An Introduction to Optimization Algorithms

Thomas Weise

2020-12-26

Contents

Preface

After writing *Global Optimization Algorithms – Theory and Applications* [205] during my time as PhD student a long time ago, I now want to write a more direct guide to optimization and metaheuristics. Currently, this book is in an early stage of development and work-in-progress, so expect many changes. It is available as pdf, html, epub, and azw3.

The text tries to introduce optimization in an accessible way for an audience of undergraduate and graduate students without background in the field. It tries to provide an intuition about how optimization algorithms work in practice, 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 actual results. We try to improve the algorithms step-by-step, moving from very simple approaches, which do not work well, to efficient metaheuristics.

```
@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   = {2020-12-26}
}
```

We use concrete examples and algorithm implementations written in Java. While originally designed for educational purposes, the code is general and may applicable to real research experiments, too. All of it is freely available in the repository *thomasWeise/aitoa-code* on GitHub under the MIT License. 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. Many of the charts and tables in the book have been generated with R scripts, whose source code is available in the aitoaEvaluate on GitHub under the MIT License, too. The experimental results are available in the repository aitoa-data. The text of the book itself is actively written and available in the repository *thomasWeise/aitoa* on GitHub. There, you can also submit *issues*, such as change requests, suggestions, errors, typos, or you can inform me that something is unclear, so that I can improve the book.

repository: http://github.com/thomasWeise/aitoa
commit: 2bf8b8a844a810daa8eefe6bc77bc96f65d17f7a
time and date: 2020-12-26 04:43:48 UTC+0000
example source repository: http://github.com/thomasWeise/aitoa-code
example source commit: 97666709e6ea4b473742cd8f032a16993c341410
experimental results: http://github.com/thomasWeise/aitoa-data
code for generating diagrams: http://github.com/thomasWeise/aitoaEvaluate

Prof. Dr. Thomas Weise
Institute of Applied Optimization (IAO),
School of Artificial Intelligence and Big Data,
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.

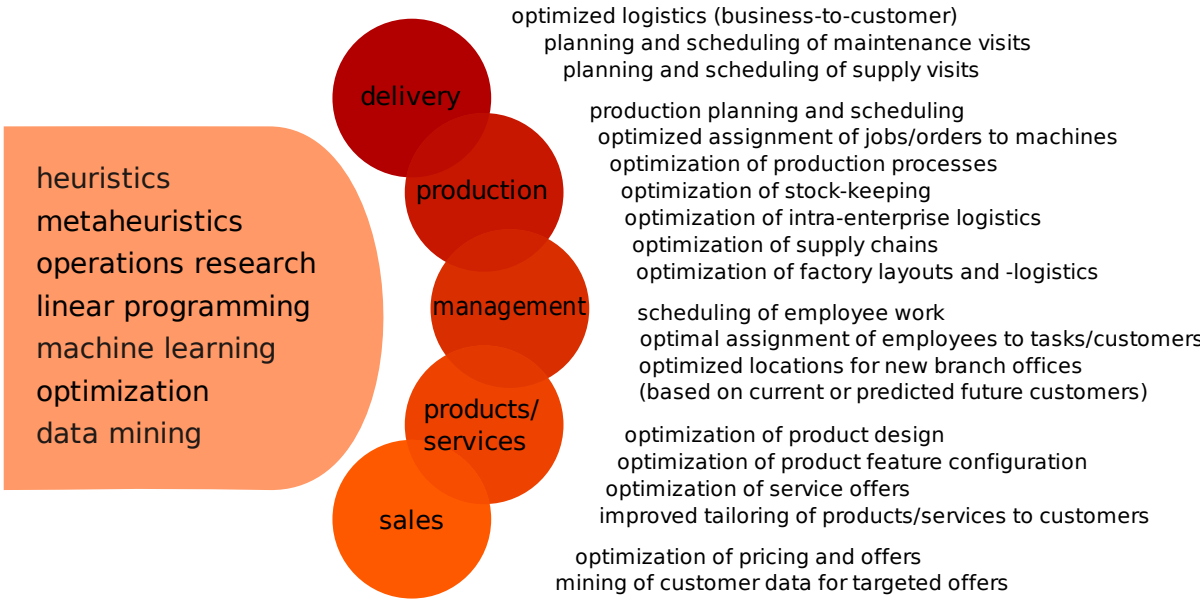

Figure 1.1: Examples for applications of optimization, computational intelligence, machine learning techniques in five fields of smart manufacturing: the production itself, the delivery of the products, the management of the production, the products and services, and the sales level.

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. 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. One key technology for achieving this is better planning via optimization.
3. Our world suffers from both depleting resources and too much pollution. Optimization can "give us more while needing less." It 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 try to answer all of these questions. We will explore quite a lot of different optimization algorithms. We will look at their actual implementations and we will apply them to example problems to see what their strengths and weaknesses are.

1.1 Examples

Let us first get a feeling about typical use cases of optimization.

1.1.1 Example: Layout of Factories

When we think about intelligent production, then there both dynamic and static aspects, as well as all sorts of nuances in between. The question how a factory should look like is a rather static, but quite important aspect. Let us assume we own a company and buy a plot of land to construct a new factory. Of course we know which products we will produce in this factory. We therefore also know the set of facilities that we need to construct, i.e., the workshops, storage depots, and maybe an administrative building. What we need to decide is where to place them on our land, as illustrated in Figure 1.2.

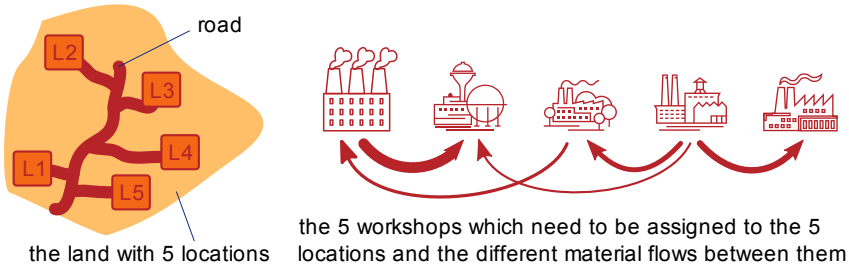

Figure 1.2: Illustrative sketch of a quadratic assignment scenario, where different buildings of a factory need to be laid out on a plot of land.

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

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

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

This scenario is called *quadratic assignment problem* (QAP) [31]. It has been subject to research since the 1950s [20]. QAPs appear in wide variety of scenarios such as the location of facilities on a plot of land or the placement of work stations on the factory floor. But even if we need to place components on a circuit board in a way that minimizes the total wire length, we basically have a QAP, too [190]! Despite being relatively simple to understand, the QAP is hard to solve [176].

