losing data. This is also a great way to show what you are capable of to potential employers…
9. If your code depends on external libraries or frameworks, consider using an automated dependency management and build tool. For the code associated with this book, I use Apache Maven, which ensures that my code is compiled using the correct dependencies (e.g., the right JUnit version) and that the unit tests are executed on each built. If I or someone else wants to use the code later again, the chances are good that the build tool can find the same, right versions of all required libraries.

From the above, I think it should have become clear that reproducibility is nothing that we can consider after we have done the experiments. Hence, like the search for bugs, it is a problem we need to think about beforehand. Several of the above are basic suggestions which I found useful in my own work. Some of them are important points that are necessary for good research and which sadly are never mentioned in any course.

4.2 Measuring Time

Let us investigate the question: "What does good optimization algorithm performance mean?" As a first approximation, we could state that an optimization algorithm performs well if it can solve the optimization problem to optimality. If two optimization algorithms can solve the problem, then we prefer the faster one. This brings us to the question what *faster* means. If we want to compare algorithms, we need a concept of time.

4.2.1 Clock Time

Of course, we already know a very well-understood concept of time. We use it every day: the clock time. In our experiments with the JSSP, we have measured the runtime mainly in terms of milliseconds that have passed on the clock as well.

Definition 28. The consumed *clock time* is the time that has passed since the optimization process was started.

This has several *advantages*:

- Clock time is a quantity which makes physical sense and which is intuitive clear to us.
- In applications, we often have well-defined computational budgets and thus need to know how much time our processes really need.
- Many research works report the consumed runtime, so there is a wide basis for comparisons.
- If you want to publish your own work, you should report the runtime that your implementation of your algorithm needs as well.
- If we measure the runtime of your algorithm implementation, it will include everything that the code you are executing does. If your code loads files, allocates data structures, or does complicated calculations – everything will be included in the measurement.
- If we can parallelize or even distribute our algorithms, clock time measurements still make sense.

But reporting the clock time consumed by an algorithm implementation also has *disadvantages*:

- The measured time strongly depends on your computer and system configuration. Runtimes measured on different machines or on different system setups are therefore inherently incomparable or, at least, it is easy to make mistakes here. Measured runtimes reported twenty years ago are basically useless now, unless they differ from current measurements very significantly, by orders of magnitudes.
- Runtime measurements also are measurements based on a given *implementation*, not *algorithm*. An algorithm implemented in the C programming language may perform very different compared to the very same algorithm implemented in `Java`. An algorithm implementation using a hash map to store and retrieve certain objects may perform entirely different from the same algorithm implemented using a sorted list. Hence, effort should be invested to create good implementations before measuring their consumed runtime and, very important, the same effort should be invested into all compared algorithms…
- Runtime measurements are not always very accurate. There may be many effects which can mess up our measurements, ranging from other processes being executed on the same system and slowing down our process, delays caused by swapping or paging, to shifts of CPU speeds due to dynamic CPU clocking.
- Runtime measurements are not very precise. Often, clocks have resolutions only down to a few milliseconds, and within even a millisecond many action can happen on today's CPUs.

4.2.2 Consumed Function Evaluations

Instead of measuring how many milliseconds our algorithm needs, we often want a more abstract measure. Another idea is to count the so-called (objective) *function evaluations* or FEs for short.

Definition 29. The consumed *function evaluations* (FEs) are the number of calls to the objective function issued since the beginning of the optimization process.

Performing one function evaluation means to take one point from the search space $x \in \mathbb{X}$, map it to a candidate solution $y \in \mathbb{Y}$ by applying the representation mapping $y = \gamma(x)$ and then computing the quality of y by evaluating the objective function $f(y)$. Usually, the number of FEs is also equal to the number of search operations applied, which means that each FE includes one application of either a nullary, unary, or binary search operator. Counting the FEs instead of measuring time directly has the following *advantages*:

- FEs are completely machine- and implementation-independent and therefore can more easily be compared. If we re-implement an algorithm published 50 years ago, it should still consume the same number of FEs.
- Counting FEs is always accurate and precise, as there cannot be any outside effect or process influencing the measurement (because that would mean that an internal counter variable inside of our process is somehow altered artificially).
- Results in many works are reported based on FEs or in a format from which we can deduce the consumed FEs.
- If you want to publish your research work, you should probably report the consumed FEs as well.
- In many optimization processes, the steps included in an FE are the most time consuming ones. Then, the actual consumed runtime is proportional to the consumed FEs and "performing more FEs" roughly equals to "needing more runtime."
- Measured FEs are something like an empirical, simplified version of algorithmic time complexity. FEs are inherently close to theoretical computer science, roughly equivalent to "algorithm steps," which are the basis for theoretical runtime analysis. For example, researchers who are good at Maths can go an derive things like bounds for the "expected number of FEs" to solve a problem for certain problems and certain algorithms. Doing this with clock time would neither be possible nor make sense. But with FEs, it can sometimes be possible to compare experimental with theoretical results.

But measuring time in function evaluations also has some *disadvantages*, namely:

- There is no guaranteed relationship between FEs and real time.
- An algorithm may have hidden complexities which are not "appearing" in the FEs. For instance, an algorithm could necessitate a lengthy pre-processing procedure before sampling even the first point from the search space. This would not be visible in the FE counter, because, well, it is

not an FE. The same holds for the selection step in an Evolutionary Algorithm (realized as sorting in Section 3.4.1.1). Although this is probably a very fast procedure, it will be outside of what we can measure with FEs.

- A big problem is that one function evaluation can have extremely different actual time requirements and algorithmic complexity in different algorithms. For instance, it is known that in a Traveling Salesman Problem [9,79] with n cities, some algorithms can create an evaluate a new candidate solution from an existing one within a *constant* number of steps, i.e., in $\mathcal{O}(1)$, while others need a number of steps growing quadratically with n, i.e., are in $\mathcal{O}(n^2)$ [167]. If an algorithm of the former type can achieve the same quality as an algorithm of the latter type, we could consider it as better even if it would need ten times as many FEs. Hence, FEs are only fair measurements for comparing two algorithms if they take approximately the same time in both of them.
- Time measured in FEs is harder to comprehend in the context of parallelization and distribution of algorithms.

4.2.3 Summary

Both ways of measuring time have advantages and disadvantages. If we are working on a practical application, then we would maybe prefer to evaluate our algorithm implementations based on the clock time they consume. When implementing a solution for scheduling jobs in an actual factory or for routing vehicles in an actual logistics scenario, what matters is the real, actual time that the operator needs to wait for the results. Whether these time measurements are valuable ten years from now or not plays no role. It also does not matter too much how much time our processes would need if executed on a hardware from what we have or if they were re-implemented in a different programming language.

If we are trying to develop a new algorithm in a research scenario, then may counting FEs is slightly more important. Here we aim to make our results comparable in the long term and we very likely need to compare with results published based on FEs. Another important point is that a black-box algorithm (or metaheuristic) usually makes very few assumptions about the actual problem to which it will be applied later. While we tried to solve the JSSP with our algorithms, you probably have seen that we could plug almost arbitrary other search and solution spaces, representation mappings, or objective functions into them. Thus, we often use artificial problems where FEs can be done very quickly as test problems for our algorithms, because then we can do many experiments. Measuring the runtime of algorithms solving artificial problems does not make that much sense, unless we are working on some algorithms that consume an unusual amount of time.

That being said, I personally prefer to **measure both FEs and clock time**. This way, we are on the safe side.

4.3 Performance Indicators

Unfortunately, many optimization problems are computationally hard. If we want to guarantee that we can solve them to optimality, this would often incur an unacceptably long runtime. Assume that an algorithm $\mathcal{A}$ can solve a problem instance in ten million years while algorithm $\mathcal{B}$ only needs one million. In a practical scenario, usually neither is useful nor acceptable and the fact that $\mathcal{B}$ is better than $\mathcal{A}$ would not matter.[1]

As mentioned in Section 1.2.1, heuristic and metaheuristic optimization algorithms offer a trade-off between runtime and solution quality. This means we have two measurable performance dimensions, namely:

1. the *time*, possibly measured in different ways (see Section 4.2), and
2. the *solution quality*, measured in terms of the best objective value achieved.

If we want to break down performance to single-valued performance indicators, this leads us to two possible choices [61,82], which are:

1. the solution quality we can get within a pre-defined time and
2. the time we need to reach a pre-defined solution quality.

We illustrate these two options, which corresponds to define vertical and horizontal cuts through the progress diagrams, respectively, in Figure 4.1.

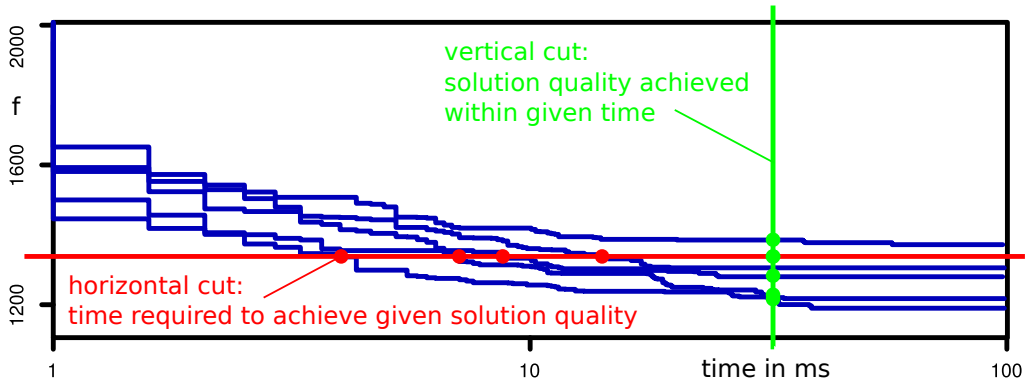

Figure 4.1: Illustration of the two basic forms to measure performance from raw data, based on a fraction of the actual experimental results illustrated in Figure 3.7 and inspired by [61,82].

[1] From a research perspective, it does matter, though.

4.3.1 Vertical Cuts: Best Solution Quality Reached within Given Time

What we did in our simple experiments so far was mainly to focus on the quality that we could achieve within a certain time, i.e., to proceed according to the "vertical cut" scenario. In a practical application, we have a limited computational budget and what counts is the quality of the solutions that we can produce within this budget. The vertical cuts correspond directly to this goal. When creating the final version of an actual implementation of an optimization method, we will have to focus on this measure. Since we then will also have to measure time in clock time, this means that our results will depend on the applied hardware and software configuration as well as on the way we implemented our algorithm, down to the choice of the programming language or even compiler. The advantage of the vertical cut approach is that it can capture all of these issues, as well as performance gains from parallelization or distribution of the algorithms. Our results obtained with vertical cuts will, however not necessarily carry over to other system configurations or problems.

The "vertical cuts" approach is applied in quite a few competitions and research publications, including, for instance, [154].

4.3.2 Horizontal Cuts: Runtime Needed until Reaching a Solution of a Given Quality

The idea horizontal cuts corresponds to defining fixed goal qualities and measuring the runtime needed to get there. For a given problem instance, we would define the target solution quality at which we would consider the problem as solved. This could be a globally optimal quality or a threshold at which the user considers the solution quality as satisfying. This approach is preferred in [61,82] for benchmarking algorithms.

It has the advantage that the number of algorithm steps or seconds needed to solve the problem is a meaningful and interpretable quantity. We can then make statements such as "Algorithm $\mathcal{B}$ is ten times faster than algorithm $\mathcal{A}$ [in solving this problem]." An improvement in the objective value, as we could measure in the vertical cut approach, has no such interpretable meaning, since we do not know whether it is hard or easy to, for instance, squeeze out 10 more time units of makespan in a JSSP instance.

The "horizontal cuts" idea is applied, for instance, in the COCO Framework for benchmarking numerical optimization algorithms [61,82].

One disadvantage of this method is that we cannot guarantee that a run will reach the specified goal quality. Maybe sometimes the algorithm will get trapped in a local optimum before that. This is also visible in Figure 4.1, where one of the runs did not reach the horizontal cut. How to interpret such a situation is harder.[2] In the vertical cut scenario, all runs will always reach the pre-defined

[2] This can be done by assuming that the algorithms would be restarted after consuming certain FEs, but this will be subject to another section (not yet written).

maximum runtimes, as long as we do not artificially abort them earlier, so we always have a full set of measurements.

4.3.3 Determining Goal Values

Regardless of whether we choose vertical or horizontal cuts through the progress diagrams as performance indicators, we will need to define corresponding target values. In some cases, e.g., in a practical application with fixed budgets and/or upper bounds for the acceptable solution quality, we may trivially know them as parts of the specified requirements. In other cases, we may:

- first conduct a set of smaller experiments and get an understand of time requirements or obtainable solution qualities,
- know reasonable defaults from experience,
- set goal objective values based on known lower bounds or even known global optima (e.g., from literature), or
- set them based on what is used in current literature.

Especially in a research setup, the latter is advised. Here, we need to run experiments that produce outputs which are comparable to what we can find in literature, so we need to have the same goal thresholds.

4.3.4 Summary

Despite its major use in research scenarios, the horizontal cut method can also make sense in practical applications. Remember that it is our goal to develop algorithms that can solve the optimization problems within the computational budget, where "solve" again means "reaching a solution of a quality that the user can accept". If we fail to do so, then our software will probably be rejected. If we succeed, then the vertical view would allow us to distinguish algorithms which can *over-achieve* the user requirements. The horizontal view would allow us to distinguish algorithms which can achieve the user requirements *earlier*.

In my opinion, it makes sense to use both indicators. In [167,168,171], for example, we voted for defining a couple of horizontal and vertical cuts to describe the performance of algorithms solving the Traveling Salesman Problem. By using both horizontal and vertical cuts *and* measure runtime both in FEs and milliseconds, we can get a better understanding of the performance and behavior of our algorithms.

Finally, it should be noted that the goal thresholds for horizontal or vertical cuts can directly lead us to defining termination criteria (see Section 2.8).

4.4 Statistical Measures

Most of the optimization algorithms that we have discussed so far are randomized (Section 3.1.3). A randomized algorithm makes at least one random decision which is not a priori known or fixed. Such an algorithm can behave differently every time it is executed.

Definition 30. One independent application of one optimization algorithm to one instance of an optimization problem is called a *run*.

Each *run* is considered as independent and may thus lead to a different result. This also means that the measurements of the basic performance indicators discussed in Section 4.3 can take on different values as well. We may measure k different result solution qualities at the end of k times applications of the same algorithm to the same problem instance (which was also visible in Figure 4.1). In order to get a handy overview about what is going on, we often want to reduce this potentially large amount of information to a few, meaningful and easy-to-interpret values. These values are statistical measures. Of course, this here is neither a book about statistics nor probability, so we can only scratch on the surface of these topics. For better discussions, please refer to text books such as [105,139,149,153,155].

4.4.1 Statistical Samples vs. Probability Distributions

One issues we need to clarify first is that there is a difference between a probability distribution and data sample.

Definition 31. A *probability distribution* F is an assignment of probabilities of occurrence to different possible outcomes in an experiment.

Definition 32. A *random sample* of length $k \geq 1$ is a set of k independent observations of an experiment following a random distribution F.

Definition 33. An *observation* is a measured outcome of an experiment or random variable.

The specification of an optimization algorithm together with its input data, i.e., the problem instance to which it is applied, defines a probability distribution over the possible values a basic performance indicator takes on. If I would possess sufficient mathematical wisdom, I could develop a mathematical formula for the probability of every possible makespan that the 1-swap hill climber `hc_1swap` without restarts could produce on the `swv15` JSSP instance within 100'000 FEs. I could say something like: "With 4% probability, we will find a Gantt chart with a makespan of 2885 time units within 100'000 FEs (by applying `hc_1swap` to `swv15`." With sufficient mathematical skills, I could define such probability distributions for all algorithms. Then, I would know absolutely which algorithm will be the best for which problem.

However, I do not possess such skill and, so far, nobody seems to possess. Despite significant advances in modeling and deriving statistical properties of algorithms for various optimization problems, we are not yet at a point where we can get deep and complete information for most of the relevant problems and algorithms.

We cannot obtain the actual probability distributions describing the results. We can, however, try to *estimate* their parameters by running experiments and measuring results, i.e., by sampling the results.

Table 4.1: The results of one possible outcome of an experiment with several simulated dice throws. The number *# throws* and the thrown *number* are given in the first two columns, whereas the relative frequency of occurrence of number i is given in the columns f_i.

# throws	number	f_1	f_2	f_3	f_4	f_5	f_6
1	5	0.0000	0.0000	0.0000	0.0000	1.0000	0.0000
2	4	0.0000	0.0000	0.0000	0.5000	0.5000	0.0000
3	1	0.3333	0.0000	0.0000	0.3333	0.3333	0.0000
4	4	0.2500	0.0000	0.0000	0.5000	0.2500	0.0000
5	3	0.2000	0.0000	0.2000	0.4000	0.2000	0.0000
6	3	0.1667	0.0000	0.3333	0.3333	0.1667	0.0000
7	2	0.1429	0.1429	0.2857	0.2857	0.1429	0.0000
8	1	0.2500	0.1250	0.2500	0.2500	0.1250	0.0000
9	4	0.2222	0.1111	0.2222	0.3333	0.1111	0.0000
10	2	0.2000	0.2000	0.2000	0.3000	0.1000	0.0000
11	6	0.1818	0.1818	0.1818	0.2727	0.0909	0.0909
12	3	0.1667	0.1667	0.2500	0.2500	0.0833	0.0833
100	…	0.1900	0.2100	0.1500	0.1600	0.1200	0.1700
1’000	…	0.1700	0.1670	0.1620	0.1670	0.1570	0.1770
10’000	…	0.1682	0.1699	0.1680	0.1661	0.1655	0.1623
100’000	…	0.1671	0.1649	0.1664	0.1676	0.1668	0.1672
1’000’000	…	0.1673	0.1663	0.1662	0.1673	0.1666	0.1664
10’000’000	…	0.1667	0.1667	0.1666	0.1668	0.1667	0.1665
100’000’000	…	0.1667	0.1666	0.1666	0.1667	0.1667	0.1667

# throws	number	f_1	f_2	f_3	f_4	f_5	f_6
1'000'000'000	...	0.1667	0.1667	0.1667	0.1667	0.1667	0.1667

Think about throwing an ideal dice. Each number from one to six has the same probability to occur, i.e., the probability $\frac{1}{6} = 0.166\overline{6}$. If we throw a dice a single time, we will get one number. If we throw it twice, we see two numbers. Let f_i be the relative frequency of each number in $k =$ # throws of the dice, i.e., $f_i = \frac{\text{number of times we got } i}{k}$. The more often we throw the dice, the more similar should f_i get to $\frac{1}{6}$, as illustrated in Table 4.1 for a simulated experiments with of many dice throws.