1.1.2 Example: Route Planning for a Logistics Company

Another, more dynamic application area for optimization is logistics. Let us look at a typical real-world scenario from this field [211,212]: the situation of a logistics company that fulfills delivery tasks for its clients. A client can order one or multiple containers to be delivered to her location within a certain time window. She will fill the containers with goods, which are then to be transported to a destination

location, again within a certain time window. The logistics company may receive many such customer orders per day, maybe several hundreds or even thousands. The company may have multiple depots, where containers and trucks are stored. For each order, it needs to decide which container(s) to use and how to get them to the customer, as sketched in Figure 1.3. The trucks it owns may have different capacities and could, e.g., carry either one or two containers. Besides using trucks, which can travel freely on the map, it may also be possible to utilize trains. Trains have higher capacities and can carry many containers. Different from trucks, they must follow specific schedules. They arrive and depart at fixed times to/from fixed locations. For each possible vehicle, different costs could occur. Containers can be exchanged between different vehicles at locations such as parking lots, depots, or train stations.

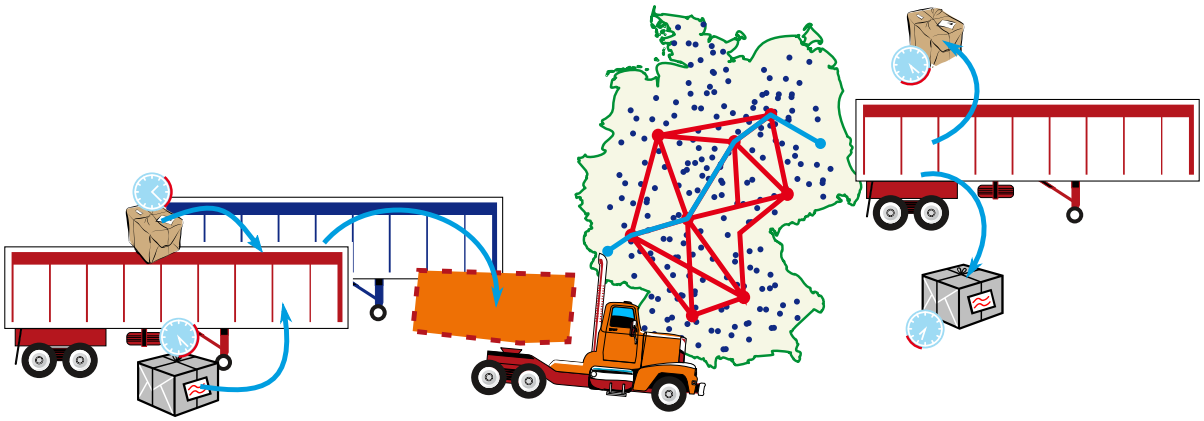

Figure 1.3: Illustrative sketch of logistics problems: Orders require us to pick up some items at source locations within certain time windows and deliver them to their destination locations, again within certain time windows. We need to decide which containers and vehicles to use and over which routes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

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.

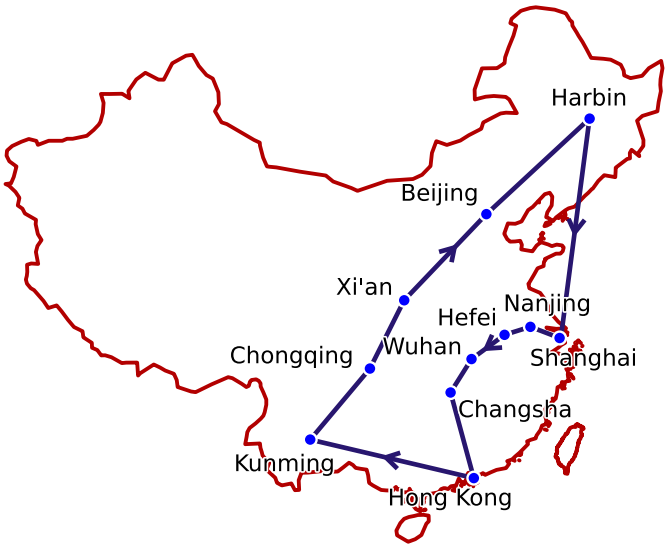

Figure 1.4: A Traveling Salesman Problem (TSP) through eleven cities in China.

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) [8,94,133], where the goal is to find the shortest round-trip tour through$n$ cities, as sketched in Figure 1.4. Many other scenarios can be modeled as such logistics questions, too: If a robot arm needs to several drill holes into a circuit board, finding the shortest tour means solving a TSP and will speed up the production process, for instance [91].

1.1.3 Example: Packing, Cutting Stock, and Knapsack

Let's say that your family is moving to a new home in another city. This means that you need to transport all of your belongings from your old to your new place, your PC, your clothes, maybe some furniture, a washing machine, and a fridge, as sketched in Figure 1.5. You cannot pack everything into your car at once, so you will have to drive back and forth a couple of times. But how often will you have to drive? Packing problems [68,180] 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.

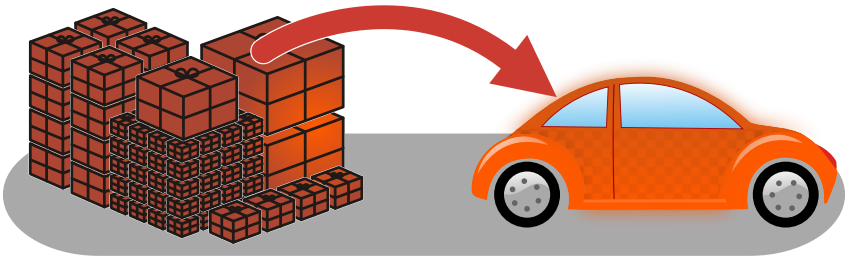

Figure 1.5: A sketch illustrating a packing problem.

Such bin packing problems exist in many variants and are very related to cutting stock problems [68]. They can be one-dimensional [56], 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 [136] 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 [141]).

1.1.4 Example: Job Shop Scheduling Problem

Another typical optimization task arises in manufacturing, namely the assignment ("scheduling") of tasks ("jobs") to machines in order to optimize a given performance criterion ("objective"). Scheduling [165,166] is one of the most active areas of operational research for more than six decades.

In the *Job Shop Scheduling Problem* (JSSP) [33,43,66,88,132,134], we have a factory ("shop") with several machines. We receive a set of customer orders for products which we have to produce. We know the exact sequence in which each product/order needs to pass through the machines and how long it will need at each machine. Each production job has one sub-job ("operation") for each machine on which it needs to be processed. These operations must be performed in the right sequence. Of course, no machine can process more than one operation at the same time. While we must obey

these constraints, we can decide about the time at which each of the operations should begin. Often, we are looking for the starting times that lead to the earliest completion of all jobs, i.e., the shortest makespan.

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

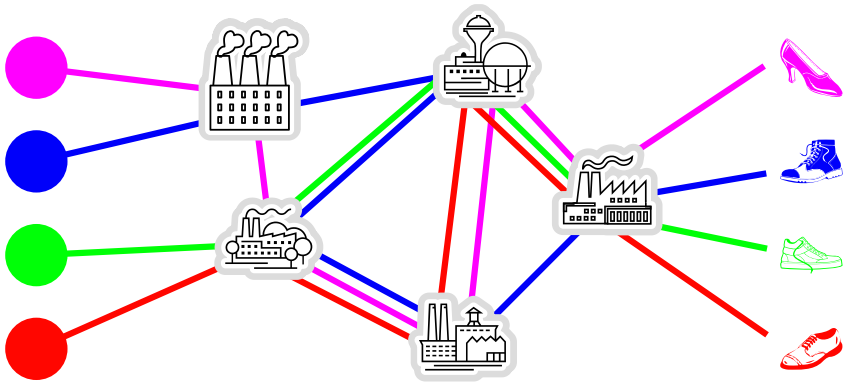

Figure 1.6: Illustrative sketch of a JSSP scenario with four jobs where four different types of shoe should be produced, which require different workshops ("machines") to perform different production steps.