As can be seen in Table 4.1, the first ten or so dice throws tell us very little about the actual probability of each result. However, when we throw the dice many times, the observed relative frequencies become more similar to what we expect. This is called the Law of Large Numbers – and it holds for the application of optimization algorithms too.

There are two take-away messages from this section:

1. It is *never* enough to just apply an optimization algorithm once or twice to a problem instance to get a good impression of a performance indicator. It is a good rule of thumb to always perform at least 20 independent runs. In our experiments on the JSSP, for instance, we did 101 runs per problem instance.
2. We can *estimate* the performance indicators of our algorithms or their implementations via experiments, but we do not know their true value.

4.4.2 Averages: Arithmetic Mean vs. Median

Assume that we have obtained a sample $A = (a_0, a_1, \ldots, a_{n-1})$ of n observations from an experiment, e.g., we have measured the quality of the best discovered solutions of 101 independent runs of an optimization algorithm. We usually want to get reduce this set of numbers to a single value which can give us an impression of what the "average outcome" (or result quality is). Two of the most common options for doing so, for estimating the "center" of a distribution, are to either compute the *arithmetic mean* or the *median*.

4.4.2.1 Mean and Median

Definition 34. The arithmetic mean $\mathrm{mean}(A)$ is an estimate of the expected value of a data sample $A = (a_0, a_1, \ldots, a_{n-1})$. It is computed as the sum of all n elements a_i in the sample data A divided by the total number n of values.

$$\text{mean}(A) = \frac{1}{n}\sum_{i=0}^{n-1} a_i$$

Definition 35. The median $\text{med}(A)$ is the value separating the bigger half from the lower half of a data sample or distribution. It is the value right in the middle of a *sorted* data sample $A = (a_0, a_1, \ldots, a_{n-1})$ where $a_{i-1} \leq a_i \; \forall i \in 1 \ldots (n-1)$.

$$\text{med}(A) = \begin{cases} a_{\frac{n-1}{2}} & \text{if } n \text{ is odd} \\ \frac{1}{2}\left(a_{\frac{n}{2}-1} + a_{\frac{n}{2}}\right) & \text{otherwise} \end{cases} \quad \text{if } a_{i-1} \leq a_i \; \forall i \in 1 \ldots (n-1) \tag{4.1}$$

Notice the zero-based indices in our formula, i.e., the data samples A start with a_0. Of course, any data sample can be transformed to a sorted data sample fulfilling the above constraints by, well, sorting it.

4.4.2.2 Outliers

In order to understand the difference between these two average measures, let us consider two example data sets A and B, both with $n_A = n_B = 19$ values, only differing in their largest observation:

- $A = (1, 3, 4, 4, 4, 5, 6, 6, 6, 6, 7, 7, 9, 9, 9, 10, 11, 12, 14)$
- $B = (1, 3, 4, 4, 4, 5, 6, 6, 6, 6, 7, 7, 9, 9, 9, 10, 11, 12, 10008)$

We find that:

- $\text{mean}(A) = \frac{1}{19}\sum_{i=0}^{18} a_i = \frac{133}{19} = 7$ and
- $\text{mean}(B) = \frac{1}{19}\sum_{i=0}^{18} b_i = \frac{10127}{19} = 553$, while
- $\text{med}(A) = a_9 = 6$ and
- $\text{med}(B) = b_9 = 6$.

The value $b_{18} = 10008$ is an unusual value in B. It is about three orders of magnitude larger than all other measurements. Its appearance has led to a complete change in the average computed based on the arithmetic mean in comparison to dataset A, while it had no impact on the median.

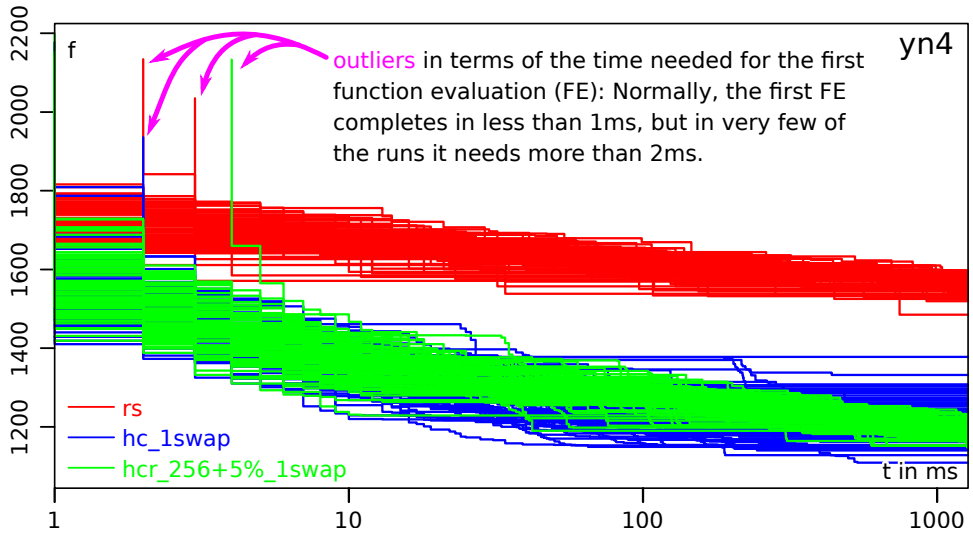

Figure 4.2: Illustrative example for outliers in our JSSP experiment: sometimes the first function evaluation takes unusually long, although this did not have an impact on the end result.

We often call such odd values outliers [78,114]. They may be important, real data, e.g., represent some unusual side-effect in a clinical trial of a new medicine. However, they also often represent measurement errors or observations which have been been disturbed by unusual effects. In our experiments on the JSSP, for instance, a run with surprisingly bad performance may occur when, for whatever reason, the operating system was busy with other things (e.g., updating itself) during the run and thus took away much of the 3 minute computation budget. Figure 4.2 illustrates that this situation may be possible in our JSSP experiment. On rare occasions, the time needed for creating and evaluating the first candidate solution was much longer than usual. This may have been caused by some management procedures inside the Java Virtual Machine executing our experiments. It did not have an impact on the final result, but if we would have computed something like the "mean time until the first solution is constructed," it might give us a wrong impression. Usually, we prefer statistical measures which do not suffer too much from anomalies in the data.

4.4.2.3 Skewed Distributions

The arithmetic mean has another inherent "vulnerability." When thinking about the mean of a data set, we often implicitly assume that the distribution is symmetric. For example, in [137] we find that the annual average income of all families in US grew by 1.2% per year from 1976 to 2007. This mean growth, however, is not distributed evenly, as the top-1% of income recipients had a 4.4% per-year growth while the bottom 99% could only improve by 0.6% per year. The arithmetic mean does not necessarily give an indicator of the range of the most likely observations to encounter.

In optimization, the quality of good results is limited by the lower bound of the objective function and most reasonable algorithms will give us solutions not too far from it. In such a case, the objective function appears almost "unbounded" towards worse solutions, because only the upper bound will be very far away. This means that we may likely encounter algorithms that often give us very good results (close to the lower bound) but rarely also bad results, which can be far from the bound. Thus, the result distribution might be skewed, too.

4.4.2.4 Summary

Take-away message: It makes sense to prefer the median over the mean, because:

- The median it is a more robust against outliers than the arithmetic mean.
- The arithmetic mean is useful especially for symmetric distributions while it does not really represent an intuitive average for skewed distributions while the median is, per definition, suitable for both kinds of distributions.
- Median values are either actually measured outcomes (if we have an odd number of observations) or are usually very close to such (if we have an even number of observations), while arithmetic means may not be similar to any measurement.

The later point is obvious in our example above: $\mathrm{mean}(B) = 533$ is far away from any of the actual samples in B. By the way: We did 101 runs of our optimization algorithms in each of our JSSP experiments instead of one so that there would be an odd number of observations. I thus could always pick a candidate solution of median quality for illustration purposes. There is no guarantee whatsoever that a solution of mean quality exists in an experiment.

It should be noted that it is very common in literature to report arithmetic means of results. While I personally think we should emphasize reporting medians over means, I suggest to report both to be on the safe side – as we did in our JSSP experiments.

4.4.3 Spread: Standard Deviation vs. Quantiles

The average gives us a good impression about the central value or location of a distribution. It does not tell us much about the range of the data. We do not know whether the data we have measured is very similar to the median or whether it may differ very much from the mean. For this, we can compute a measure of dispersion, i.e., a value that tells us whether the observations are stretched and spread far or squeezed tight around the center.

4.4.3.1 Variance, Standard Deviation, and Quantiles

Definition 36. The variance is the expectation of the squared deviation of a random variable from its mean. The variance $\mathrm{var}(A)$ of a data sample $A = (a_0, a_1, \ldots, a_{n-1})$ with n observations can be estimated as:

$$\mathrm{var}(A) = \frac{1}{n-1}\sum_{i=0}^{n-1}(a_i - \mathrm{mean}(A))^2$$

Definition 37. The statistical estimate $\mathrm{sd}(A)$ of the standard deviation of a data sample $A = (a_0, a_1, \ldots, a_{n-1})$ with n observations is the square root of the estimated variance $\mathrm{var}(A)$.

$$\mathrm{sd}(A) = \sqrt{\mathrm{var}(A)}$$

Bigger standard deviations mean that the data tends to be spread farther from the mean. Smaller standard deviations mean that the data tends to be similar to the mean.

Small standard deviations of the result quality and runtimes are good features of optimization algorithms, as they indicate reliable performance. A big standard deviation of the result quality may be exploited by restarting the algorithm, if the algorithms converge early enough so sufficient computational budget is left over to run them a couple of times. We made use of this in Section 3.3.3 when developing the hill climber with restarts. Big standard deviations of the result quality together with long runtimes are bad, as they mean that the algorithms perform unreliable.

A problem with using standard deviations as measure of dispersion becomes visible when we notice that they are derived from and thus depend on the arithmetic mean. We already found that the mean is not a robust statistic and the median should be prefered over it whenever possible. Hence, we would like to see robust measures of dispersion as well.

Definition 38. The q-quantiles are the cut points that divide a *sorted* data sample $A = (a_0, a_1, \ldots, a_{n-1})$ where $a_{i-1} \leq a_i \; \forall i \in 1 \ldots (n-1)$ into q-equally sized parts.

$\mathrm{quantile}_q^k$ be the k^{th} q-quantile, with $k \in 1 \ldots (q-n)$, i.e., there are $q-1$ of the q-quantiles. The probability $P\left[z < \mathrm{quantile}_q^k\right]$ to make an observation z which is smaller than the k^{th} q-quantile should be less or equal than k/q. The probability to encounter a sample which is less or equal to the quantile should be greater or equal to k/q:

$$P\left[z < \mathrm{quantile}_q^k\right] \leq \frac{k}{q} \leq P\left[z \leq \mathrm{quantile}_q^k\right]$$

Quantiles are a generalization of the concept of the median, in that $\mathrm{quantile}_2^1 = \mathrm{med} = \mathrm{quantile}_{2i}^i \forall i > 0$. There are actually several approaches to estimate quantiles from data. The R programming language

widely used in statistics applies Equation (4.2) as default [17,95]. In an ideally-sized data sample, the number of elements minus 1, i.e., $n-1$, would be a multiple of q. In this case, the k^{th} cut point would directly be located at index $h = (n-1)\frac{k}{q}$. Both in Equation (4.2) and in the formula for the median Equation (4.1), this is included the first of the two alternative options. Otherwise, both Equation (4.1) and Equation (4.2) interpolate linearly between the elements at the two closest indices, namely $\lfloor h \rfloor$ and $\lfloor h \rfloor + 1$.

$$\begin{aligned} h &= (n-1)\frac{k}{q} \\ \text{quantile}_q^k(A) &= \begin{cases} a_h & \text{if } h \text{ is integer} \\ a_{\lfloor h \rfloor} + (h - \lfloor h \rfloor) * \left(a_{\lfloor h \rfloor + 1} - a_{\lfloor h \rfloor}\right) & \text{otherwise} \end{cases} \end{aligned} \tag{4.2}$$

Quantiles are more robust against skewed distributions and outliers.

If we do not assume that the data sample is distributed symmetrically, it makes sense to describe the spreads both left and right from the median. A good impression can be obtained by using quantile_4^1 and quantile_4^3, which are usually called the first and third quartile (while $\text{med} = \text{quantile}_4^2$).

4.4.3.2 Outliers

Let us look again at our previous example with the two data samples

- $A = (1,3,4,4,4,5,6,6,6,6,7,7,9,9,9,10,11,12,14)$
- $B = (1,3,4,4,4,5,6,6,6,6,7,7,9,9,9,10,11,12,10008)$

We find that:

- $\text{var}(A) = \frac{1}{19-1}\sum_{i=0}^{n-1}(a_i - 7)^2 = \frac{198}{18} = 11$ and
- $\text{var}(B) = \frac{1}{19-1}\sum_{i=0}^{n-1}(b_i - 533)^2 = \frac{94763306}{18} \approx 5264628.1$, meaning
- $\text{sd}(A) = \sqrt{\text{var}(A)} \approx 3.317$ and
- $\text{sd}(B) = \sqrt{\text{var}(B)} \approx 2294.5$, while on the other hand
- $\text{quantile}_4^1(A) = \text{quantile}_4^1(B) = 4.5$ and
- $\text{quantile}_4^3(A) = \text{quantile}_4^3(B) = 9$.

4.4.3.3 Summary

There again two take-away messages from this section:

1. An average measure without a measure of dispersion does not give us much information, as we do not know whether we can rely on getting results similar to the average or not.

2. We can use quantiles to get a good understanding of the range of observations which is most likely to occur, as quantiles are more robust than standard deviations.

Many research works report standard deviations, though, so it makes sense to also report them – especially since there are probably more people who know what a standard deviation than who know the meaning of quantiles.

Nevertheless, there is one important issue: I often see reports of ranges in the form of $[\mathrm{mean}-\mathrm{sd}, \mathrm{mean}+\mathrm{sd}]$. Handle these with *extreme* caution. In particular, before writing such ranges anywhere, it should be verified first whether the observations actually contain values less than or equal to $\mathrm{mean}-\mathrm{sd}$ and greater than or equal to $\mathrm{mean}+\mathrm{sd}$. If we have a good optimization method which often finds globally optimal solutions, then distribution of discovered solution qualities is probably skewed towards the optimum with a heavy tail towards worse solutions. The mean of the returned objective values minus their standard deviation could be a value smaller than the optimal one, i.e., an invalid, non-existing objective value…

4.5 Testing for Significance

We can now e.g., perform 20 runs each with two different optimization algorithms $\mathcal{A}$ and $\mathcal{B}$ on one problem instance and compute the median of one of the two performance measures for each set of runs. Likely, they will be different. Actually, most the performance indicators in the result tables we looked at in our experiments on the JSSP were different. Almost always, one of the two algorithms will have better results. What does this mean?

It means that one of the two algorithms is better – with a certain probability. We could get the results we get either because $\mathcal{A}$ is really better than $\mathcal{B}$ or – as mentioned in Section 3.4.4 – by pure coincidence, as artifact from the randomness of our algorithms.

If we say "$\mathcal{A}$ is better than $\mathcal{B}$" because this is what we saw in our experiments, we have a certain probabilitys p to be wrong. Strictly speaking, the statement "$\mathcal{A}$ is better than $\mathcal{B}$" makes only sense if we can give an upper bound α for the error probability.

Assume that we compare two data samples $A = (a_0, a_1, \ldots, a_{n_A-1})$ and $B = (b_0, b_1, \ldots, b_{n_B-1})$. We observe that the elements in A tend to be bigger than those in B, for instance, $\mathrm{med}(A) > \mathrm{med}(B)$. Of course, just claiming that the algorithm $\mathcal{A}$ from which the data sample A stems tends to produce bigger results than $\mathcal{B}$ which has given us the observations in B, we would run the risk of being wrong. Instead of doing this directly, we try to compute the probability p that our conclusion is wrong. If p is lower than a small threshold α, say, $\alpha = 0.02$, then we can accept the conclusion. Otherwise, the differences are not significant and we do not make the claim.

4.5.1 Example for the Underlying Idea (Binomial Test)

Let's say I invited you to play a game of coin tossing. We flip a coin. If it shows up as heads, then you win 1 RMB and if it is tails, you give me 1 RMB instead. We play 160 times and I win 128 times, as illustrated in Figure 4.3.

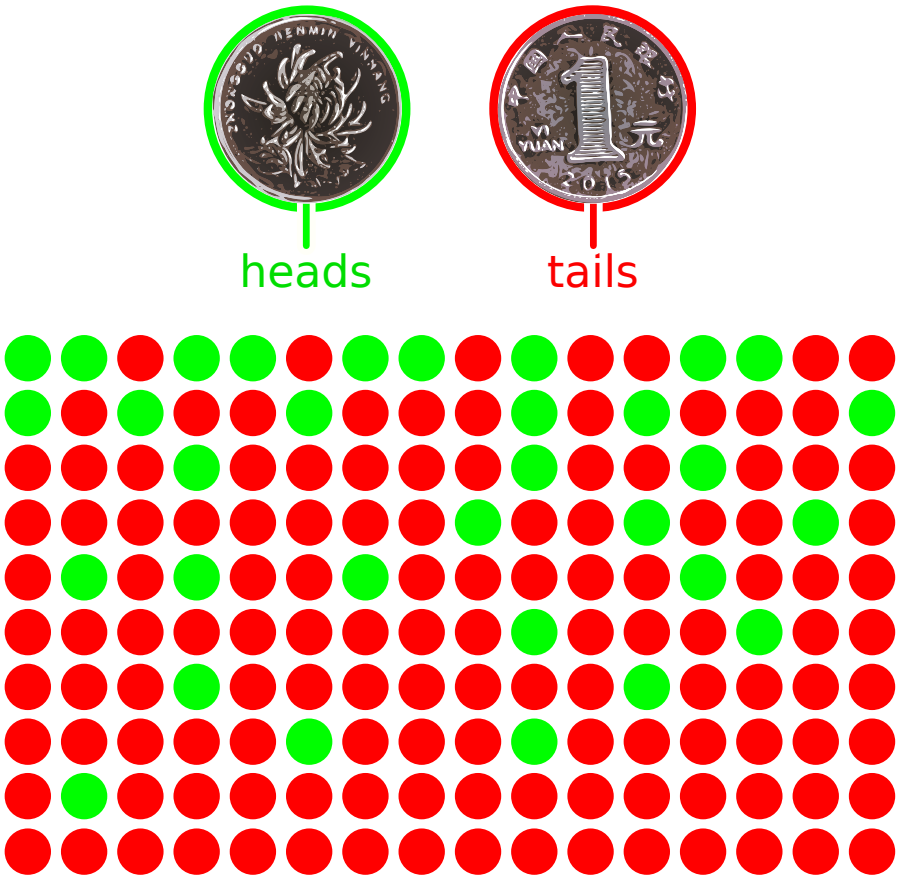

Figure 4.3: The results of our coin tossing game, where I win 128 times (red) and you only 32 times (green).

This situation makes you suspicious, as it seems unlikely to you that I would win four times as often as you with a fair coin. You wonder if I cheated on you, i.e., if used a "fixed" coin with a winning probability different from 0.5. So your hypothesis H_1 is that I cheated. Unfortunately, it is impossible to make any useful statement about my winning probability if I cheated apart from that it should be bigger than 0.5.

What you can do is use make the opposite hypothesis H_0: I did not cheat, the coin is fair and both of us have winning probability $q = 0.5$. Under this assumption you can compute the probability that I

would win at least $m = 128$ times out of $n = 160$ coin tosses. Flipping a coin n times is a Bernoulli process. The probability $P[k|n]$ to win *exactly* k times in n coin tosses is then:

$$P[k|n] = \binom{n}{k} q^k (1-q)^{n-k} = \binom{n}{k} 0.5^k 0.5^{n-k} = \binom{n}{k} 0.5^n = \binom{n}{k} \frac{1}{2^n}$$

where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is the binomial coefficient "n over k". Of course, if winning 128 times would be an indication of cheating, winning even more often would have been, too. Hence we compute the probability $P[k \geq m|n]$ for me to win *at least* m times if we had played with a fair coin, which is:

$$P[k \geq m|n] = \sum_{k=m}^{n} \binom{n}{k} \frac{1}{2^n} = \frac{1}{2^n} \sum_{k=m}^{n} \binom{n}{k}$$

In our case, we get

$$\begin{aligned} P[k \geq 128|160] &= \tfrac{1}{2^{160}} \textstyle\sum_{k=128}^{160} \binom{n}{k} \\ &= \tfrac{1'538'590'628'148'134'280'316'221'828'039'113}{365'375'409'332'725'729'550'921'208'179'070'754'913'983'135'744} \\ &\approx \tfrac{1.539 * 10^{33}}{3.654 * 10^{47}} \\ &\approx 0.00000000000000421098571 \\ &\approx 4.211 * 10^{-15} \end{aligned}$$

In other words, the chance that I would win that often in a fair game is very, very small. If you reject the hypothesis H_0, your probability $p = P[k \geq 128|160]$ to be wrong is, thus, very small as well. If you reject H_0 and accept H_1, p would be your probability to be wrong. Normally, you would set yourself beforehand a limit α, say $\alpha = 0.01$ and if p is less than that, you will risk accusing me. Since $p \ll \alpha$, you therefore can be confident to assume that the coin was fixed. The calculation that we performed here, actually, is called the *binomial test*.

4.5.2 The Concept of Many Statistical Tests

This is, roughly, how statistical tests work. We make a set of observations, for instance, we run experiments with two algorithms $\mathcal{A}$ and $\mathcal{B}$ on one problem instance and get two corresponding lists (A and B) of measurements of a performance indicator. The mean or median values of these lists will probably differ, i.e., one of the two methods will have performed better in average. Then again, it would be very unlikely to, say, apply two randomized algorithms to a problem instance, 100 times each, and get the same results. Matter of fact, it would be very unlikely to apply the same randomized algorithm to a problem instance 100 times and then again for another 100 times and get the same results again.

Still, our hypothesis H_1 could be "Algorithm $\mathcal{A}$ is better than algorithm $\mathcal{B}$." Unfortunately, if that is indeed true, we cannot really know how likely it would have been to get exactly the experimental results that we got. Instead, we define the null hypothesis H_0 that "The performance of the two algorithms is the same," i.e., $\mathcal{A} \equiv \mathcal{B}$. If that would have been the case, the the data samples A and B would stem from the same algorithm, would be observations of the same random variable, i.e., elements from the same population. If we combine A and B to a set O, we can then wonder how likely it would be to draw two sets from O that show the same characteristics as A and B. If the probability is high, then we cannot rule out that $\mathcal{A} \equiv \mathcal{B}$. If the probability is low, say below $\alpha = 0.02$, then we can reject H_0 and confidently assume that H_1 is true and our observation was significant.

4.5.3 Second Example (Randomization Test)

Let us now consider a more concrete example. We want to compare two algorithms $\mathcal{A}$ and $\mathcal{B}$ on a given problem instance. We have conducted a small experiment and measured objective values of their final runs in a few runs in form of the two data sets A and B, respectively:

- $A = (2, 5, 6, 7, 9, 10)$ and
- $B = (1, 3, 4, 8)$

From this, we can estimate the arithmetic means:

- $\mathrm{mean}(A) = \frac{39}{6} = 6.5$ and
- $\mathrm{mean}(B) = \frac{16}{4} = 4$.

It looks like algorithm $\mathcal{B}$ may produce the smaller objective values. But is this assumption justified based on the data we have? Is the difference between $\mathrm{mean}(A)$ and $\mathrm{mean}(B)$ significant at a threshold of $\alpha = 2$?

If $\mathcal{B}$ is truly better than $\mathcal{A}$, which is our hypothesis H_1, then we cannot calculate anything. Let us therefore assume as null hypothesis H_0 the observed difference did just happen by chance and, well, $\mathcal{A} \equiv \mathcal{B}$. Then, this would mean that the data samples A and B stem from the *same* algorithm (as $\mathcal{A} \equiv \mathcal{B}$). The division into the two sets would only be artificial, an artifact of our experimental design. Instead of having two data samples, we only have one, namely the union set O with 10 elements:

- $O = A \cup B = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$

Moreover, any division C of O into two sets A' and B' of sizes 6 and 4, respectively, would have had the same probability of occurrence. Maybe I had first taken all the measurements in A and then those in B afterwards. If I had first taken the measurements in B and then those for A, then I would have gotten $B' = (2, 5, 6, 7)$ and $A' = (9, 10, 1, 3, 4, 8)$. Since I could have taken the measurements in any possible way, if H_0 is true, any division of O into A and B could have happened – and I happened to get one

particular division just by pure chance. If H_0 is true, then the outcome that we observed should not be very unlikely, not very surprising. If the observation that $\mathrm{mean}(A) - \mathrm{mean}(B) \geq 2.5$ would, however, have a very low probability to occur under H_0, then we can probably reject it.

From high school combinatorics, we know that there are $\binom{10}{4} = 210$ different ways of drawing 4 elements from O. Whenever we draw 4 elements from O to form a potential set B'. This leaves the remaining 6 elements for a potential set A', meaning $\binom{10}{6} = 210$ as well. Any of these 210 possible divisions of O would have had the same probability to occur in our experiment – if H_0 holds.

If we enumerate all possible divisions with the small program Listing 4.1, we find that there are exactly 27 of them which lead to a set B' with $\mathrm{mean}(B') \leq 4$. This, of course, means that in exactly these 27 divisions, $\mathrm{mean}(A') \geq 6.5$, because A' contains the numbers which are not in B'.

Listing 4.1 An excerpt of a simple program enumerating all different four-element subsets of O and counting how many have a mean at last as extreme as 6.5. (src)

```
// how often did we find a mean <= 4?
    int meanLowerOrEqualTo4 = 0;
// total number of tested combinations
    int totalCombinations = 0;
// enumerate all sets of four different numbers from 1..10
    for (int i = 10; i > 0; i--) { // as O = numbers from 1 to 10
      for (int j = (i - 1); j > 0; j--) { // we can iterate over
        for (int k = (j - 1); k > 0; k--) { // the sets of size 4
          for (int l = (k - 1); l > 0; l--) { // with 4 loops
            if (((i + j + k + l) / 4.0) <= 4) {
              meanLowerOrEqualTo4++;// yes, found an extreme case
            } // count the extreme case
            totalCombinations++; // add up combos, to verify
          }
        }
      }
    }
// print the result: 27 210
    System.out.println(
        meanLowerOrEqualTo4 + " " + totalCombinations);
```

If H_0 holds, there would have been a probability of $p = \frac{27}{210} = \frac{9}{70} \approx 0.1286$ that we would see arithmetic mean performances *as extreme* as we did. If we would reject H_0 and instead claim that H_1 is true, i.e., algorithm $\mathcal{B}$ is better than $\mathcal{A}$, then we have a 13% chance of being wrong. Since this is more than our pre-defined significance threshold of $\alpha = 0.02$, we cannot reject H_0. Based on the little data we collected, we cannot be sure whether algorithm $\mathcal{B}$ is better or not.

While we cannot reject H_0, this does not mean that it might not be true – actually, the p-value is just 13%. H_0 may or may not be true, and the same holds for H_1. We just do not have enough experimental evidence to reach a conclusion. Thus, we need to be conservative, which here means to not reject H_0 and not accept H_1.

This here just was an example for a Randomization Test [25,57]. It exemplifies how many statistical (non-parametric) tests work.

The number of all possible divisions the joint sets O of measurements grows very quickly with the size of O. In our experiments, where we always conducted 101 runs per experiment, we would already need to enumerate $\binom{202}{101} \approx 3.6 * 10^{59}$ possible divisions when comparing two sets of results. This, of course, is not possible. Hence, practically relevant tests avoid this by applying clever mathematical tricks.

4.5.4 Parametric vs. Non-Parametric Tests

There are two types of tests: parametric and non-parametric tests. The so-called parametric tests assume that the data follows certain distributions. Examples for parametric tests [28] include the t-test, which assumes normal distribution. This means that if our observations follow the normal distribution, then we cannot apply the t-test. Since we often do not know which distribution our results follow, we should not apply the t-test. In general, if we are not 100% sure that our data fulfills the requirements of the tests, we should not apply the tests. Hence, we are on the safe side if we do not use parametric tests.

Non-Parametric tests, on the other hand, are more robust in that make very few assumptions about the distributions behind the data. Examples include

- the Wilcoxon rank sum test with continuity correction (also called Mann-Whitney U test) [14,91,115,149],
- Fisher's Exact Test [63],
- the Sign Test [80,149],
- the Randomization Test [25,57], and
- Wilcoxon's Signed Rank Test [179].

They tend to work similar to the examples given above. When comparing optimization methods, we should always apply non-parametric tests.

The most suitable test in many cases is the above-mentioned **Mann-Whitney U test**. Here, the hypothesis H_1 is that one of the two distributions $\mathcal{A}$ and $\mathcal{B}$ producing the two measured data samples A and B, which are compared by the test, tends to produce larger or smaller values than the other. The null hypothesis H_0 would be that this is not true and it can be rejected if the computed p-values are small. Doing this test manually is quite complicated and describing it is beyond the scope of this book.

Luckily, it is implemented in many tools, e.g., as the function `wilcox.test` in the R programming language, where you can simply feed it with two lists of numbers and it returns the p-value.

Good significance thresholds α are 0.02 or 0.01.

4.5.5 Performing Multiple Tests

We do not just compare two algorithms on a single problem instance. Instead, we may have multiple algorithms and several problem instances. In this case, we need to perform multiple comparisons and thus apply $N > 1$ statistical tests. Before we begin this procedure, we will define a significance threshold α, say 0.01. In each single test, we check one hypothesis, e.g., "this algorithm is better than that one" and estimate a certain probability p to err. If $p < \alpha$, we can accept the hypothesis.

However, with $N > 1$ tests at a significance level α each, our overall probability to accept at least one wrong hypothesis is not α. In *each* of the N test, the probability to err is α and the probability to be right is $1 - \alpha$. The chance to always be right is therefore $(1 - \alpha)^N$ and the chance to accept at least one wrong hypothesis becomes

$$P[\text{error}|\alpha] = 1 - (1 - \alpha)^N$$

For $N = 100$ comparisons and $\alpha = 0.01$ we already arrive at $P[\text{error}|\alpha] \approx 0.63$, i.e., are very likely to accept at least one conclusion. One hundred comparisons is not an unlikely situation: Many benchmark problem sets contain at 100 instances or more. One comparison of two algorithms on each instance means that $N = 100$. Also, we often compare more than two algorithms. For k algorithms on a single problem instance, we would already have $N = k(k-1)/2$ pairwise comparisons.

In all cases with $N > 1$, we therefore need to use an adjusted significance level α' in order to ensure that the overall probability to make wrong conclusions stays below α. The most conservative – and therefore my favorite – way to do so is to apply the Bonferroni correction [55]. It defines:

$$\alpha' = \alpha/N$$

If we use α' as significance level in each of the N tests, we can ensure that the resulting probability to accept at least one wrong hypothesis $P[\text{error}|\alpha'] \leq \alpha$, as illustrated in Figure 4.4.

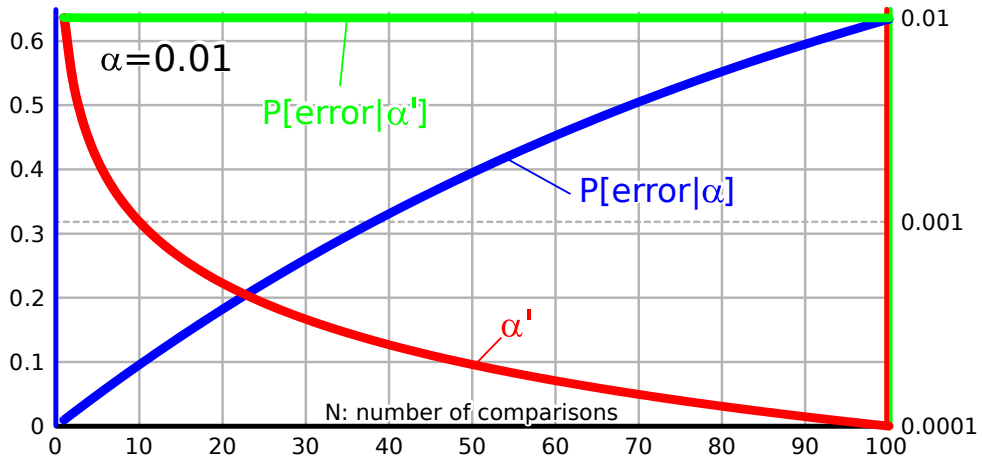

Figure 4.4: The probability $P[\text{error}|\alpha]$ of accepting at least one wrong hypothesis when applying an unchanged significance level α in N tests (left axis) versus similar – and almost constant – $P[\text{error}|\alpha']$ when using corrected value $\alpha' = \alpha/N$ instead (both right axis), for $\alpha = 0.01$.

4.6 Comparing Algorithm Behaviors: Processes over Time

We already discussed that optimization algorithm performance has two dimensions: the required runtime and the solution quality we can get. However, this is not all. Many optimization algorithms are *anytime algorithms*. In Section 3.1.1 and in our experiments we have learned that they attempt to improve their solutions incrementally. The performance of an algorithm on a given problem instance is thus not a single point in the two-dimensional "time vs. quality"-space. It is a curve. We have plotted several diagrams illustrating exactly this, the progress of algorithms over time, in our JSSP experiments in chapter 3. However, in all of our previous discussions, we have ignored this fact and concentrated on computing statistics and comparing "end results."

Is this a problem? In my opinon, yes. In a practical application, like in our example scenario of the JSSP, we have a clear computational budget. If this is exhausted, we have an end result.

However, in research, this is not actually true. If we develop a new algorithm or tackle a new problem in a research setup, we do not necessarily have an industry partner who wants to directly apply our results. This is not the job of research, the job of research is to find new methods and concepts that are promising, from which concrete practical applications may arise later. As researchers, we therefore do often not have a concrete application scenario. We therefore need to find results which should be valid in a wide variety of scenarios defined by the people who later use our research.

This means we do not have a computational budget fixed due to constraints arising from an application. Anytime optimization algorithms, such as metaheuristics, do usually not guarantee that they will find

the global optimum. Often we cannot determine whether the current best solution is a global optimum or not either. This means that such algorithms do not have a "natural" end point – we could let them run forever. Instead, we define termination criteria that we deem reasonable.

4.6.1 Why reporting only end results is bad.

As a result, many publications only provide statistics about the results they have measured at these self-selected termination criteria in form of tables in their papers. When doing so, the imaginary situation illustrated in Figure 4.5 could occur.

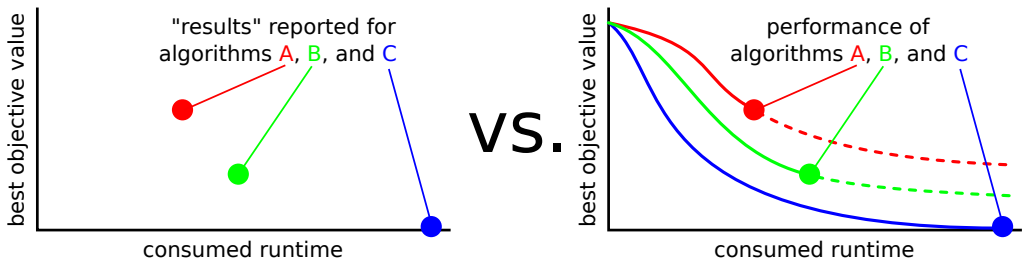

Figure 4.5: "End results" experiments with algorithms versus how the algorithms could actually have performed.

Here, three imaginary researchers have applied three imaginary algorithms to an imaginary problem instance. Independently, they have chosen three different computational budgets and report the median "end results" of their algorithms. From the diagram on the left-hand side, *it looks as if* we have three incomparable algorithms. Algorithm $\mathcal{C}$ needs a long time, but provides the best median result quality. Algorithm $\mathcal{B}$ is faster, but we pay for it by getting worse results. Finally, algorithm $\mathcal{A}$ is the fastest, but has the worst median result quality. We could conclude that, if we would have much time, we would choose algorithm $\mathcal{C}$ while for small computational budgets, algorithm $\mathcal{A}$ looks best.

In reality, the actual course of the optimization algorithms could have looked as illustrated in the diagram on the right-hand side. Here, we find that algorithm $\mathcal{C}$ is always better than algorithm $\mathcal{B}$, which, in turn, is always better than algorithm $\mathcal{A}$. However, we cannot get this information as only the "end results" were reported.

Takeaway-message: Analyzing end results is normally not enough, you need to analyze the whole algorithm behavior [167,171,172].

4.6.2 Progress Plots

We, too, provide tables for the average achieved result qualities in our JSSP examples. However, we always provide diagrams that illustrate the progress of our algorithms over time, too. Visualizations of

the algorithm behavior over runtime can provide us important information.