This general scenario encompasses many simpler problems. For example, if we only produce one single product, then all jobs would pass through the same machines in the same order. Customers may be able to order different quantities of the product, so the operations of the different jobs for the same machine may need different amounts of time. This is the so-called Flow Shop Scheduling Problem (FSSP) – and it has been defined back in 1954 [121]!

Clearly, since the JSSP allows for an *arbitrary* machine order per job, being able to solve the JSSP would also enable us to solve the FSSP, where the machine order is fixed. We will introduce the JSSP in detail in Section 2.2.2 and use it as the main example in this book on which we will step-by-step exercise different optimization methods.

1.1.5 Summary

The four examples we have discussed so far are, actually, quite related. They all fit into the broad areas of operational research, which and smart manufacturing [52,106]. 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 linear equation. They require algorithms, they require computer science.

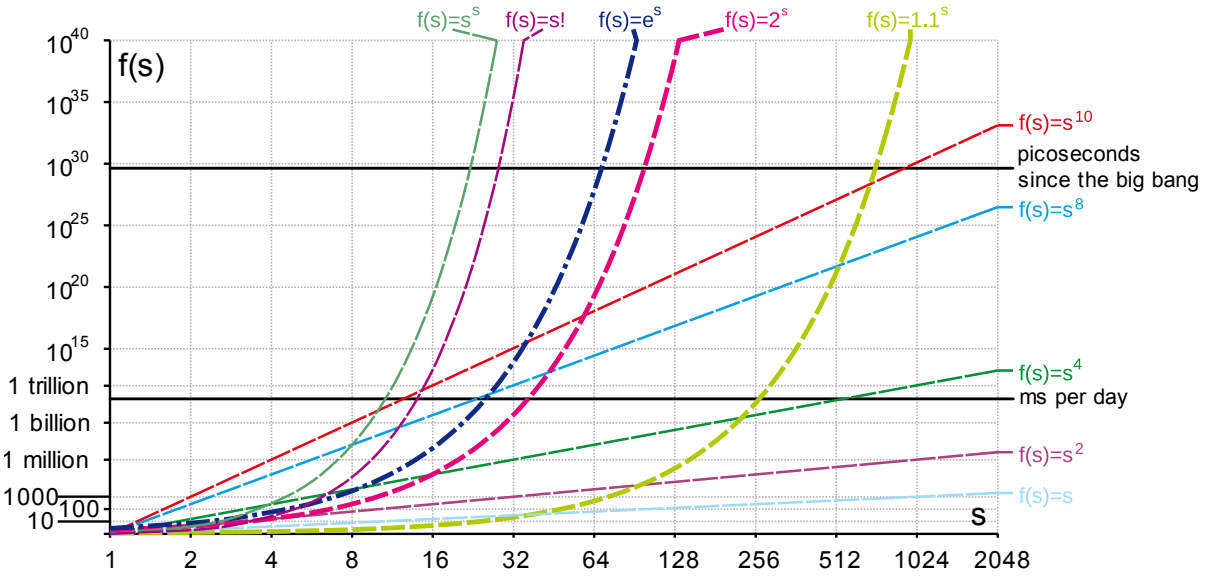

Figure 1.7: The growth of different functions in a log-log scaled plot. Exponential functions grow very fast, so that an algorithm which needs $\sim 2^s$ steps to solve an optimization problem of size s quickly becomes infeasible. (compare with Table 2.1 and Table 2.3)

Ever since primary school, we have learned many problems and types of equations that we can solve. Unfortunately, theoretical computer science shows that for many problems, the time we need to find the best-possible solution can grow *exponentially* with the number of involved variables in the worst case. The number of involved variables here could be the number of cities in a TSP, the number of jobs or machines in a JSSP, or the number of objects to pack in a, well, packing problem. A big group of such complicated problems are called $\mathcal{NP}$-hard [37,134]. Unless some unlikely breakthrough happens [44,124], there will be many problems that we cannot always solve exactly within reasonable time. Each and every one of the example problems discussed belongs to this type!

As sketched in Figure 1.7, the exponential function rises very quickly. One idea would be to buy more computers for bigger problems and to simply parallelize the computation. Well, parallelization can provide a linear speed-up at best, but we are dealing with problems where the runtime requirements may double every time we add a new decision variable. And no: Quantum computers are not the answer. Most likely, they cannot even solve these problems qualitatively faster either [1].

So what can we do to solve such problems? The exponential time requirement occurs if we make *guarantees* about the solution quality, especially about its optimality, over all possible scenarios. What we can do, therefore, is that we can trade-in the *guarantee* of finding the best possible solution for lower runtime requirements. We can use algorithms from which we hope that they find a good *approximation* of the optimum, i.e., a solution which is very good with respect to the objective function, but which do not *guarantee* that 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.

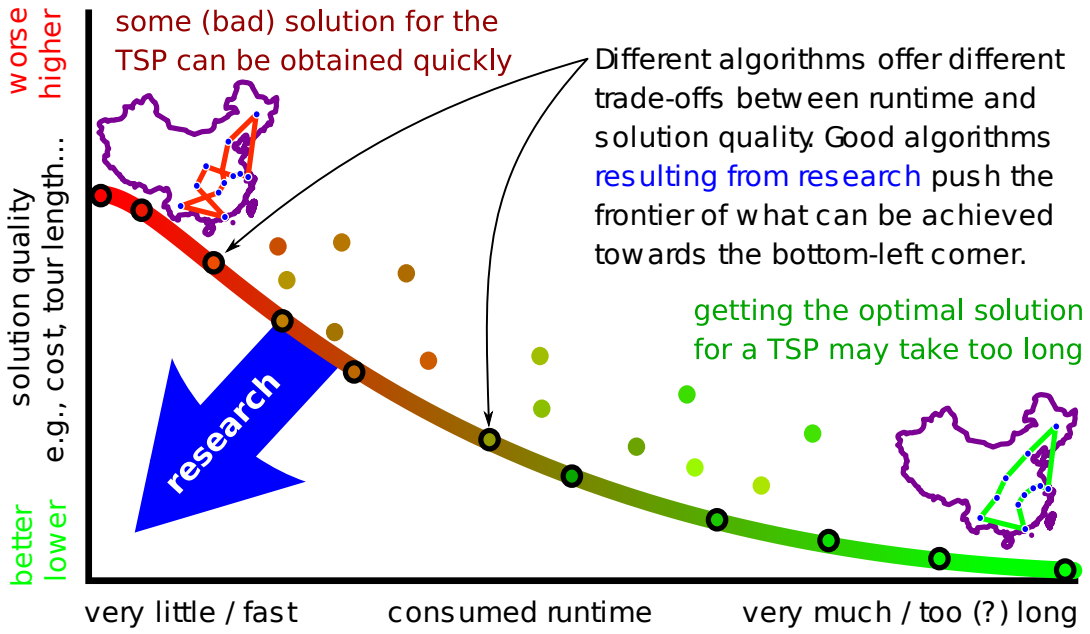

Figure 1.8: The trade-off between solution quality and runtime.