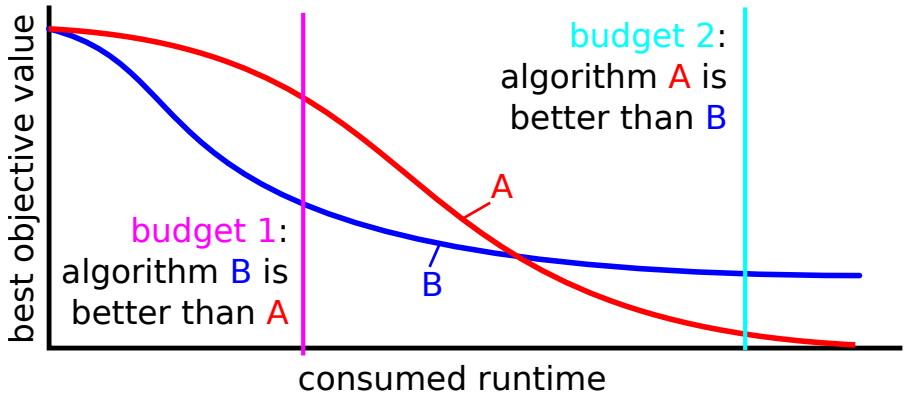

Figure 4.6: Different algorithms may perform best at different points in time.

Figure 4.6, for instance, illustrates a scenario where the best algorithm to choose depends on the available computational budget. Initially, an algorithm $\mathcal{B}$ produces the better median solution quality. Eventually, it is overtaken by another algorithm $\mathcal{A}$, which initially is slower but converges to better results later on. Such a scenario would be invisible if only results for one of the two computational budgets are provided.

Hence, such progress diagrams thus cannot only tell us which algorithms to choose in an actual application scenario later on, where an exact computational budget is defined. During our research, they can also tell us if it makes sense to, e.g., restart our algorithms. If the algorithm does not improve early on but we have time left, a restarting may be helpful – which is what we did for the hill climbing algorithm in Section 3.3.3, for instance.

5 Why is optimization difficult?

So far, we have learned quite a lot of optimization algorithms. These algorithms have different strengths and weaknesses. We have gathered some experience in solving optimization problems. Some optimization problems are hard to solve, some are easy. Actually, sometimes there are *instances* of the same problem that are harder than others. It is natural to ask what makes an optimization problem hard for a given algorithm. It is natural to ask *Why is optimization difficult?* [166,174]

5.1 Premature Convergence

Definition 39. An optimization process has converged if it cannot reach new candidate solutions anymore or if it keeps on producing candidate solutions from a small subset of the solution space $\mathbb{Y}$.

One of the problems in global optimization is that it is often not possible to determine whether the best solution currently known is situated on local or a global optimum and thus, if convergence is acceptable. We often cannot even know if the current best solution is a local optimum or not. In other words, it is usually not clear whether the optimization process can be stopped, whether it should concentrate on refining the current best solution, or whether it should examine other parts of the search space instead. This can, of course, only become cumbersome if there are multiple (local) optima, i.e., the problem is *multi-modal*.

Definition 40. An optimization problem is multi-modal if it has more than one local optimum [47,93,134,145].

The existence of multiple global optima (which, by definition, are also local optima) itself is not problematic and the discovery of only a subset of them can still be considered as successful in many cases. The occurrence of numerous local optima, however, is more complicated, as the phenomenon of *premature convergence* can occur.

5.1.1 The Problem: Convergence to a Local Optimum

Definition 41. Convergence to a local optimum is called *premature convergence* ([166,174], see also Definition 23).

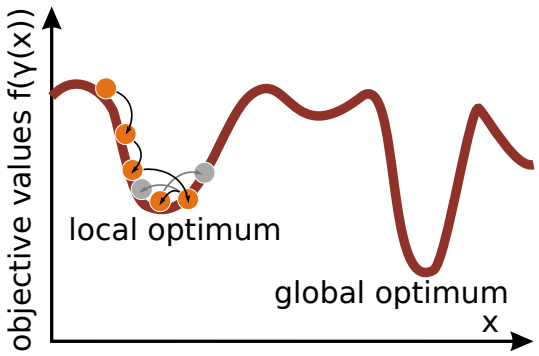

Figure 5.1: An example for how a hill climber from Section 3.3 could get trapped in a local optimum when minimizing over a one-dimensional, real-valued search space.

Figure 5.1 illustrates how a simple hill climber as introduced in Section 3.3 could get trapped in a local optimum. In the example, we assume that we have a sub-range of the real numbers as one-dimensional search space and try to minimize a multi-model objective function. There are more than three optima in the figure, but only one of them is the global minimum. The optimization process, however, discovers the basin of attraction of one of the local optima first.

Definition 42. As *basin of attraction* of a local optimum, we can loosely define the set of points in the search space where applications of the search operator that yield improvements in objective value are likely to guide an optimization process towards the optimum.

Once the hill climber has traced deep enough into this hole, all the new solutions it can produce are higher on the walls around the local optimum and will thus be rejected (illustrated in gray color). The algorithm has prematurely converged.

5.1.2 Countermeasures

What can we do to prevent premature convergence? Actually, we already learned a wide set of techniques! Many of them boil down to balancing exploitation and exploration, as already discovered back in Section 3.4.1.3.

5.1.2.1 Restarts

The first method we learned is to simple restart the algorithm if the optimization process did not improve for a long time, as we did, for instance, with the hill climber in Section 3.3.3. This can help us to exploit the variance in the end solution quality, but whether it can work strongly depends on the number of local optima and the relative size of their basins of attraction. Assume that we have an objective function with s optima and that one of which is the global optimum. Further assume

that the basins of attraction of all optima have the same size and are uniformly distributed over the search space. One would then expect that we need to restart an hill climber about s times in average to discover the global optimum. Unfortunately, there are problems where the number of optima grows exponentially with the dimension of the search space [60], so restarts alone will often not help us to discover the global optimum. This is also what we found in Section 3.3.3: While restarting the hill climber improved its solution quality, we did not discover any globally optimal schedule. Indeed, we did not even prematurely converge to the better local optima.

5.1.2.2 Search Operator Design

To a certain degree we can also combat premature convergence by designing search operators that induce a larger neighborhood. We introduced the nswap operator for our hill climber in Section 3.3.4.1 in such a way that it, most of the time, behaves similar to the original 1swap operator. Sometimes, however, it can make a larger move. A hill climber using this operator will always have a non-zero probability from escaping a local optimum. This would require that the nswap operator makes a step large enough to leave the basin of attraction of the local optimum that it is trapped in *and* that the result of this step is better than the current local optimum. However, nswap also can swap three jobs in the job sequence, which is a relatively small change but still something that 1swap cannot do. This happens much more likely and may help in cases where the optimization process is already at a solution which is locally optimal from the perspective of the 1swap operator but could be improved by, say, swapping three jobs at once. This latter scenario is more likely and larger neighborhoods take longer to be explored, which further decreases the speed of convergence. Nevertheless, a search operator whose neighborhood spans the entire search space could still sometimes help to escape local optima, especially during early stages of the search, where the optimization process did not yet trace down to the bottom of a really good local optimum.

5.1.2.3 Investigating Multiple Points in the Search Space at Once

With the Evolutionary Algorithms in Section 3.4, we attempted yet another approach. The population, i.e., the μ solutions that an $(\mu + \lambda)$ EA preserves, also guard against premature convergence. While a local search might always fall into the same local optimum if it has a large-enough basin of attraction, an EA that preserves a sufficiently large set of diverse points from the search space may find a better solution. If we consider using a population, say in a $(\mu + \lambda)$ EA, we need to think about its size. Clearly, a very small population will render the performance of the EA similar to a hill climber: it will be fast, but might converge to a local optimum. A large population, say big μ and λ values, will increase the chance of eventually finding a better solution. This comes at the cost that every single solution is investigated more slowly: In a $(1 + 1)$-EA, every single function evaluation is spent on improving the

current best solution (as it is a hill climber). In a $(2+1)$-EA, we preserve two solutions and, in average, the neighborhood of each of them is investigated by creating a modified copy only every second FE, and so on. We sacrifice speed for a higher chance of getting better results. Populations mark a trade-off.

5.1.2.4 Diversity Preservation

If we have already chosen to use a population of solutions, as mentioned in the previous section, we can add measures to preserve the diversity of solutions in it. Of course, a population is only useful if it consists of different elements. A population that has collapsed to only include copies of the same point from the search space is not better than performing hill climbing and preserving only that one single current best solution. In other words, only that part of the μ elements of the population is effective that contains different points in the search space. Several techniques have been developed to increase and preserve the diversity in the population, including:

1. Sharing and Niching [44,90,143] are techniques that decrease the fitness of a solution if it is similar to the other solutions in the population. In other words, if solutions are similar, their chance to survive is decreased and different solutions, which are worse from the perspective of the objective function, can remain in the population.
2. Clearing [129,130] takes this idea one step further and only allows the best solution within a certain radius survive.

5.1.2.5 Sometimes Accepting Worse Solutions

Another approach to escape from local optima is to sometimes accept worse solutions. This is a softer approach than performing full restarts. It allows the search to retain some information about the optimization, whereas a "hard" restart discards all knowledge gathered so far. Examples for the idea of sometimes moving towards worse solutions include:

1. When the Simulated Annealing algorithm (Section 3.5) creates a new solution by applying the unary operator to its current point in the search space, it will make the new point current if it is better. If the new point is worse, however, it may still move to this point with a certain probability. This allows the algorithm to escape local optima.
2. Evolutionary Algorithms do not always have to apply the strict truncation selection scheme "$(\mu+\lambda)$ that we introduced in Section 3.4. There exist alternative methods, such as
   a. (μ,λ) population strategies, where the μ current best solutions are always disposed and replaced by the μ best ones the λ newly sampled points in the search space.

b. When the EAs we have discussed so far have to select some solutions from a given population, they always pick those with the best objective value. This is actually not necessary. Actually, there exists a wide variety of different selection methods [23,71] such as Tournament selection [23,26], Ranking Selection [12,26], or the (discouraged! [23,49,176]) fitness-proportionate selection [47,71,90] may also select worse candidate solutions with a certain probability.

5.2 Ruggedness and Weak Causality

All the optimization algorithms we have discussed utilize *memory* in one form or another. The hill climbers remember the best-so-far point in the search space. Evolutionary algorithms even remember a set of multiple such points, called the population. We do this because we expect that the optimization problem exhibits *causality*: Small changes to a candidate solution will lead to small changes in its utility (see Definition 21 in Section 3.3). If this is true, than we are more likely to discover a great solution in the neighborhood of a good solution than in the neighborhood of a solution with bad corresponding objective value. But what if the causality is *weak*?

5.2.1 The Problem: Ruggedness

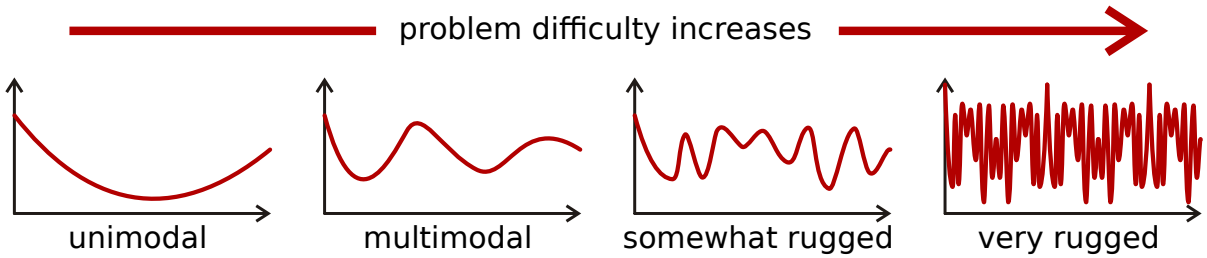

Figure 5.2: An illustration of problems exhibiting increasing ruggedness (from left to right).

Figure 5.2 illustrates different problems with increasing ruggedness of the objective function. Obviously, unimodal problems, which only have a single optimum, are the easiest to solve. Multi-modal problems (Definition 40) are harder, but the difficulty steeply increases if the objective function gets rugged, i.e., rises and falls quickly. Ruggedness has detrimental effects on the performance because it de-values the use of memory in optimization. Under a highly rugged objective function, there is little relationship between the objective values of a given solution and its neighbors. Remembering and investigating the neighborhood of the best-so-far solution will then not be more promising than remembering any other solution or, in the worst case, simply conducting random sampling.

Moderately rugged landscapes already pose a problem, too, because they will have many local optima. Then, techniques like restarting local searches will become less successful, because each restarted search will likely again end up in a local optimum.

5.2.2 Countermeasures

5.2.2.1 Hybridization with Local Search

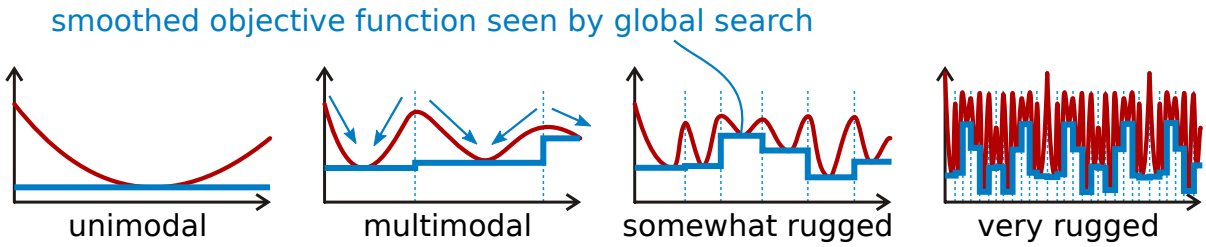

Figure 5.3: An illustration of how the objective functions from Figure 5.2 would look like from the perspective of a Memetic Algorithm: The local search traces down into local optima and the MA hence only "sees" the objective values of optima [178].

It has been suggested that combining global and local search can mitigate the effects of ruggedness to some degree [178]. There are two options for this:

Memetic Algorithms or Lamarckian Evolution (see Section 3.7): Here, the "hosting" global optimization method, say an evolutionary algorithm, samples new points from the search space. It could create them randomly or obtain them as result of a binary search operator. These points are then the starting points of local searches. The result of the local search is then entered into the population. Since the result of a local search is a local optimum, this means that the EA actually only sees the "bottoms" of valleys of the objective functions and never the "peaks". From its perspective, the objective function looks more smoothly.

A similar idea is utilizing the Baldwin Effect [77,89,178]. Here, the global optimization algorithm still works in the search space $\mathbb{X}$ while the local search (in this context also called "learning") is applied in the solution space $\mathbb{Y}$. In other words, the hosting algorithm generates new points $x \in \mathbb{X}$ in the search space and maps them to points $y = \gamma(x)$ in the solution space $\mathbb{Y}$ by applying the representation mapping γ. These points are then refined directly in the solution space, but the refinements are *not* coded back by some reverse mapping. Instead, only their objective values are assigned to the original points in the search space. The algorithm will remember the overall best-ever candidate solution, of course. In our context, the goal here is again to smoothen out the objective function that is seen by the global search method. This "smoothing" is illustrated in Figure 5.3, which is inspired by [178].

5.3 Deceptiveness

Besides causality, another very basic assumption behind metaheuristic optimization is that if candidate solution y_1 is better than y_2, it is more likely that we can find even better solutions in the neighborhood around y_1 than in the neighborhood of y_2. In other words, we assume that following a trail of solutions with improving objective values is in average our best chance of discovering the optimum or, at least, some very good solutions.

5.3.1 The Problem: Deceptiveness

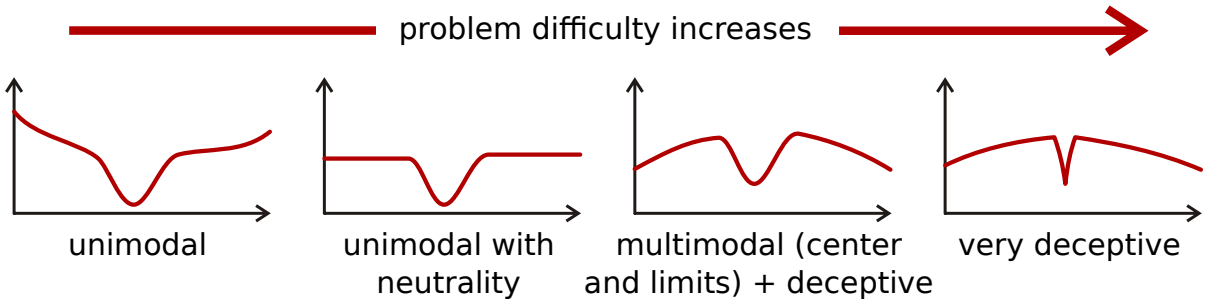

Figure 5.4: An illustration of problems exhibiting increasing deceptiveness (from left to right).

A problem is deceptive if following such a trail of improving solutions leads us away from the actual optimum [166,174]. Figure 5.4 illustrates different problems with increasing deceptiveness of the objective function.

Definition 43. A objective function is *deceptive* (under a given representation and over a subset of the search space) if a hill climber started at any point in this subset will move away from the global optimum.

Definition 43 is an attempt to formalize this concept. We define a specific area $X \subseteq \mathbb{X}$ of the search space $\mathbb{X}$. In this area, we can apply a hill climbing algorithm using a unary search operator $\mathrm{searchOp}$ and a representation mapping $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ to optimize an objective function f. If this objective function f is deceptive on X, then regardless where we start the hill climber, it will move away from the nearest global optimum $x^\star$. "Move away" here means that we also need to have some way to measure the distance between $x^\star$. and another point in the search space and that this distance increases while the hill climber proceeds. OK, maybe not a very handy definition after all – but it describes the phenomena shown in Figure 5.4. The bigger the subset X over which f is deceptive, the harder the problem tends to become for the metaheuristics, as they have an increasing chance of searching into the wrong direction.

5.3.2 Countermeasures

5.3.2.1 Representation Design

From the explanation of the attempted Definition 43 of deceptiveness, we can already see that the design of the search space, representation mapping, and search operators will play a major role in whether a problem is deceptive or not.

5.4 Neutrality and Redundancy

An optimization problem and its representation have the property of causality if small changes in a candidate solution lead to small changes in the objective value. If the resulting changes are large, then causality is weak and the objective function is rugged, which has negative effects on optimization performance. However, if the resulting changes are *zero*, this can have a similar negative impact.

5.4.1 The Problem: Neutrality

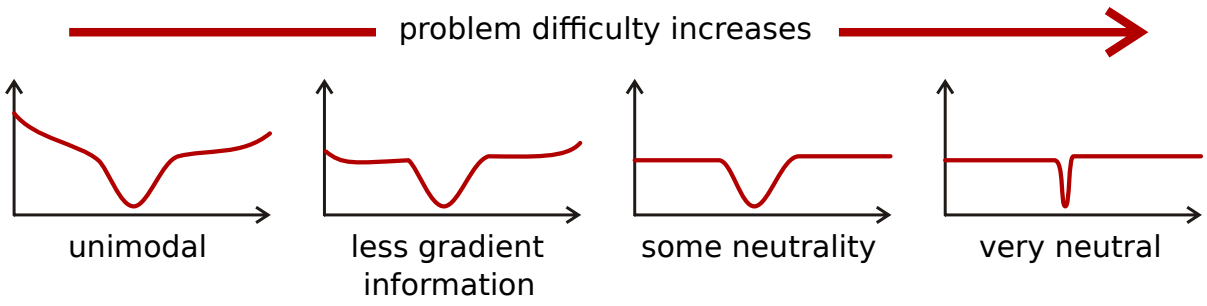

Figure 5.5: An illustration of problems exhibiting increasing neutrality (from left to right).

Neutrality means that a significant fraction of the points in neighborhood of a given point in the search space map to candidate solutions with the same objective value. From the perspective of an optimization process, exploring the neighborhood of a good solution will yield the same solution again and again, i.e., there is no direction into which it can progress in a meaningful way. If half of the candidate solutions have the same objective value, then every second search step cannot lead to an improvement and, for most algorithms, does not yield useful information. This will slow down the search.

Definition 44. The *evolvability* of an optimization process in its current state defines how likely the search operations will lead to candidate solutions with new (and eventually, better) objectives values.

While there are various slightly differing definitions of evolvability both in optimization and evolutionary biology (see [94]), they all condense to the ability to eventually produce better offspring. Researchers in the late 1990s and early 2000s hoped that *adding* neutrality to the representation could increase the evolvability in an optimization process and may hence lead to better performance [13,147,156]. A common idea on how neutrality could be beneficial was the that *neutral networks* would form connections in the search space [13,147].

Definition 45. *Neutral networks* are sets of points in the search space which map to candidate solutions of the same objective value and which are transitively connected by neighborhoods spanned by the unary search operator [147].

The members of a neutral network may have neighborhoods that contain solutions with the same objective value (forming the network), but also solutions with worse and better objective values. An optimization process may drift along a neutral network until eventually discovering a better candidate solution, which then would be in a (better) neutral network of its own. The question then arises how we can introduce such a beneficial form of neutrality into the search space and representation mapping, i.e., how we can create such networks intentionally and controlled. Indeed, it was shown that random neutrality is not beneficial for optimization [106]. Actually, there is no reason why neutral networks should provide a better method for escaping local optima than other methods, such as well-designed search operators (remember Section 3.3.4.1), even if we could create them [106]. Random, uniform, or non-uniform redundancy in the representation are not helpful for optimization [106,138] and should be avoided.

Another idea [156] to achieve self-adaptation in the search is to encode the parameters of search operators in the points in the search space. This means that, e.g., the magnitude to which a unary search operator may modify a certain decision variable is stored in an additional variable which undergoes optimization together with the "actual" variables. Since the search space size increases due to the additional variables, this necessarily leads to some redundancy. (We will discuss this useful concept when I get to writing a chapter on Evolution Strategy, which I will get to eventually, sorry for now.)

5.4.2 Countermeasures

5.4.2.1 Representation Design

From Table 2.3 we know that in our job shop example, the search space is larger than the solution space. Hence, we have some form of redundancy and neutrality. We did not introduce this "additionally," however, but it is an artifact of our representation design with which we pay for a gain in simplicity and avoiding infeasible solutions. Generally, when designing a representation, we should try to construct it as compact and non-redundant as possible. A smaller search space can be searched more efficiently.

5.5 Epistasis: One Root of the Evil

Did you notice that we often said and found that optimization problems get the harder, the more decision variables we have? Why is that? The simple answer is this: Let's say each element $y \in \mathbb{Y}$ from the solution space $\mathbb{Y}$ has n variables, each of which can take on q possible values. Then, there are $|\mathbb{Y}| = q^n$ points in the solution space – in other words, the size of $\mathbb{Y}$ grows exponentially with n. Hence, it takes longer to find the best elements it.

But this is only partially true! It is only true if the variables depend on each other. As a counter example, consider the following problem subject to minimization:

$$f(y) = (y_1 - 3)^2 + (y_2 + 5)^2 + (y_3 - 1)^2, \quad y \in \{-10 \ldots 10\}^3$$

There are three decision variables. However, upon close inspection, we find that they are entirely unrelated. Indeed, we could solve the three *separate* minimization problems given below one-by-one instead, and would obtain the same values for y_1, y_2, and y_3.

$$\begin{aligned} f_1(y_1) &= (y_1 - 3)^2 \qquad y_1 \in -10 \ldots 10 \\ f_2(y_2) &= (y_1 + 5)^2 \qquad y_2 \in -10 \ldots 10 \\ f_3(y_3) &= (y_1 - 1)^2 \qquad y_3 \in -10 \ldots 10 \end{aligned}$$

Both times, the best value for y_1 is 3, for y_2 its -5, and for y_3, it is 1. However, while the three solution spaces of the second set of problems each contain 21 possible values, the solution space of the original problem contains $21^3 = 9261$ values. Obviously, we would prefer to solve the three separate problems, because even in sum, they are much smaller. But in this example, we very lucky: our optimization problem was *separable*, i.e., we could split it into several easier, independent problems.

Definition 46. A function of n variables is *separable* if it can be rewritten as a sum of n functions of just one variable [81,83].

For the JSSP problem that we use as example application domain in this book, this is not the case: Neither can we schedule each jobs separately without considering the other jobs nor can we consider the machines separately. There is also no way in which we could try to find the best time slot for any sub-job without considering the other jobs.

5.5.1 The Problem: Epistasis

The feature that makes optimization problems with more decision variables *much* harder is called *epistasis*.

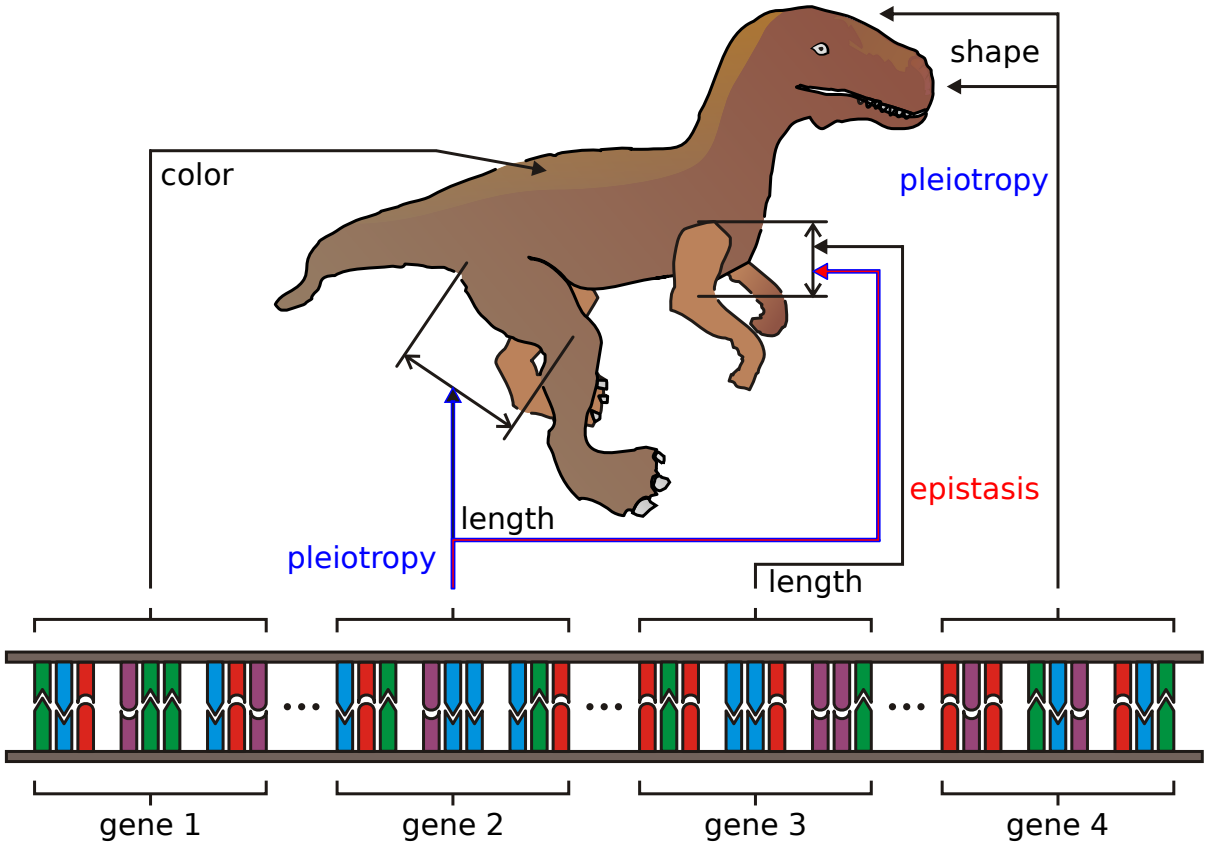

Figure 5.6: An illustration of how genes in biology could exhibit epistatic and pleiotropic interactions in an (entirely fictional) dinosaur.

In biology, epistasis is defined as a form of interaction between different genes [131]. The interaction between genes is epistatic if the effect on the fitness of resulting from altering one gene depends on the allelic state of other genes [113].

Definition 47. In optimization, *epistasis* is the dependency of the contribution of one decision variable to the value of the objective functions on the value of other decision variables [6,45,125,166,174].

A representation has minimal epistasis when every decision variable is independent of every other one. Then, the optimization problem is separable and can be solved by finding the best value for each decision variable separately. A problem is maximally epistatic (or non-separable [83]) when no proper subset of decision variables is independent of any other decision variable [125].

Another related biological phenomenon is *pleiotropy*, which means that a single gene is responsible for multiple phenotypical traits [94]. Like epistasis, pleiotropy can sometimes lead to unexpected improvements but often is harmful. Both effects are sketched in Figure 5.6.

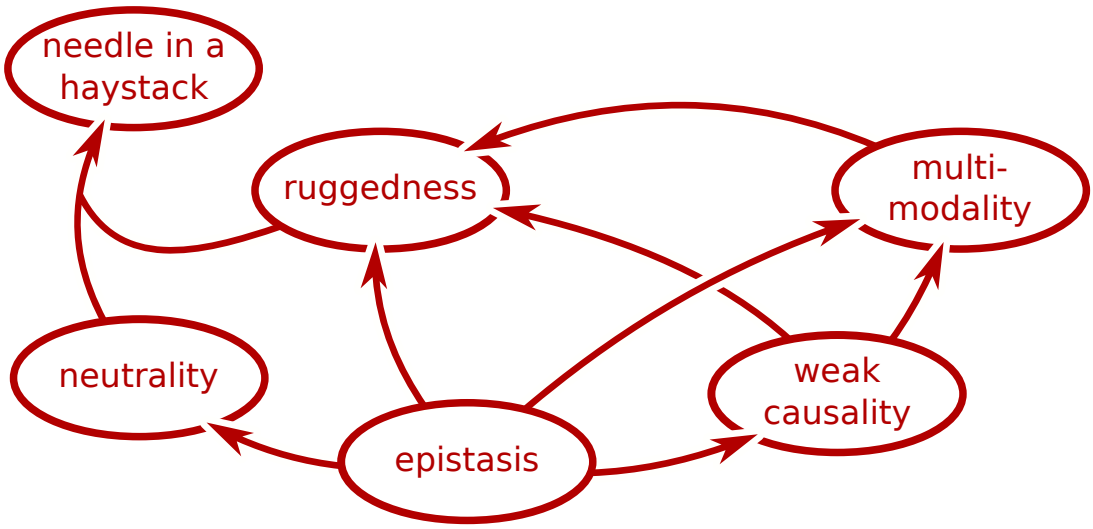

Figure 5.7: How epistasis creates and influences the problematic problem features discussed in the previous sections.

As Figure 5.7 illustrates, epistasis causes or contributes to the problematic traits we have discussed before [166,174]. First, it reduces the causality because changing the value of one decision variable now has an impact on the meaning of other variables. In our representation for the JSSP problem, for instance, changing the order of job IDs at the beginning of an encoded solution can have an impact on the times at which the sub-jobs coming later will be scheduled, even if these themselves were not changed.

If two decision variables interact epistatically, this can introduce local optima, i.e., render the problem multi-modal. The stronger the interaction is, the more rugged the problem becomes. In a maximally-epistatic problem, every decision variable depends on every other one, so applying a small change to one variable can have a large impact.

It is also possible that one decision variable have such semantics that it may turn on or off the impact of another one. Of course, any change applied to a decision variable which has no impact on the objective value then, well, also has no impact, i.e., is *neutral*. Finding rugged, deep valleys in a neutral plane in the objective space corresponds to finding a needle-in-a-haystack, i.e., an ill-defined optimization task.

5.5.2 Countermeasures

Many of the countermeasures for ruggedness, deceptiveness, and neutrality are also valid for epistatic problems. In particular, a good representation design should aim to make the decision variables in the search space as independent as possible

5.5.2.1 Learning the Variable Interactions

Often, a problem may neither be fully-separable nor maximally epistasic. Sometimes, there are groups of decision variables which depend on each others while being independent from other groups. Or, at least, groups of variables which interact strongly and which interact only weakly with variables outside of the group. In such a scenario, it makes sense trying to learn which variables interact during the optimization process. We could then consider each group as a unit, e.g., make sure to pass their values on together when applying a binary operator, or even try to optimize each group separately. Examples for such techniques are:

- linkage learning in EAs [34,72,84,123]
- modeling of variable dependency via statistical models [30,128]
- variable interaction learning [33]

5.6 Scalability

The time required to *solve* a hard problem grows exponentially with the input size, e.g., the number of jobs n or machines m in JSSP. Many optimization problems with practically relevant size cannot be solved to optimality in reasonable time. The purpose of metaheuristics is to deliver a reasonably good solution within a reasonable computational budget. Nevertheless, *any* will take longer for a growing number of decision variables for any (non-trivial) problems. In other words, the *"curse of dimensionality"* [18,19] will also strike metaheuristics.

5.6.1 The Problem: Lack of Scalability

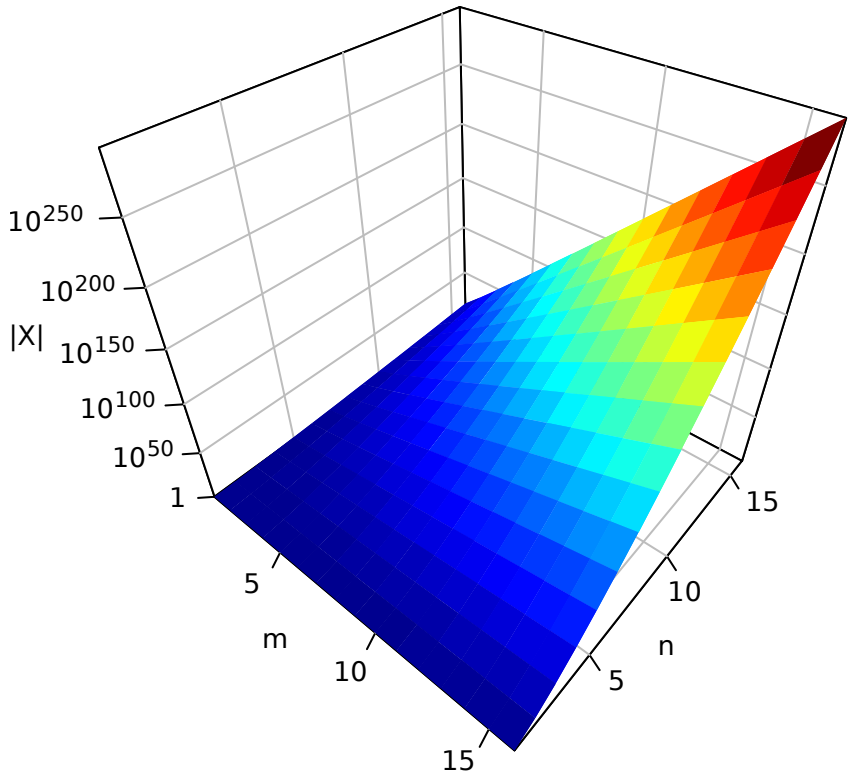

Figure 5.8: The growth of the size of the search space for our representation for the Job Shop Scheduling Problem; compare with Table 2.3.

Figure 5.8 illustrates how the size $|\mathbb{X}|$ of the search space $\mathbb{X}$ grows with the number of machines m and jobs n in our representation for the JSSP. Since the axis for $|\mathbb{X}|$ is logarithmically scaled, it is easy to see that the size grows very fast, exponentially with m and n. This means that most likely, the number of points to be investigated by an algorithm to discover a near-optimal solution also increases quickly with these problem parameters. In other words, if we are trying to schedule the production jobs for a larger factory with more machines and customers, the time needed to find good solutions will increase drastically.

This is also reflected in our experimental results: Simulated Annealing could discover the globally optimal solution for instance `la24` (Section 3.5.4) and in median is only 1.1% off. `la24` is the instance with the smallest search space size. For `abz7`, the second smallest instance, we almost reached the optimum with SA and in median were 3% off, while for the largest instances, the difference was bigger.

5.6.2 Countermeasures

5.6.2.1 Parallelization and Distribution

First, we can try to improve the performance of our algorithms by parallelization and distribution. Parallelization means that we utilize multiple CPUs or CPU cores on the same machine at the same time. Distribution means that we use multiple computers connected by network. Using either approach approach makes sense if we already perform "close to acceptable."

For example, I could try to use the four CPU cores on my laptop to solve a JSSP instance instead of only one. I could, for instance, execute four separate runs of the hill climber of Simulated Annealing in parallel and then just take the best result after the three minutes have elapsed. Matter of fact, I could four different algorithm setups or four different algorithms at once. It makes sense to assume that this would give me a better chance to obtain a good solution. However, it is also clear that, overall, I am still just utilizing the variance of the results. In other words, the result I obtain this way will not really be better than the results I could expect from the best of setups or algorithms if run alone.

One more interesting option is that I could run a metaheuristic together with an exact algorithm which can guarantee to find the optimal solution. For the JSSP, for instance, there exists an efficient dynamic programming algorithm which can solve several well-known benchmark instances within seconds or minutes [75,158,160]. Of course, there can and will be instances that it cannot solve. So the idea would be that in case the exact algorithm can find the optimal solution within the computational budget, we take it. In case it fails, one or multiple metaheuristics running other CPUs may give us a good approximate solution.