In Figure 1.8 we illustrate this idea on the example of the Traveling Salesman Problem [8,94,133] briefly mentioned in Section 1.1.2. The goal of solving the TSP is to find the shortest round trip tour through n cities. The TSP is $\mathcal{NP}$-hard [79,94]. Today, it is possible to solve many large instances of this problem to optimality by using sophisticated *exact* algorithms [45,46]. Yet, finding the *shortest possible tour* for a particular TSP might still take many years if you are unlucky. Finding just *one tour* is, however, very easy: I can write down the cities in any particular order. Of course, I can visit the cities in an arbitrary order. That is an entirely valid solution, and I can obtain it basically in 0 time. This "tour" would probably be very bad, very long, and generally not a good idea.

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

1.2.2 Good Solutions within Acceptable 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 [45,46], 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 [105,153,219] and their implementation from the algorithm specification alone can take months [215]. 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 1.1.2), 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 1.1.4 [33,66,88,132,134]. 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 3.5.5).

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 [66,88,132,134], 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 – the operations – one for each machine. Each job may need to pass through these machines in a different order. The operation 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 operation indices j. The jobs would pass through all machines in the same order but may have different processing times (due to the different quantities).

We may also have scenarios where customers can order different types of products, say the same liquid soap, but either in bottles or big cannisters. Then, different machines may be needed for different orders. This is similar to the situation illustrated in Figure 1.6, 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 operation 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 [3,9,57,66,74,135,191,224]. Their data can be found (sometimes partially) in several repositories in the internet, such as

- the *OR-Library* managed by Beasley [18],
- the comprehensive set of JSSP instances provided by van Hoorn [199,201], where also state-of-the-art results are listed,
- Oleg Shylo's Page [185], which, too, contains up-to-date experimental results,
- Éric Taillard's Page, or, finally,
- my own repository jsspInstancesAndResults [207], 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 [135] with 15 jobs and 10 machines,
3. instance `swv15` by Storer et al. [191] with 50 jobs and 10 machines, and
4. instance `yn4` by Yamada and Nakano [224] 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, http://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.

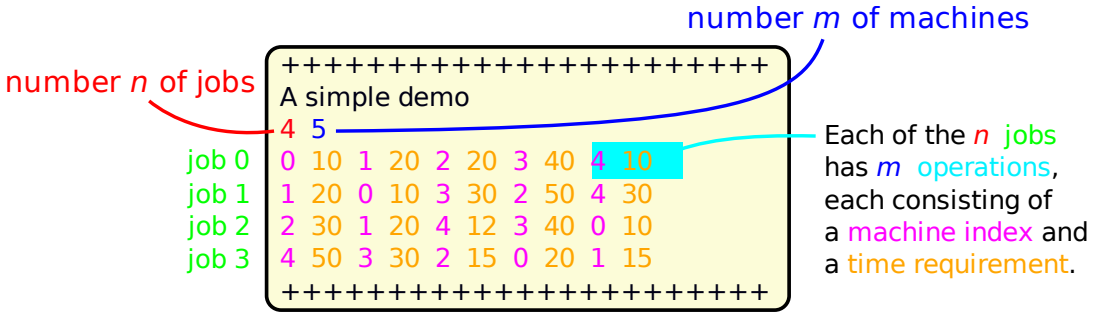

Figure 2.1: The meaning of the text representing our demo instance of the JSSP, as an example of the format used in the OR-Library.

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 operations of one job $i \in 0 \ldots (n-1)$. Each operation j is specified as a pair of two numbers, the ID $M_{i,j}$ of the machine that is to be used (violet), from the interval $0 \ldots (m-1)$, followed by the number of time units $T_{i,j}$ the job will take on that machine. The order of the operations 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 operations can directly be processed by their corresponding machines. Job 3 first needs to be processed by machine 4 for 50 time units, then by machine 3 for 30 time units, then by machine 2 for 15 time units, then by machine 0 for 20 time units, and finally by machine 1 for 15 time units. It would not be allowed to first send Job 3 to any machine different from machine 4 and after being processed by machine 4, it must be processed by machine 3 – 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 operation 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 1.1.4 that our goal is to complete the jobs, i.e., the production tasks, as soon as possible. Hence, a candidate solution should tell us what to do, i.e., how to process the jobs on the machines.

2.3.2.1 Idea: Gantt Chart

This is basically what Gantt chart [126,223] 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 operations of each job are assigned to time windows on their corresponding machines.

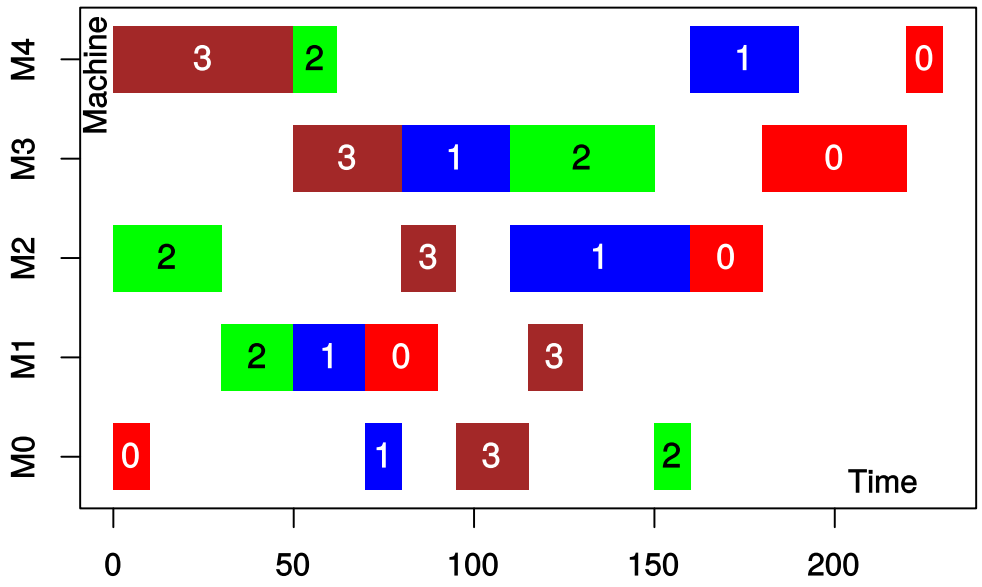

Figure 2.2: One example candidate solution for the demo instance given in Figure 2.1: A Gantt chart assigning a time window to each job on each machine.

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 operation j of job i is painted in the row of machine $M_{i,j}$ and its length equals the time requirement $T_{i,j}$.

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 operations 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)

```java
public class JSSPCandidateSolution {
  public int[][] schedule;
}
```

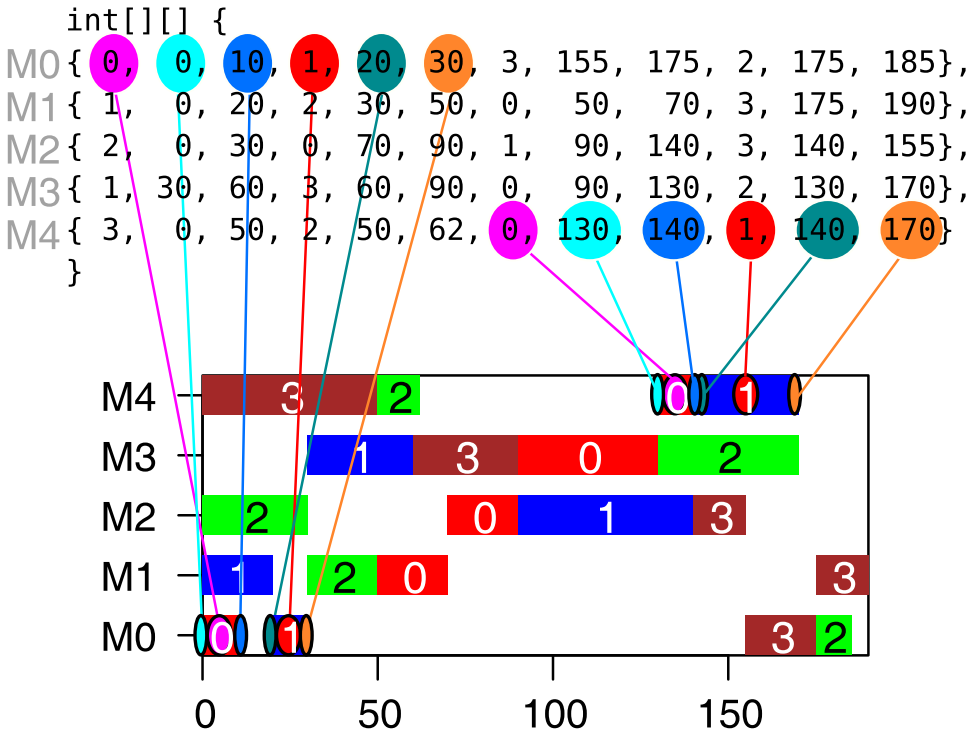

Figure 2.3: An example how the internal `int[][]` data of the `JSSPCandidateSolution` class maps to a Gantt chart.

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 operation to a starting time, leading to $m * n$ integer values per candidate solution [200]. But the presented structure – illustrated on an example in Figure 2.3 – 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 operations of all jobs must be assigned to their respective machines and properly be completed,
2. only the jobs and machines specified by the problem instance must occur in the chart,
3. a operation will must be assigned a time window on its corresponding machine which is exactly as long as the operation needs on that machine,
4. the operations cannot intersect or overlap, each machine can only carry out one job at a time, and
5. the precedence constraints of the operations 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.

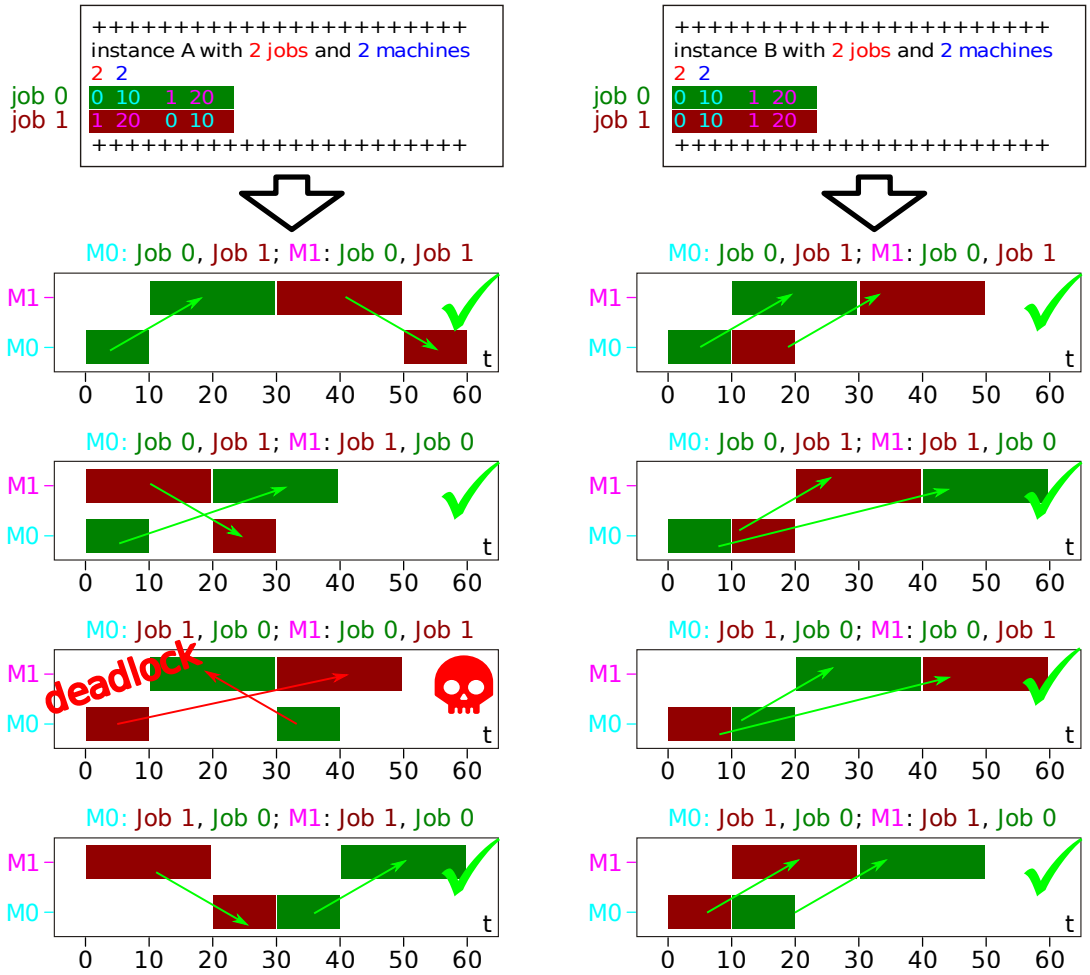

Figure 2.4: Two different JSSP instances with $m = 2$ machines and $n = 2$ jobs, one of which has only three feasible candidate solutions while the other has four.

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 operations are assigned to proper machines and receive the right processing times. Still, the schedule is infeasible, because it cannot be executed or written down without breaking the precedence constraint.

Hence, there are only three out of four possible Gantt charts that work for this problem instance. For a

problem instance where the jobs need to pass through all machines in the same sequence, however, all possible Gantt charts will work, as also illustrated in 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 1.7).