Alternatively, I could take a population-based metaheuristic like an Evolutionary Algorithm. Instead of executing ν independent runs on ν CPU cores, I could divide the offspring generation between the different cores. In other words, each core could create, map, and evaluate λ/ν offsprings. Later populations are more likely to find better solutions, but require more computational time to do so. By parallelizing them, I thus could utilize this power without needed to wait longer.

However, there is a limit to the speed-up we can achieve with either parallelization or distribution. Amdahl's Law [7], in particular with the refinements by Kalfa [101] shows that we can get at most a sub-linear speed-up. On the one hand, only a certain fraction of a program can be parallelized and each parallel block has a minimum required execution time (e.g., a block must take at least as long as one single CPU instruction). On the other hand, communication and synchronization between the ν involved threads or processes is required, and the amount of it grows with their number ν. There is a limit value for the number of parallel processes ν above which no further runtime reduction can be achieved. In summary, when battling an exponential growth of the search space size with a sub-linear gain in speed, we will hit certain limits, which may only be surpassed by qualitatively better algorithms.

5.6.2.2 Indirect Representations

In several application areas, we can try to speed up the search by reducing the size of the search space. The idea is to define a small search space $\mathbb{X}$ which is mapped by a representation mapping $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ to a much larger solution space $\mathbb{Y}$, i.e., $|\mathbb{X}| \ll |\mathbb{Y}|$ [20,52].

The first group of indirect representations uses so-called *generative mappings* assume some underlying structure, usually forms of symmetry, in $\mathbb{Y}$ [43,141]. When trying to optimize, e.g., the profile of a tire, it makes sense to assume that it will by symmetrically repeated over the whole tire. Most houses, bridges, trains, car frames, or even plants are symmetric, too. Many physical or chemical processes exhibit symmetries towards the surrounding system or vessel as well. Representing both sides of a symmetric solution separately would be a form of redundancy. If a part of a structure can be repeated, rotated, scaled, or copied to obtain "the whole", then we only need to represent this part. Of course, there might be asymmetric tire profiles or oddly-shaped bridges which could perform even better and which we would then be unable to discover. Yet, the gain in optimization speed may make up for this potential loss.

If there are two decision variables x_1 and x_2 and, usually, $x_2 \approx -x_1$, for example, we could reduce the number of decision variables by one by always setting $x_2 = -x_1$. Of course, we then cannot investigate solutions where $x_2 \neq -x_1$, so we may lose some generality.

Based on these symmetries, indirect representations create a "compressed" version $\mathbb{X}$ of $\mathbb{Y}$ of a much smaller size $|\mathbb{X}| \ll |\mathbb{Y}|$. The search then takes place in this compressed search space and thus only needs to consider much fewer possible solutions. If the assumptions about the structure of the search space is correct, then we will lose only very little solution quality.

A second form of indirect representations is called *ontogenic representation* or *developmental mappings* [52,53,59]. They are similar to generative mapping in that the search space is smaller than the solution space. However, their representational mappings are more complex and often iteratively transform an initial candidate solution with feedback from simulations. Assume that we want to optimize a metal structure composed of hundreds of beams. Instead of encoding the diameter of each beam, we encode a neural network that tells us how the diameter of a beam should be changed based on the stress on it. Then, some initial truss structure is simulated several times. After each simulation, the diameters of the beams are updated according to the neural network, which is fed with the stress computed in the simulation. Here, the search space encodes the weights of the neural network $\mathbb{X}$ while the solution space $\mathbb{Y}$ represents the diameters of the beams. Notice that the size of $\mathbb{X}$ is unrelated to the size of $\mathbb{Y}$, i.e., could be the same for 100 or for 1000 beam structures.

5.6.2.3 Exploiting Separability

Sometimes, some decision variables may be unrelated to each other. If this information can be discovered (see Section 5.5.2.1), the groups of independent decision variables can be optimized separately. This will then be faster.

6 Appendix

It is my goal to make this book easy to read and fast to understand. This goal somehow conflicts with two other goals, namely those of following a clear, abstract, and unified structure as well as being very comprehensive. Unfortunately, we cannot have all at once. Therefore, I choose to sometimes just describe some issues from the surface perspective and dump the details into this appendix.

6.1 Job Shop Scheduling Problem

The Job Shop Scheduling Problem (JSSP) is used as leading example to describe the structure of optimization in chapter 2 and then serves again as application and experiment example when introducing the different metaheuristic algorithms in chapter 3. In order to not divert too much from the most important issues in these sections, we moved the detailed discussions into this appendix.

6.1.1 Lower Bounds

The way to compute the lower bound from Section 2.5.3 for the JSSP is discussed by Taillard in [54]. As said there, the makespan of a JSSP schedule cannot be smaller than the total processing time of the "longest" job. But we also know that the makespan cannot be shorter than the latest "finishing time" F_j of any machine j in the optimal schedule. For a machine j to finish, it will take at least the sum b_j of the runtimes of all the sub-jobs to be executed on it, where

$$b_j = \sum_{i=0}^{n-1} T_{i,j'} \text{ with } M_{i,j'} = j$$

Of course, some sub-jobs j' cannot start right away on the machine, namely if they are not the first sub-job of their job. The minimum idle time of such a sub job is then the sum of the runtimes of the sub-jobs that come before it in the same job i. This means there may be an initial idle period a_j for the machine j, which is at least as big as the shortest possible idle time.

$$a_j \geq \min_{\forall i \in 0 \ldots (n-1)} \left\{ \sum_{j''=0}^{j-1} T_{i,j'} \text{ with } M_{i,j'} = j \right\}$$

Vice versa, there also is a minimum time c_j that the machine will stay idle after finishing all of its sub-jobs.

$$c_j \geq \min_{\forall i \in 0 \ldots (n-1)} \left\{ \sum_{j''=j+1}^{n-1} T_{i,j'} \text{ with } M_{i,j'} = j \right\}$$

With this, we now have all the necessary components of Equation (2.2). We now can put everything together in Listing 6.1.

More information about lower bounds of the JSSP can be found in [10,54,117,157,161,162].

Listing 6.1 Excerpt from the function for computing the lower bound of the makespan of a JSSP instance. (src)

```
// a, b: int[m] filled with MAX_VALUE, T: int[m] filled with 0
    int lowerBound = 0; // overall lower bound

    for (int n = inst.n; (--n) >= 0;) {
      int[] job = inst.jobs[n];

// for each job, first compute the total job runtime
      int jobTimeTotal = 0; // total time
      for (int m = 1; m < job.length; m += 2) {
        jobTimeTotal += job[m];
      }
// lower bound of the makespan must be >= total job time
      lowerBound = Math.max(lowerBound, jobTimeTotal);

      // now compute machine values
      int jobTimeSoFar = 0;
      for (int m = 0; m < job.length;) {
        int machine = job[m++];

// if the sub-job for machine m starts at jobTimeSoFar, the
// smallest machine start idle time cannot be bigger than that
        a[machine] = Math.min(a[machine], jobTimeSoFar);

        int time = job[m++];
// add the sub-job execution time to the machine total time
        T[machine] += time;

        jobTimeSoFar += time;
// compute the remaining time of the job and check if this is
// less than the smallest-so-far machine end idle time
        b[machine] =
            Math.min(b[machine], jobTimeTotal - jobTimeSoFar);
      }
    }

// For each machine, we now know the smallest possible initial
// idle time and the smallest possible end idle time and the
// total execution time. The lower bound of the makespan cannot
// be less than their sum.
    for (int m = inst.m; (--m) >= 0;) {
      lowerBound = Math.max(lowerBound, a[m] + T[m] + b[m]);
    }
```

6.1.2 Probabilities for the `1swap` Operator

Every point in the search space contains $m * n$ integer values. If we swap two of them, we have $m * n * m * (n-1) = m^2n^2 - n$ choices for the indices, half of which would be redundant (like swapping the jobs at index $(10, 5)$ and $(5, 10)$). In total, this yields $T = 0.5 * m^2 * n * (n-1)$ possible different outcomes for a given point from the search space, and our `1swap` operator produces each of them with the same probability.

If $0 < k \leq T$ of outcomes would be an improvement, then the number A of times we need to apply the operator to obtain one of these improvements would follow a geometric distribution and have expected value $\mathbb{E}A$:

$$\mathbb{E}A = \frac{1}{\frac{k}{T}} = \frac{T}{k}$$

We could instead enumerate all possible outcomes and stop as soon as we arrive at an improving move. Again assume that we have k improving moves within the set of T possible outcomes. Let B be the number of steps we need to perform until we haven an improvement. B follows the negative hypergeometric distribution, with "successes" and "failures" swapped, with one trial added (for drawing the improving move). The expected value $\mathbb{E}B$ becomes:

$$\mathbb{E}B = 1 + \frac{(T-k)}{T-(T-k)+1} = 1 + \frac{T-k}{k+1} = \frac{T-k+k+1}{k+1} = \frac{T+1}{k+1}$$

It holds that $\mathbb{E}B \leq \mathbb{E}A$ since $\frac{T}{k} - \frac{T+1}{k+1} = \frac{T(k+1)-(T+1)k}{k(k+1)} = \frac{Tk+T-Tk-k}{k(k+1)} = \frac{T-k}{k(k+1)}$ is positive or zero. This makes sense, as no point would be produced twice during an exhaustive enumeration, whereas random sampling might sample some points multiple times.

This means that enumerating all possible outcomes of the `1swap` operator should also normally yield an improving move faster than randomly sampling them!

Bibliography

[1] Scott Aaronson. 2008. The limits of quantum computers. *Scientific American* 298, 3 (2008), 62–69. DOI:https://doi.org/10.1038/scientificamerican0308-62

[2] Tamer F. Abdelmaguid. 2010. Representations in genetic algorithm for the job shop scheduling problem: A computational study. *Journal of Software Engineering and Applications (JSEA)* 3, 12 (2010), 1155–1162. DOI:https://doi.org/10.4236/jsea.2010.312135

[3] Joseph Adams, Egon Balas, and Daniel Zawack. 1988. The shifting bottleneck procedure for job shop scheduling. *Management Science* 34, 3 (1988), 391–401. DOI:https://doi.org/10.1287/mnsc.34.3.391

[4] Kashif Akram, Khurram Kamal, and Alam Zeb. 2016. Fast simulated annealing hybridized with quenching for solving job shop scheduling problem. *Applied Soft Computing Journal (ASOC)* 49, (2016), 510–523. DOI:https://doi.org/10.1016/j.asoc.2016.08.037

[5] Ali Allahverdi, C. T. Ng, T. C. Edwin Cheng, and Mikhail Y. Kovalyov. 2008. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research (EJOR)* 187, 3 (2008), 985–1032. DOI:https://doi.org/10.1016/j.ejor.2006.06.060

[6] Lee Altenberg. 1997. NK fitness landscapes. In *Handbook of evolutionary computation*, Thomas Bäck, David B. Fogel and Zbigniew Michalewicz (eds.). Oxford University Press, New York, NY, USA. Retrieved from http://dynamics.org/Altenberg/FILES/LeeNKFL.pdf

[7] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large-scale computing capabilities. In *American federation of information processing societies: Proceedings of the spring joint computer conference (AFIPS), April 18–20, 167, Atlantic City, NJ, USA*, New York, NY, USA, 483–485. DOI:https://doi.org/10.1145/1465482.1465560

[8] Mehrdad Amirghasemi and Reza Zamani. 2015. An effective asexual genetic algorithm for solving the job shop scheduling problem. *Computers & Industrial Engineering* 83, (2015), 123–138. DOI:https://doi.org/10.1016/j.cie.2015.02.011

[9] David Lee Applegate, Robert E. Bixby, Vašek Chvátal, and William John Cook. 2007. *The traveling salesman problem: A computational study* (2nd ed.). Princeton University Press, Princeton, NJ, USA.

[10] David Lee Applegate and William John Cook. 1991. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing* 3, 2 (1991), 149–156. DOI:https://doi.org/10.1287/ijoc.3.2.149

[11] Leila Asadzadeh. 2015. A local search genetic algorithm for the job shop scheduling problem with intelligent agents. *Computers & Industrial Engineering* 85, (2015), 376–383. DOI:https://doi.org/10.1016/j.cie.2015.04.006

[12] James E. Baker. 1985. Adaptive selection methods for genetic algorithms. In *Proceedings of the 1st international conference on genetic algorithms and their applications (ICGA'85), June 24–26, 1985, Pittsburgh, PA, USA*, Lawrence Erlbaum Associates, Hillsdale, NJ, USA, 101–111.

[13] Lionel Barnett. 1998. Ruggedness and neutrality – the NKp family of fitness landscapes. In *Artificial life vi: Proceedings of the 6th international conference on the simulation and synthesis of living systems, June 26–29, 1998, Los Angeles, CA, USA* (Complex Adaptive Systems), MIT Press, Cambridge, MA, USA, 18–27. Retrieved from http://users.sussex.ac.uk/~lionelb/downloads/EASy/publications/alife6_paper.pdf

[14] Daniel F. Bauer. 1972. Constructing confidence sets using rank statistics. *Journal of the American Statistical Association (J AM STAT ASSOC)* 67, 339 (1972), 687–690. DOI:https://doi.org/10.1080/01621459.1972.10481279

[15] Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz (Eds.). 1997. *Handbook of evolutionary computation*. Oxford University Press, Inc., New York, NY, USA.

[16] John Edward Beasley. 1990. OR-library: Distributing test problems by electronic mail. *The Journal of the Operational Research Society (JORS)* 41, (1990), 1069–1072. DOI:https://doi.org/10.1057/jors.1990.166

[17] Richard A. Becker, John M. Chambers, and Allan R. Wilks. 1988. *The new S language: A programming environment for data analysis and graphics*. Chapman & Hall, London, UK.

[18] Richard Ernest Bellman. 1957. *Dynamic programming*. Princeton University Press, Princeton, NJ, USA.

[19] Richard Ernest Bellman. 1961. *Adaptive control processes: A guided tour*. Princeton University Press, Princeton, NJ, USA.

[20] Peter John Bentley and Sanjeev Kumar. 1999. Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *Proceedings of the genetic and evolutionary computation conference (GECCO'99), July 13–17, 1999, Orlando, FL, USA*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 35–43.

[21] Christian Bierwirth. 1995. A generalized permutation approach to job shop scheduling with genetic algorithms. *Operations-Research-Spektrum (OR Spectrum)* 17, 2–3 (1995), 87–92. DOI:https://doi.org/10.1007/BF01719250

[22] Christian Bierwirth, Dirk C. Mattfeld, and Herbert Kopfer. 1996. On permutation representations for scheduling problems. In *Proceedings of the 4th international conference on parallel problem solving from*

nature (PPSN IV), September 22–24, 1996, Berlin, Germany (Lecture Notes in Computer Science (LNCS)), Springer-Verlag GmbH, Berlin, Germany, 310–318. DOI:https://doi.org/10.1007/3-540-61723-X_995

[23] Tobias Blickle and Lothar Thiele. 1995. *A comparison of selection schemes used in genetic algorithms* (2nd ed.). Eidgenössische Technische Hochschule (ETH) Zürich, Department of Electrical Engineering, Computer Engineering; Networks Laboratory (TIK), Zürich, Switzerland. Retrieved from ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report11.ps

[24] Mark S. Boddy and Thomas L. Dean. 1989. *Solving time-dependent planning problems*. Brown University, Department of Computer Science, Providence, RI, USA. Retrieved from ftp://ftp.cs.brown.edu/pub/techreports/89/cs89-03.pdf

[25] Jürgen Bortz, Gustav Adolf Lienert, and Klaus Boehnke. 2008. *Verteilungsfreie methoden in der biostatistik* (3rd ed.). Springer Medizin Verlag, Heidelberg, Germany. DOI:https://doi.org/10.1007/978-3-540-74707-9

[26] Anne F. Brindle. 1980. Genetic algorithms for function optimization. PhD thesis. University of Alberta, Edmonton, Alberta, Canada.

[27] Alexander M. Bronstein and Michael M. Bronstein. 2008. Numerical optimization. In *Project TOSCA – tools for non-rigid shape comparison and analysis*. Technion – Israel Institute of Technology, Computer Science Department, Haifa, Israel. Retrieved from http://tosca.cs.technion.ac.il/book/slides/Milano08_optimization.ppt

[28] Shaun Burke. 2001. Missing values, outliers, robust statistics & non-parametric methods. *LC.GC Europe Online Supplement* 1, 2 (2001), 19–24.

[29] Jacek Błażewicz, Wolfgang Domschke, and Erwin Pesch. 1996. The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research (EJOR)* 93, 1 (1996), 1–33. DOI:https://doi.org/10.1016/0377-2217(95)00362-2

[30] Erick Cantú-Paz, Martin Pelikan, and David Edward Goldberg. 2000. Linkage problem, distribution estimation, and bayesian networks. *Evolutionary Computation* 8, 3 (2000), 311–340. DOI:https://doi.org/10.1162/106365600750078808

[31] Uday Kumar Chakraborty, Kalyanmoy Deb, and Mandira Chakraborty. 1996. Analysis of selection algorithms: A markov chain approach. *Evolutionary Computation* 4, 2 (1996), 133–167. DOI:https://doi.org/10.1162/evco.1996.4.2.133

[32] Bo Chen, Chris N. Potts, and Gerhard J. Woeginger. 1998. A review of machine scheduling: Complexity, algorithms and approximability. In *Handbook of combinatorial optimization*, Ding-Zhu Du and Panos M. Pardalos (eds.). Springer-Verlag US, Boston, MA, USA, 1493–1641. DOI:https://doi.org/10.1007/978-1-4613-0303-9_25

[33] Wenxiang Chen, Thomas Weise, Zhenyu Yang, and Ke Tang. 2010. Large-scale global optimization using cooperative coevolution with variable interaction learning. In *Proceedings of the 11th international conference on parallel problem solving from nature, (PPSN'10), part 2, September 11–15, 2010, Kraków, Poland* (Lecture Notes in Computer Science (LNCS)), Springer-Verlag GmbH, Berlin, Germany, 300–309. DOI:https://doi.org/10.1007/978-3-642-15871-1_31

[34] Ying-Ping Chen. 2004. *Extending the scalability of linkage learning genetic algorithms – theory & practice*. Springer-Verlag GmbH, Berlin, Germany. DOI:https://doi.org/10.1007/b102053

[35] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. 1996. A tutorial survey of job-shop scheduling problems using genetic algorithms – I. Representation. *Computers & Industrial Engineering* 30, 4 (1996), 983–997. DOI:https://doi.org/10.1016/0360-8352(96)00047-2

[36] Raymond Chiong, Thomas Weise, and Zbigniew Michalewicz. 2012. *Variants of evolutionary algorithms for real-world applications*. Springer-Verlag, Berlin/Heidelberg. DOI:https://doi.org/10.1007/978-3-642-23424-8

[37] Bastien Chopard and Marco Tomassini. 2018. *An introduction to metaheuristics for optimization*. Springer Nature Switzerland AG, Cham, Switzerland. DOI:https://doi.org/10.1007/978-3-319-93073-2

[38] Philippe Chrétienne, Edward G. Coffman, Jan Karel Lenstra, and Zhen Liu (Eds.). 1995. *Scheduling theory and its applications*. John Wiley & Sons, Chichester, NY, USA.