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   double evaluate(Y y);
3 }
```

2.4.2 Example: Job Shop Scheduling

As stated in Section 1.1.4, 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 operation of job 0, executed on machine 4.

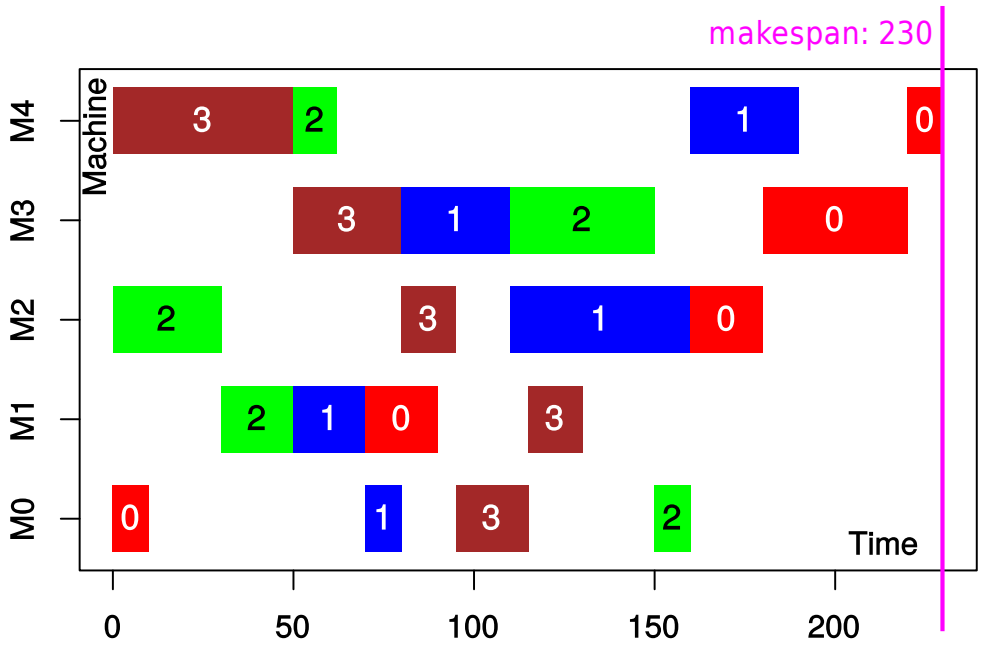

Figure 2.5: The makespan (purple), i.e., the time when the last job is completed, for the example candidate solution illustrated in Figure 2.2 for the demo instance from Figure 2.1.

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 the Lower Bound of the Objective Function

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" [30]. 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.

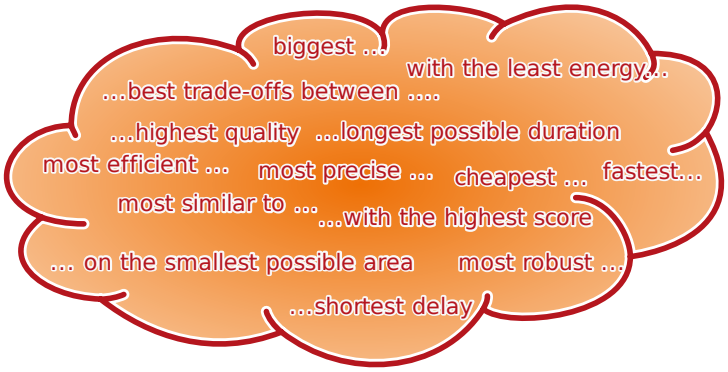

Figure 2.6: Optimization is the search for superlatives [30].

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 [121]. 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 [37,134], 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 directly 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 operations. 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 operations assigned to this machine. But it may also include a least initial idle time a_j: If the operations for machine j never come first in their job, then for each job, we need to sum up the runtimes of the operations coming before the one on machine j. The least initial idle time a_j is then the smallest of these sums. Similarly, there is a least idle time c_j at the end if these operations 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 [66]. Often, we may not have such lower bounds, but it does never hurt to think about them, because it will provide us with some more understanding about the nature of the problem we are trying to solve.

Even if we have a lower bound for the objective function, we can usually not know whether any solution of that quality actually exists. In other words, we do not know whether it is actually possible to find a schedule whose makespan equals the lower bound. There simply may not be any way to arrange

the jobs such that no operation stalls any other operation. This is why the value $\mathrm{lb}(f)$ is called lower bound: We know no solution can be better than this, but we do not know whether a solution with such minimal makespan exists.

However, if our algorithms produce solutions with a quality close to $\mathrm{lb}(f)$, we know that we are doing well. Also, if we would actually find a solution with that makespan, then we would know that we have perfectly solved the problem. 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	[143,199,203,204]
`la24`	15	10	872	935	[9,199]
`swv15`	50	10	2885	2885	Equation (2.2)
`yn4`	20	20	818	929	[199,203,204]

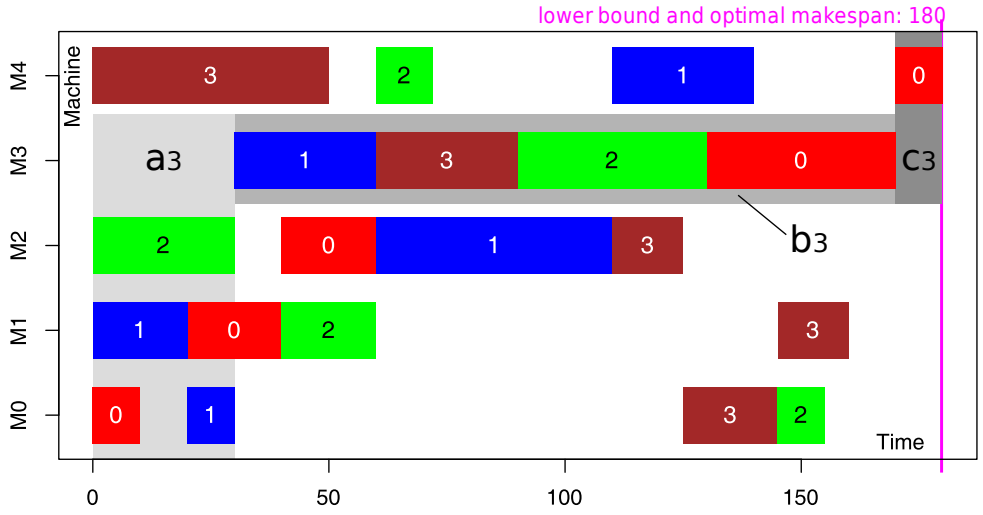

Figure 2.7: The globally optimal solution of the demo instance Figure 2.1, whose makespan happens to be the same as the lower bound.

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 operation 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 operations 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   double evaluate(Y y);
3   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 [128]. 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.

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

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

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.

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.[3] 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 [2,40,217,225], 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 operations, one for each machine. We could simply begin by choosing one job and placing its first operation on the machine to which it belongs, i.e., write it into the Gantt chart. Then we again pick a job, take the first not-yet-scheduled

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

operation of this job, and "add" it to the end of the row of its corresponding machine in the Gantt chart. Of course, we cannot pick a job whose operations all have already be assigned. We can continue doing this until all 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 $m = 5$ operations that must be distributed to the machines. We use a string of length $m * n = 20$ denoting the priority of the operations. We *know* the order of the operations per job as part of the problem instance data $\mathcal{I}$. We therefore do not need to 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*.

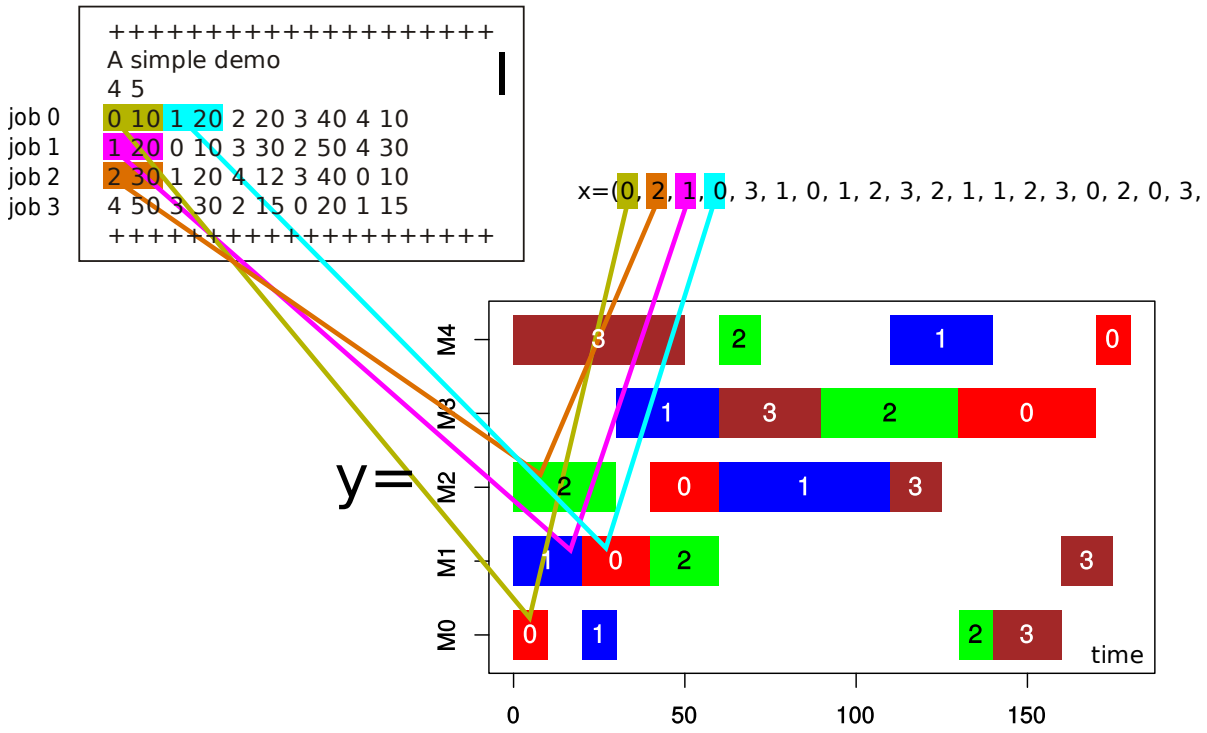

Figure 2.8: Illustration of the first four steps of the representation mapping of an example point in the search space to a candidate solution.

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 operation can definitely be executed, i.e., won't cause a deadlock, because it is the first operation 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 operation of job 0, we now would like to schedule its second operation. According to the instance data, the second operation takes place on machine 1 and will take 20 time units. We know that completing the first operation 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 operation 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 operation and then can be processed by machine 1. No deadlock can occur, as we made sure that the first operation 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 operation, go to machine 0. Its first operation 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 operation 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 operation, 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 operation of job 0 finishes on time index 40 at machine 1. Hence, we can begin with the third operation 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 operation $i > 0$ of any given job to a machine, then we already had assigned all operations at smaller indices first. No deadlock can occur and y must therefore be feasible.

In Listing 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 operation 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. [81], Bierwirth [24,25], and Shi et al. [182], 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* [24]. The mapping γ will ensure that the order of the operations 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 operations 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 operations would be assigned to "machine types" instead of "machines" and there could be more than one machine per machine type, then the representation mapping could assign the operations to the next machine of their type which becomes idle. Our representation also trivially covers the situation where each job may have more than m operations, i.e., where a job may need to cycle back and pass one machine twice. It is also suitable to simple scenarios, such as the Flow Shop Problem, where all jobs pass through the machines in the same, pre-determined order [66,80,217].

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 $\mathbb{X}$ [182]. 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 1.7 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	$\approx 1.462*10^{121}$	$\approx 2.293*10^{164}$

name	n	m	$\lvert\mathbb{Y}\rvert$	$\lvert\mathbb{X}\rvert$
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)

```java
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.mJobs[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 k-ary *search operator* $\mathrm{searchOp} : \mathbb{X}^k \mapsto \mathbb{X}$ is a left-total relation which accepts k 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 ($k = 0$, see Listing 2.8), which sample a new point from the search space without using any information from an existing points,
- unary operators ($k = 1$, see Listing 2.9), which sample a new point from the search space based on the information of one existing point,
- binary operators ($k = 2$, see Listing 2.10), which sample a new point from the search space by combining information from two existing points, and
- ternary operators ($k = 3$), 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     extends ISetupPrintable {
3   void apply(X dest, Random random);
4 }
```

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

```
1 public interface IUnarySearchOperator<X>
2     extends ISetupPrintable {
3   void apply(X x, X dest, 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 `java.util.Random`, a pseudorandom number generator.

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

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

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

Definition 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   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 [37,134]. 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 [117,222] 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 [142]. 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

Optimization problems are solved by optimization algorithms. We can roughly divide these into *exact* and *heuristic* methods.

An exact algorithm guarantees to find the optimal solution if sufficient runtime is granted. This required runtime might, in the worst case, exceed what we can afford, in particular for $\mathcal{NP}$-hard problems, such as the JSSP. Alternatively, many exact methods can be halted before completing their run and they can then still provide an approximate solution (without the guarantee that it is optimal).

For heuristic algorithms, this directly is the basic premise; They give us some approximate solution relatively quickly. They either do not make any guarantees at all how good it will be or, sometimes, provide some bound guarantee (like: "This solution will not cost more than two times of the optimal cost.") Simple heuristics are usually tailor-made for specific problems, like the TSP or JSSP.

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

Metaheuristics [42,82,83,205] are the most important class of algorithms that we explore in this book. 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. We will introduce several such general algorithms in this book. We explore them by again using the Job Shop Scheduling Problem (JSSP) from Section 1.1.4 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 provide an approximate result during almost any time of its execution.