[39] Stephen Arthur Cook. 1971. 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*, ACM, New York, NY, USA, 151–158. DOI:https://doi.org/10.1145/800157.805047

[40] William John Cook. 2003. Results of concorde for tsplib benchmark. Retrieved from http://www.tsp.gatech.edu/concorde/benchmarks/bench99.html

[41] William John Cook, Daniel G. Espinoza, and Marcos Goycoolea. 2005. *Computing with domino-parity inequalities for the tsp*. Georgia Institute of Technology, Industrial; Systems Engineering, Atlanta, GA, USA. Retrieved from http://www.dii.uchile.cl/~daespino/PApers/DP_paper.pdf

[42] Vladimír Černý. 1985. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications* 45, 1 (1985), 41–51. DOI:https://doi.org/10.1007/BF00940812

[43] David B. D'Ambrosio and Kenneth Owen Stanley. 2007. A novel generative encoding for exploiting neural network sensor and output geometry. In *Proceedings of the 9th genetic and evolutionary computation conference (GECCO'07) July 7–11, 2007, London, England*, ACM, 974–981. DOI:https://doi.org/10.1145/1276958.1277155

[44] Paul J. Darwen and Xin Yao. 1996. Every niching method has its niche: Fitness sharing and implicit sharing compared. In *Proceedings the 4th international conference on parallel problem solving from*

nature PPSN IV, international conference on evolutionary computation, September 22–26, 1996, Berlin, Germany (Lecture Notes in Computer Science (LNCS)), Springer, 398–407. DOI:https://doi.org/10.1007/3-540-61723-X_1004

[45] Yuval Davidor. 1990. Epistasis variance: A viewpoint on GA-hardness. In *Proceedings of the first workshop on foundations of genetic algorithms (FOGA'9), July 15–18, 1990, Bloomington, IN, USA*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 23–35.

[46] Jim Davis, Thomas F. Edgar, James Porter, John Bernaden, and Michael Sarli. 2012. Smart manufacturing, manufacturing intelligence and demand-dynamic performance. *Computers & Chemical Engineering* 47, (2012), 145–156. DOI:https://doi.org/10.1016/j.compchemeng.2012.06.037

[47] Kenneth Alan De Jong. 1975. An analysis of the behavior of a class of genetic adaptive systems. PhD thesis. University of Michigan, Ann Arbor, MI, USA. Retrieved from http://cs.gmu.edu/~eclab/kdj_thesis.html

[48] Kenneth Alan De Jong. 2006. *Evolutionary computation: A unified approach*. MIT Press, Cambridge, MA, USA.

[49] Michael de la Maza and Bruce Tidor. 1993. An analysis of selection procedures with particular attention paid to proportional and bolzmann selection. In *Proceedings of the 5th International Conference on Genetic Algorithms (ICGA'93), July 17–21, 1993, Urbana-Champaign, IL, USA*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 124–131.

[50] Maxence Delorme, Manuel Iori, and Silvano Martello. 2016. Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research (EJOR)* 255, 1 (2016), 1–20. DOI:https://doi.org/10.1016/j.ejor.2016.04.030

[51] Ebru Demirkol, Sanjay V. Mehta, and Reha Uzsoy. 1998. Benchmarks for shop scheduling problems. *European Journal of Operational Research (EJOR)* 109, 1 (1998), 137–141. DOI:https://doi.org/10.1016/S0377-2217(97)00019-2

[52] Alexandre Devert. 2009. When and why development is needed: Generative and developmental systems. In *Proceedings of the genetic and evolutionary computation conference (GECCO'09), July 8–12, 2009, Montreal, Québec, Canada*, ACM, New York, NY, USA, 1843–1844. DOI:https://doi.org/10.1145/1569901.1570194

[53] Alexandre Devert, Thomas Weise, and Ke Tang. 2012. A study on scalable representations for evolutionary optimization of ground structures. *Evolutionary Computation* 20, 3 (2012), 453–472. DOI:https://doi.org/10.1162/EVCO_a_00054

[54] Éric D. Taillard. 1993. Benchmarks for basic scheduling problems. *European Journal of Operational Research (EJOR)* 64, 2 (1993), 278–285. DOI:https://doi.org/10.1016/0377-2217(93)90182-M

[55] Olive Jean Dunn. 1961. Multiple comparisons among means. *Journal of the American Statistical Association (J AM STAT ASSOC)* 56, 293 (1961), 52–64. DOI:https://doi.org/10.1080/01621459.1961.10482090

[56] Harald Dyckhoff and Ute Finke. 1992. *Cutting and packing in production and distribution: A typology and bibliography*. Physica-Verlag, Heidelberg, Germany. DOI:https://doi.org/10.1007/978-3-642-58165-6

[57] Eugene S. Edgington. 1995. *Randomization tests* (3rd ed.). CRC Press, Inc., Boca Raton, FL, USA.

[58] Ágoston E. Eiben and C. A. Schippers. 1998. On evolutionary exploration and exploitation. *Fundamenta Informaticae – Annales Societatis Mathematicae Polonae, Series IV* 35, 1-2 (1998), 35–50. DOI:https://doi.org/10.3233/FI-1998-35123403

[59] Nicolás S. Estévez and Hod Lipson. 2007. Dynamical blueprints: Exploiting levels of system-environment interaction. In *Proceedings of the 9th genetic and evolutionary computation conference (GECCO'07) July 7–11, 2007, London, England*, ACM, 238–244. DOI:https://doi.org/10.1145/1276958.1277009

[60] Steffen Finck, Nikolaus Hansen, Raymond Ros, and Anne Auger. 2009. *Real-parameter black-box optimization benchmarking 2010: Presentation of the noiseless functions*. Institut National de Recherche en Informatique et en Automatique (INRIA). Retrieved from http://coco.gforge.inria.fr/downloads/download16.00/bbobdocfunctions.pdf

[61] Steffen Finck, Nikolaus Hansen, Raymond Ros, and Anne Auger. 2015. COCO documentation, release 15.03. Retrieved from http://coco.lri.fr/COCOdoc/COCO.pdf

[62] Henry Fisher and Gerald L. Thompson. 1963. Probabilistic learning combinations of local job-shop scheduling rules. In *Industrial scheduling*, John F. Muth and Gerald L. Thompson (eds.). Prentice-Hall, Englewood Cliffs, NJ, USA, 225–251.

[63] Sir Ronald Aylmer Fisher. 1922. On the interpretation of χ^2 from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society* 85, (1922), 87–94. Retrieved from http://hdl.handle.net/2440/15173

[64] Sir Ronald Aylmer Fisher and Frank Yates. 1948. *Statistical tables for biological, agricultural and medical research* (3rd ed.). Oliver & Boyd, London, UK.

[65] Michael R. Garey and David S. Johnson. 1979. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman; Company, New York, NY, USA.

[66] Michael R. Garey, David S. Johnson, and Ravi Sethi. 1976. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research (MOR)* 1, 2 (1976), 117–129. DOI:https://doi.org/10.1287/moor.1.2.117

[67] Mitsuo Gen, Yasuhiro Tsujimura, and Erika Kubota. 1994. 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, October 2–5, 1994, San Antonio, TX, USA*, IEEE. DOI:https://doi.org/10.1109/ICSMC.1994.400072

[68] Michel Gendreau and Jean-Yves Potvin (Eds.). 2010. *Handbook of metaheuristics* (2nd ed.). Springer Science+Business Media, LLC, Boston, MA, USA. DOI:https://doi.org/10.1007/978-1-4419-1665-5

[69] Fred Glover and Gary A. Kochenberger (Eds.). 2003. *Handbook of metaheuristics*. Springer Netherlands, Dordrecht, Netherlands. DOI:https://doi.org/10.1007/b101874

[70] David Edward Goldberg. 1989. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[71] David Edward Goldberg and Kalyanmoy Deb. 1990. A comparative analysis of selection schemes used in genetic algorithms. In *Proceedings of the first workshop on foundations of genetic algorithms (FOGA'90), July 15–18, 1990, Bloomington, IN, USA*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 69–93. Retrieved from http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.101.9494

[72] David Edward Goldberg, Kalyanmoy Deb, and Bradley Korb. 1989. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems* 3, 5 (1989), 493–530. Retrieved from http://www.complex-systems.com/pdf/03-5-5.pdf

[73] Ronald Lewis Graham, Eugene Leighton Lawler, Jan Karel Lenstra, and Alexander Hendrik George Rinnooy Kan. 1979. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics* 5, (1979), 287–326. DOI:https://doi.org/10.1016/S0167-5060(08)70356-X

[74] Vincent Granville, Mirko Křivánek, and Jean-Paul Rasson. 1994. Simulated annealing: A proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 16, 6 (1994), 652–656. DOI:https://doi.org/10.1109/34.295910

[75] Joaquim A. S. Gromicho, Jelke Jeroen van Hoorn, Francisco Saldanha-da-Gama, and Gerrit T. Timmer. 2012. Solving the job-shop scheduling problem optimally by dynamic programming. *Computers & Operations Research* 39, 12 (2012), 2968–2977. DOI:https://doi.org/10.1016/j.cor.2012.02.024

[76] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. 1991. Optimal control of plotting and drilling machines: A case study. *Zeitschrift für Operations Research (ZOR) – Methods and Models of Operations Research* 35, 1 (1991), 61–84. DOI:https://doi.org/10.1007/BF01415960

[77] Frédéric Gruau and L. Darrell Whitley. 1993. Adding learning to the cellular development of neural networks: Evolution and the Baldwin effect. *Evolutionary Computation* 1, 3 (1993), 213–233. DOI:https://doi.org/10.1162/evco.1993.1.3.213

[78] Frank E. Grubbs. 1969. Procedures for detecting outlying observations in samples. *Technometrics* 11, 1 (1969), 1–21. DOI:https://doi.org/10.1080/00401706.1969.10490657

[79] Gregory Z. Gutin and Abraham P. Punnen (Eds.). 2002. *The traveling salesman problem and its variations*. Kluwer Academic Publishers, Norwell, MA, USA. DOI:https://doi.org/10.1007/b101971

[80] Lorenz Gygax. 2003. Statistik für Nutztierethologen – Einführung in die statistische Denkweise: Was ist, was macht ein statistischer Test? Retrieved from http://www.proximate-biology.ch/documents/introEtho.pdf

[81] George Hadley. 1964. *Nonlinear and dynamics programming*. Addison-Wesley Professional, Reading, MA, USA.

[82] Nikolaus Hansen, Anne Auger, Steffen Finck, and Raymond Ros. 2010. *Real-parameter black-box optimization benchmarking 2010: Experimental setup*. Institut National de Recherche en Informatique et en Automatique (INRIA). Retrieved from https://hal.inria.fr/inria-00462481

[83] Nikolaus Hansen, Raymond Ros, Nikolas Mauny, Marc Schoenauer, and Anne Auger. 2008. *PSO facing non-separable and ill-conditioned problems*. Institut National de Recherche en Informatique et en Automatique (INRIA). Retrieved from http://hal.archives-ouvertes.fr/docs/00/25/01/60/PDF/RR-6447.pdf

[84] Georges Raif Harik. 1997. Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms. PhD thesis. University of Michigan, Ann Arbor, MI, USA. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.7092

[85] William Eugene Hart, James E. Smith, and Natalio Krasnogor (Eds.). 2005. *Recent advances in memetic algorithms*. Springer, Berlin, Heidelberg. DOI:https://doi.org/10.1007/3-540-32363-5

[86] Keld Helsgaun. 2009. General k-opt submoves for the Lin-Kernighan TSP heuristic. *Mathematical Programming Computation (MPC): A Publication of the Mathematical Optimization Society* 1, 2-3 (2009), 119–163. DOI:https://doi.org/10.1007/s12532-009-0004-6

[87] Mario Hermann, Tobias Pentek, and Boris Otto. 2016. Design principles for industrie 4.0 scenarios. In *Proceedings of the 49th hawaii international conference on system sciences (HICSS), January 5–8, 2016, Koloa, HI, USA*, IEEE Computer Society Press, Los Alamitos, CA, USA, 3928–3937. DOI:https://doi.org/10.1109/HICSS.2016.488

[88] Leonor Hernández-Ramírez, Juan Frausto Solis, Guadalupe Castilla-Valdez, Juan Javier González-Barbosa, David Terán-Villanueva, and María Lucila Morales-Rodríguez. 2019. A hybrid simulated annealing for job shop scheduling problem. *International Journal of Combinatorial Optimization Problems and Informatics (IJCOPI)* 10, 1 (2019), 6–15. Retrieved from http://ijcopi.org/index.php/ojs/article/view/111

[89] Geoffrey Everest Hinton and Steven J. Nowlan. 1987. How learning can guide evolution. *Complex Systems* 1, 3 (1987). Retrieved from https://www.complex-systems.com/abstracts/v01_i03_a06/

[90] John Henry Holland. 1975. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, Ann Arbor, MI, USA.

[91] Myles Hollander and Douglas Alan Wolfe. 1973. *Nonparametric statistical methods*. John Wiley & Sons, New York, USA.

[92] Holger H. Hoos and Thomas Stützle. 2005. *Stochastic local search: Foundations and applications*. Elsevier.

[93] Jeffrey Horn and David Edward Goldberg. 1995. Genetic algorithm difficulty and the modality of the fitness landscape. In *Proceedings of the third workshop on foundations of genetic algorithms (FOGA 3), July 31–August 2, 1994, Estes Park, CO, USA*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 243–269.

[94] Ting Hu. 2010. Evolvability and rate of evolution in evolutionary computation. PhD thesis. Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada. Retrieved from http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/TingHu_thesis.html

[95] Rob J. Hyndman and Yanan Fan. 1996. Sample quantiles in statistical packages. *The American Statistician* 50, 4 (1996), 361–365. DOI:https://doi.org/10.2307/2684934

[96] Dean Jacobs, Jan Prins, Peter Siegel, and Kenneth Wilson. 1982. Monte carlo techniques in code optimization. *ACM SIGMICRO Newsletter* 13, 4 (1982), 143–148.

[97] Klaus Jansen, Monaldo Mastrolilli, and Roberto Solis-Oba. 2005. Approximation schemes for job shop scheduling problems with controllable processing times. *European Journal of Operational Research (EJOR)* 167, 2 (2005), 297–319. DOI:https://doi.org/10.1016/j.ejor.2004.03.025

[98] Tianhua Jiang and Chao Zhang. 2018. Application of grey wolf optimization for solving combinatorial problems: Job shop and flexible job shop scheduling cases. *IEEE Access* 6, (2018), 26231–26240. DOI:https://doi.org/10.1109/ACCESS.2018.2833552

[99] Selmer Martin Johnson. 1954. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly* I, (1954), 61–68. DOI:https://doi.org/10.1002/nav.3800010110

[100] Vedavyasrao Jorapur, V. S. Puranik, A. S. Deshpande, and M. R. Sharma. 2014. Comparative study of different representations in genetic algorithms for job shop scheduling problem. *Journal of Software Engineering and Applications (JSEA)* 7, 7 (2014), 571–580. DOI:https://doi.org/10.4236/jsea.2014.77053

[101] Winfried Kalfa. 1988. *Betriebssysteme*. Akademie-Verlag, Berlin, Germany.

[102] Richard M. Karp. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations. The ibm research symposia series.*, Raymond E. Miller and James W. Thatcher (eds.). Springer, Boston, MA, USA, 85–103. DOI:https://doi.org/10.1007/978-1-4684-2001-2_9

[103] Scott Kirkpatrick, C. Daniel Gelatt, Jr., and Mario P. Vecchi. 1983. Optimization by simulated annealing. *Science Magazine* 220, 4598 (1983), 671–680. DOI:https://doi.org/10.1126/science.220.4598.671

[104] Robert Klein. 2000. *Scheduling of resource-constrained projects*. Springer US, New York, NY, USA. DOI:https://doi.org/10.1007/978-1-4615-4629-0

[105] Achim Klenke. 2014. *Probability theory: A comprehensive course* (2nd ed.). Springer-Verlag, London, UK. DOI:https://doi.org/10.1007/978-1-4471-5361-0

[106] Joshua D. Knowles and Richard A. Watson. 2002. 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), September 7–11, 2002, Granada, Spain*, Springer-Verlag, Berlin, Heidelberg, 88–98. DOI:https://doi.org/10.1007/3-540-45712-7_9

[107] Donald Ervin Knuth. 1969. *Seminumerical algorithms*. Addison–Wesley, Reading, MA, USA.

[108] Eugene Leighton Lawler. 1982. Recent results in the theory of machine scheduling. In *Math programming: The state of the art*, AAchim Bachem, Bernhard Korte and Martin Grötschel (eds.). Springer-Verlag, Bonn/New York, 202–234. DOI:https://doi.org/10.1007/978-3-642-68874-4_9

[109] Eugene Leighton Lawler, Jan Karel Lenstra, Alexander Hendrik George Rinnooy Kan, and David B. Shmoys. 1985. *The traveling salesman problem: A guided tour of combinatorial optimization*. Wiley Interscience, Chichester, West Sussex, UK.

[110] Eugene Leighton Lawler, Jan Karel Lenstra, Alexander Hendrik George Rinnooy Kan, and David B. Shmoys. 1993. Sequencing and scheduling: Algorithms and complexity. In *Handbook of operations research and management science*, Stephen C. Graves, Alexander Hendrik George Rinnooy Kan and Paul H. Zipkin (eds.). North-Holland Scientific Publishers Ltd., Amsterdam, The Netherlands, 445–522. DOI:https://doi.org/10.1016/S0927-0507(05)80189-6

[111] Stephen R. Lawrence. 1984. Resource constrained project scheduling: An experimental investigation of heuristic scheduling techniques (supplement). PhD thesis. Graduate School of Industrial Administration (GSIA), Carnegie-Mellon University; Graduate School of Industrial Administration (GSIA), Carnegie-Mellon University, Pittsburgh, PA, USA.

[112] Andrea Lodi, Silvano Martello, and Michele Monaci. 2002. Two-dimensional packing problems: A survey. *European Journal of Operational Research (EJOR)* 141, 2 (2002), 241–252. DOI:https://doi.org/10.1016/S0377-2217(02)00123-6

[113] Jay L. Lush. 1935. Progeny test and individual performance as indicators of an animal's breeding value. *Journal of Dairy Science (JDS)* 18, 1 (1935), 1–19. DOI:https://doi.org/10.3168/jds.S0022-0302(35)93109-5

[114] Gangadharrao Soundalyarao Maddala. 1992. *Introduction to econometrics* (Second ed.). MacMillan, New York, NY, USA.

[115] Henry B. Mann and Donald R. Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics (AOMS)* 18, 1 (1947), 50–60. DOI:https://doi.org/10.1214/aoms/1177730491

[116] Monaldo Mastrolilli and Ola Svensson. 2011. Hardness of approximating flow and job shop scheduling problems. *Journal of the ACM (JACM)* 58, 5 (2011), 20:1–20:32. DOI:https://doi.org/10.1145/2027216.2027218

[117] Graham McMahon and Michael Florian. 1975. On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research* 23, 3 (1975). DOI:https://doi.org/10.1287/opre.23.3.475

[118] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall Nicholas Rosenbluth, Augusta H. Teller, and Edward Teller. 1953. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics* 21, 6 (1953), 1087–1092. DOI:https://doi.org/10.1063/1.1699114

[119] Zbigniew Michalewicz. 1996. *Genetic algorithms + data structures = evolution programs*. Springer-Verlag GmbH, Berlin, Germany.

[120] Melanie Mitchell. 1998. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, USA.

[121] Melanie Mitchell, Stephanie Forrest, and John Henry Holland. 1991. The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Toward a practice of autonomous systems: Proceedings of the first european conference on artificial life (actes de la première conférence européenne sur la vie artificielle) (ECAL'91), December 11–13, 1991, Paris, France* (Bradford Books), MIT Press, Cambridge, MA, USA, 245–254. Retrieved from http://web.cecs.pdx.edu/~mm/ecal92.pdf

[122] Pablo Moscato. 1989. *On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms*. California Institute of Technology (Caltech), Caltech Concurrent Computation Program (C3P), Pasadena, CA, USA. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.27.9474

[123] Masaharu Munetomo and David Edward Goldberg. 1999. Linkage identification by non-monotonicity detection for overlapping functions. *Evolutionary Computation* 7, 4 (1999), 377–398. DOI:https://doi.org/10.1162/evco.1999.7.4.377

[124] Yuichi Nagata and Shigenobu Kobayashi. 2013. A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem. *INFORMS Journal on Computing* 25, 2 (2013), 346–363. DOI:https://doi.org/10.1287/ijoc.1120.0506

[125] Bart Naudts and Alain Verschoren. 1996. Epistasis on finite and infinite spaces. In *Proceedings of the eigth international conference on systems research, informatics and cybernetics (InterSymp'96),*

August 14–18, 1996, Baden-Baden, Germany, International Institute for Advanced Studies in Systems Research; Cybernetic (IIAS), Tecumseh, ON, Canada, 19–23. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.6455

[126] Ferrante Neri, Carlos Cotta, and Pablo Moscato (Eds.). 2012. *Handbook of memetic algorithms*. Springer, Berlin/Heidelberg. DOI:https://doi.org/10.1007/978-3-642-23247-3

[127] Andreas Nolte and Rainer Schrader. 2000. A note on the finite time behaviour of simulated annealing. *Mathematics of Operations Research (MOR)* 25, 3 (2000), 476–484. DOI:https://doi.org/10.1287/moor.25.3.476.12211

[128] Martin Pelikan, David Edward Goldberg, and Erick Cantú-Paz. 1999. BOA: The bayesian optimization algorithm. In *Proceedings of the genetic and evolutionary computation conference (GECCO'99), July 13–17, 1999, Orlando, FL, USA*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 525–532.

[129] Alan Pétrowski. 1996. A clearing procedure as a niching method for genetic algorithms. In *Proceedings of IEEE international conference on evolutionary computation (CEC'96), May 20–22, 1996, Nagoya, Japan*, IEEE Computer Society Press, Los Alamitos, CA, USA, 798–803. DOI:https://doi.org/10.1109/ICEC.1996.542703

[130] Alan Pétrowski. 1997. *An efficient hierarchical clustering technique for speciation*. Institut National des Télécommunications, Evry Cedex, France.

[131] Patrick C. Phillips. 1998. The language of gene interaction. *Genetics* 149, 3 (1998), 1167–1171. Retrieved from http://www.genetics.org/content/149/3/1167

[132] Martin Pincus. 1970. Letter to the editor – a monte carlo method for the approximate solution of certain types of constrained optimization problems. *Operations Research* 18, 6 (1970), 1225–1228. DOI:https://doi.org/10.1287/opre.18.6.1225

[133] Michael L. Pinedo. 2016. *Scheduling: Theory, algorithms, and systems* (5th ed.). Springer International Publishing. DOI:https://doi.org/10.1007/978-3-319-26580-3

[134] Soraya Rana. 1999. Examining the role of local optima and schema processing in genetic search. PhD thesis. Colorado State University, Department of Computer Science, GENITOR Research Group in Genetic Algorithms; Evolutionary Computation, Fort Collins, CO, USA.

[135] Ingo Rechenberg. 1994. *Evolutionsstrategie '94*. Frommann-Holzboog Verlag, Bad Cannstadt, Stuttgart, Baden-Württemberg, Germany.

[136] Ingo Rechenberg. Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution. PhD thesis. Technische Universität Berlin; Friedrick Frommann Verlag, Stuttgart, Germany, Berlin, Germany.

[137] Uwe E. Reinhardt. 2011. What does 'economic growth' mean for americans? *The New York Times, Economix, Today's Economist* (2011). Retrieved from https://economix.blogs.nytimes.com/2011/09/02/what-does-economic-growth-mean-for-americans

[138] Franz Rothlauf. 2006. *Representations for genetic and evolutionary algorithms* (2nd ed.). Springer-Verlag, Berlin/Heidelberg. DOI:https://doi.org/10.1007/3-540-32444-5

[139] Y.A. Rozanov. 1977. *Probability theory: A concise course* (new ed.). Dover Publications, Mineola, NY, USA.

[140] Stuart J. Russell and Peter Norvig. 2002. *Artificial intelligence: A modern approach (AIMA)* (2nd ed.). Prentice Hall International Inc., Upper Saddle River, NJ, USA.

[141] Conor Ryan, John James Collins, and Michael O'Neill. 1998. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of the first european workshop on genetic programming (EuroGP'98), April 14–15, 1998, Paris, France* (Lecture Notes in Computer Science (LNCS)), Springer-Verlag GmbH, Berlin, Germany, 83–95. DOI:https://doi.org/10.1007/BFb0055930

[142] Paul Anthony Samuelson and William Dawbney Nordhaus. 2001. *Microeconomics* (17th ed.). McGraw-Hill Education (ISE Editions), Boston, MA, USA.

[143] Bruno Sareni and Laurent Krähenbühl. 1998. Fitness sharing and niching methods revisited. *IEEE Transactions on Evolutionary Computation (TEVC)* 2, 3 (1998), 97–106. DOI:https://doi.org/10.1109/4235.735432

[144] Guntram Scheithauer. 2018. *Introduction to cutting and packing optimization: Problems, modeling approaches, solution methods*. Springer International Publishing. DOI:https://doi.org/10.1007/978-3-319-64403-5

[145] J. Shekel. 1971. Test functions for multimodal search techniques. In *Fifth annual princeton conference on information science and systems*, Princeton University Press, Princeton, NJ, USA, 354–359.

[146] Guoyong Shi, Hitoshi Iima, and Nobuo Sannomiya. 1997. New encoding scheme for solving job shop problems by genetic algorithm. In *Proceedings of the 35th IEEE conference on decision and control (CDC'96), December 11–13, 1996, Kobe, Japan*, IEEE, 4395–4400. DOI:https://doi.org/10.1109/CDC.1996.577484

[147] Rob Shipman. 1999. Genetic redundancy: Desirable or problematic for evolutionary adaptation? In *Proceedings of the 4th international conference on artificial neural nets and genetic algorithms (ICANNGA'99), April 6–9, 1999, Protorož, Slovenia*, Springer-Verlag, Vienna, Austria, 337–344. DOI:https://doi.org/10.1007/978-3-7091-6384-9_57

[148] Oleg V. Shylo. 2019. Job shop scheduling (personal homepage). Retrieved from http://optimizizer.com/jobshop.php

[149] Sidney Siegel and N. John Castellan Jr. 1988. *Nonparametric statistics for the behavioral sciences*. McGraw-Hill, New York, NY, USA.

[150] Steven S. Skiena. 2008. *The algorithm design manual* (2nd ed.). Springer-Verlag, London, UK. DOI:https://doi.org/10.1007/978-1-84800-070-4

[151] James C. Spall. 2003. *Introduction to stochastic search and optimization*. Wiley Interscience, Chichester, West Sussex, UK. Retrieved from https://www.jhuapl.edu/ISSO/

[152] Robert H. Storer, S. David Wu, and Renzo Vaccari. 1992. New search spaces for sequencing problems with application to job shop scheduling. *Management Science* 38, 10 (1992), 1495–1509. DOI:https://doi.org/10.1287/mnsc.38.10.1495

[153] Marco Taboga. 2017. *Lectures on probability theory and mathematical statistics* (3rd ed.). CreateSpace Independent Publishing Platform (On-Demand Publishing, LLC), Scotts Valley, CA, USA. Retrieved from http://www.statlect.com/

[154] Ke Tang, Xiaodong Li, Ponnuthurai Nagaratnam Suganthan, Zhenyu Yang, and Thomas Weise. 2010. *Benchmark functions for the cec'2010 special session and competition on large-scale global optimization*. University of Science; Technology of China (USTC), School of Computer Science; Technology, Nature Inspired Computation; Applications Laboratory (NICAL), Hefei, Anhui, China.

[155] Oliver Theobald. 2018. *Statistics for absolute beginners* (Paperback ed.). Independently published.

[156] Marc Toussaint and Christian Igel. 2002. Neutrality: A necessity for self-adaptation. In *Proceedings of the IEEE congress on evolutionary computation (CEC'02), May 12-17, 2002, Honolulu, HI, USA*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1354–1359. DOI:https://doi.org/10.1109/CEC.2002.1004440

[157] Jelke Jeroen van Hoorn. 2015. Job shop instances and solutions. Retrieved from http://jobshop.jjvh.nl

[158] Jelke Jeroen van Hoorn. 2016. Dynamic programming for routing and scheduling: Optimizing sequences of decisions. PhD thesis. Vrije Universiteit Amsterdam, Amsterdam, The Netherlands. Retrieved from http://jobshop.jjvh.nl/dissertation

[159] Jelke Jeroen van Hoorn. 2018. The current state of bounds on benchmark instances of the job-shop scheduling problem. *Journal of Scheduling* 21, 1 (February 2018), 127–128. DOI:https://doi.org/10.1007/s10951-017-0547-8

[160] Jelke Jeroen van Hoorn, Agustín Nogueira, Ignacio Ojea, and Joaquim A. S. Gromicho. 2017. An corrigendum on the paper: Solving the job-shop scheduling problem optimally by dynamic programming. *Computers & Operations Research* 78, (2017), 381. DOI:https://doi.org/10.1016/j.cor.2016.09.001

[161] Petr Vilím, Philippe Laborie, and Paul Shaw. 2015. Failure-directed search for constraint-based scheduling. In *International conference integration of AI and OR techniques in constraint programming: 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* (Lecture Notes in Computer Science (LNCS) and Theoretical Computer Science and General Issues book sub series (LNTCS)), Springer, Cham, Switzerland, 437–453. DOI:https://doi.org/10.1007/978-3-319-18008-3_30

[162] Petr Vilím, Philippe Laborie, and Paul Shaw. 2015. Failure-directed search for constraint-based scheduling – detailed experimental results. Retrieved from http://vilim.eu/petr/cpaior2015-results.pdf

[163] Thomas Weise. 2009. *Global optimization algorithms – theory and application*. it-weise.de (self-published), Germany. Retrieved from http://www.it-weise.de/projects/book.pdf

[164] Thomas Weise. 2017. From standardized data formats to standardized tools for optimization algorithm benchmarking. In *Proceedings of the 16th IEEE conference on cognitive informatics & cognitive computing (ICCI*CC'17), July 26–28, 2017, University of Oxford, Oxford, UK*, IEEE Computer Society Press, Los Alamitos, CA, USA, 490–497. DOI:https://doi.org/10.1109/ICCI-CC.2017.8109794

[165] Thomas Weise. 2019. JsspInstancesAndResults: Results, data, and instances of the job shop scheduling problem. Retrieved from https://github.com/thomasWeise/jsspInstancesAndResults

[166] Thomas Weise, Raymond Chiong, and Ke Tang. 2012. Evolutionary optimization: Pitfalls and booby traps. *Journal of Computer Science and Technology (JCST)* 27, 5 (2012), 907–936. DOI:https://doi.org/10.1007/s11390-012-1274-4

[167] Thomas Weise, Raymond Chiong, Ke Tang, Jörg Lässig, Shigeyoshi Tsutsui, Wenxiang Chen, Zbigniew Michalewicz, and Xin Yao. 2014. Benchmarking optimization algorithms: An open source framework for the traveling salesman problem. *IEEE Computational Intelligence Magazine (CIM)* 9, 3 (2014), 40–52. DOI:https://doi.org/10.1109/MCI.2014.2326101

[168] Thomas Weise, Li Niu, and Ke Tang. 2010. AOAB – automated optimization algorithm benchmarking. In *Proceedings of the 12th annual conference companion on genetic and evolutionary computation (GECCO'10), July 7–11,2010, Portland, OR, USA*, ACM Press, New York, NY, USA, 1479–1486. DOI:https://doi.org/10.1145/1830761.1830763

[169] Thomas Weise, Alexander Podlich, and Christian Gorldt. 2009. Solving real-world vehicle routing problems with evolutionary algorithms. In *Natural intelligence for scheduling, planning and packing problems*, Raymond Chiong and Sandeep Dhakal (eds.). Springer-Verlag, Berlin/Heidelberg, 29–53. DOI:https://doi.org/10.1007/978-3-642-04039-9_2

[170] Thomas Weise, Alexander Podlich, Kai Reinhard, Christian Gorldt, and Kurt Geihs. 2009. Evolutionary freight transportation planning. In *Applications of evolutionary computing – proceedings of*

EvoWorkshops 2009: EvoCOMNET, EvoENVIRONMENT, EvoFIN, EvoGAMES, EvoHOT, EvoIASP, EvoINTERACTION, EvoMUSART, EvoNUM, EvoSTOC, EvoTRANSLOG, April 15–17, 2009, Tübingen, Germany (Lecture Notes in Computer Science (LNCS)), Springer-Verlag GmbH, Berlin, Germany, 768–777. DOI:https://doi.org/10.1007/978-3-642-01129-0_87

[171] Thomas Weise, Xiaofeng Wang, Qi Qi, Bin Li, and Ke Tang. 2018. Automatically discovering clusters of algorithm and problem instance behaviors as well as their causes from experimental data, algorithm setups, and instance features. *Applied Soft Computing Journal (ASOC)* 73, (2018), 366–382. DOI:https://doi.org/10.1016/j.asoc.2018.08.030

[172] Thomas Weise, Yuezhong Wu, Raymond Chiong, Ke Tang, and Jörg Lässig. 2016. Global versus local search: The impact of population sizes on evolutionary algorithm performance. *Journal of Global Optimization* 66, 3 (2016), 511–534. DOI:https://doi.org/10.1007/s10898-016-0417-5

[173] Thomas Weise, Yuezhong Wu, Weichen Liu, and Raymond Chiong. 2019. Implementation issues in optimization algorithms: Do they matter? *Journal of Experimental & Theoretical Artificial Intelligence (JETAI)* 31, (2019). DOI:https://doi.org/10.1080/0952813X.2019.1574908

[174] Thomas Weise, Michael Zapf, Raymond Chiong, and Antonio Jesús Nebro Urbaneja. 2009. Why is optimization difficult? In *Nature-inspired algorithms for optimisation*, Raymond Chiong (ed.). Springer-Verlag, Berlin/Heidelberg, 1–50. DOI:https://doi.org/10.1007/978-3-642-00267-0_1

[175] Frank Werner. 2013. Genetic algorithms for shop scheduling problems: A survey. In *Heuristics: Theory and applications*, Patrick Siarry (ed.). Nova Science Publishers, New York, NY, USA. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.718.2312&type=pdf

[176] L. Darrell Whitley. 1989. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the 3rd international conference on genetic algorithms (ICGA'89), June 4–7, 1989, Fairfax, VA, USA*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 116–121. Retrieved from http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.8195

[177] L. Darrell Whitley. 2016. 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*, ACM, 515–532. DOI:https://doi.org/10.1145/2908961.2926987

[178] L. Darrell Whitley, V. Scott Gordon, and Keith E. Mathias. 1994. Lamarckian evolution, the Baldwin effect and function optimization. In *Proceedings of the third conference on parallel problem solving from nature; international conference on evolutionary computation (PPSN III), October 9–14, 1994, Jerusalem, Israel* (Lecture Notes in Computer Science (LNCS)), Springer-Verlag GmbH, Berlin, Germany, 5–15. DOI:https://doi.org/10.1007/3-540-58484-6_245

[179] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. Retrieved from http://sci2s.ugr.es/keel/pdf/algorithm/articulo/wilcoxon1945.pdf

[180] David Paul Williamson, Leslie A. Hall, J. A. Hoogeveen, Cor A. J. Hurkens, Jan Karel Lenstra, Sergey Vasil'evich Sevast'janov, and David B. Shmoys. 1997. Short shop schedules. *Operations Research* 45, 2 (1997), 288–294. DOI:https://doi.org/10.1287/opre.45.2.288

[181] James M. Wilson. 2003. Gantt charts: A centenary appreciation. *European Journal of Operational Research (EJOR)* 149, (2003), 430–437. DOI:https://doi.org/10.1016/S0377-2217(02)00769-5

[182] Takeshi Yamada and Ryohei Nakano. 1992. A genetic algorithm applicable to large-scale job-shop instances. In *Proceedings of parallel problem solving from nature 2 (PPSN II), September 28–30, 1992, Brussels, Belgium*, Elsevier, Amsterdam, The Netherlands, 281–290.

[183] Takeshi Yamada and Ryohei Nakano. 1997. Genetic algorithms for job-shop scheduling problems. In *Proceedings of modern heuristic for decision support, March18–19, 1997, London, England, UK*, UNICOM seminar, 67–81.