All metaheuristics – and many other optimization and machine learning methods – are anytime algorithms [27]. 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 (and stop the optimization process if we want to). 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

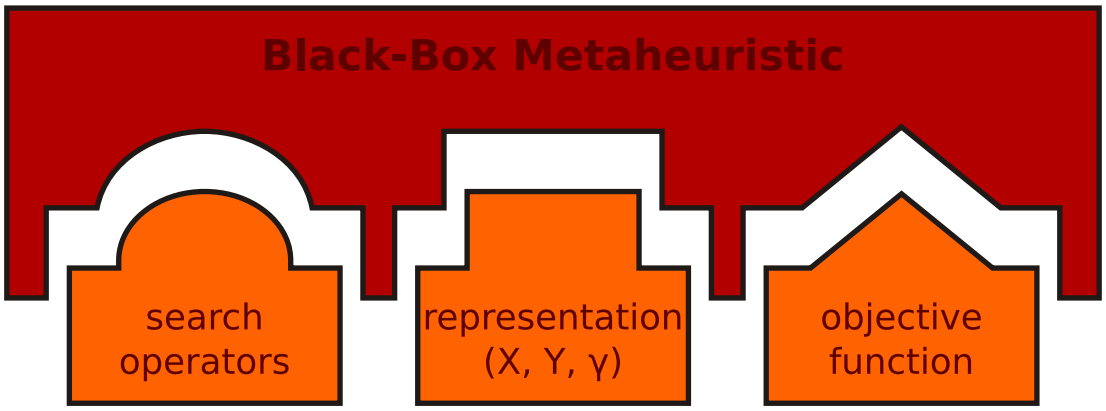

Figure 3.1: The black-box character of many metaheuristics, which can often accept arbitrary search operators, representations, and objective functions.

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.

There are different degrees of how general black-box metaheuristics can be. For the most general algorithms, it does not matter what exactly the search operators do or even what data structure is used as search space $\mathbb{X}$. For them, 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). Then, 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 $\mathbb{X}$.

Then there are also black-box metaheuristics that demand a special type of search space, e.g., a specific subset of the n-dimensional real numbers ($\mathbb{X} \subset \mathbb{R}^n$), bit strings of a given length, or permutations of the first n natural numbers. These algorithms can still be general, as they may make very few assumptions about the nature of the objective function f defined over the solution space $\mathbb{Y}$.

The solution space is relevant for the human operator using the algorithm only, the search space is what the algorithm works on. Of course, in many cases, $\mathbb{X} = \mathbb{Y}$.

Thus, a black-box metaheuristic is a general algorithm into which we can plug representations, objective functions, and often also search operations as needed by a specific application. This is 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

Before, I promised that we will implement all the algorithms discussed in this book.

If we would be dealing with "classical" algorithms, things would be somewhat easier: A classical algorithm has clearly defined input and output data structures. Dijkstra's shortest path algorithm [60], for instance, gets fed with a graph of weighted edges and a start node and will return the paths of minimum weight to all other nodes (or a specified target node). In Machine Learning, the situation is quite similar: We would have a lot of specialized algorithms for clearly defined situations. The input and output data would usually adhere to some basic, fixed structures. If you implement k-means clustering [77,104], for instance, you have real vectors coming in and k real vectors going out of your algorithm and that's that. Deep Learning [87] basically takes, as input, a set of labeled real vectors (plus a network structure) and, as output, produces the vector of weights for the network. However, we have to deal with the black-box concept, meaning that our algorithms will be very variable in terms of the data structures we can feed to them. Matter in fact: *Any* of the three scenarios above can be modeled as optimization problem. *Any* of them can be tackled with (most of) the metaheuristics in this book as well!

This is challenging from a programming perspective, especially when we try to tackle this in an educational setting, where stuff should not be overly complicated. What we want is to not just implement general algorithms, but also be able to execute them and obtain their results in a convenient fashion. Ideally, we do not want to be bothered with too much book keeping or the creation of log files and such and such. It should be possible to implement the most general type of black-box methods as well as problem-specific optimization methods and to run them in a uniform environment.

I therefore try to define a simple API for black-box optimization that combines all of our considerations far. The goal is to make the implementation of metaheuristics as simple as possible. We do this by clearly dividing between the optimization algorithms for solving a problem on one side and the structural components of the problem on the other side. The algorithms that we will implement will be general black-box methods. At the same time, we will develop the components that we need to plug into them to solve JSSPs as educational example.

We therefore first need to consider what an optimization process 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 (which we will discuss later) 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. The interface 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    Random getRandom();
4    ISpace<X> getSearchSpace();
5    double getBestF();
6    double getGoalF();
7    void getBestX(X dest);
8    void getBestY(Y dest);
9    long getConsumedFEs();
10   long getLastImprovementFE();
11   long getMaxFEs();
12   long getConsumedTime();
13   long getLastImprovementTime();
14   long getMaxTime();
15 }
```

Actually, such an interface does not need to expose the representation mapping γ and objective function f as separate components to an optimization algorithm. It is sufficient if the interface directly implements an `evaluate` that takes, as input, an element $x \in \mathbb{X}$, internally performs the representation mapping $y = \gamma(x)$, then invokes the objective function $f(y)$, and returns its result. This `evaluate` method could then even be implemented such that it remembers the best-so-far-solution. We then no longer need to keep track of it in the optimization itself.

Of course, we can also implement logging of the search progress inside of `evaluate`, which would make this functionality available to all of our experiments in a transparent fashion. Furthermore, we could also keep track of the total number of calls to the objective function as well as of the consumed runtime. This, in turn, can be used to implement the termination criterion.

All in all, this interface allows us to create transparent implementations that

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 function,

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 instantiating it. 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 via the builder, we can provide the termination criterion, representation mapping, and objective function. Additionally, we also need to provide the functionality to instantiate and copy the data structures making up the spaces $\mathbb{X}$ and $\mathbb{Y}$. On one hand, these are needed for the internal book-keeping so that `evaluate` can internally make a copy of the best-so-far elements in $\mathbb{X}$ and $\mathbb{Y}$). On the other hand, the black-box optimization algorithms that we will implement also must be able to make such copies. Since we do not make any assumption about the Java `classes` corresponding to the spaces $\mathbb{X}$ and $\mathbb{Y}$, we add another easy-to-implement and very simple interface, namely `ISpace`, see Listing 3.2. It can be implemented for each problem type and then provides the required functionality.

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   Z create();
3   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 really needs a single method, `solve`. This method will use the functionality provided by the `IBlackBoxProcess` handed to it as parameter `process`. The implemented algorithm can generate new points in the search space `X` and send 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> extends ISetupPrintable {
2   void solve(IBlackBoxProcess<X, Y> process);
3 }
```

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. The most general black-box metaheuristics may be able to deal with any search- and solution space, as long they are provided with the right operators. Still, 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.mLength];
4   }
5   public void copy(int[] from, int[] to) {
6     System.arraycopy(from, 0, to, 0, this.mLength);
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)

```
public class JSSPSolutionSpace
    implements ISpace<JSSPCandidateSolution> {
  public JSSPCandidateSolution create() {
    return new JSSPCandidateSolution(this.instance.m,
        this.instance.n);
  }
  public void copy(JSSPCandidateSolution from,
      JSSPCandidateSolution to) {
    int n = this.instance.n * 3;
    int i = 0;
    for (int[] s : from.schedule) {
      System.arraycopy(s, 0, to.schedule[i++], 0, n);
    }
  }
}
```

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 structures and features "make" 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 [76,129], 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, Random random) {
4  // create first sequence of jobs: n-1, n-2, ..., 0
5      for (int i = this.n; (--i) >= 0;) {
6        dest[i] = i;
7      }
8  // copy this m-1 times: n-1, n-2, ..., 0, n-1, ... 0, n-1, ...
9      for (int i = dest.length; (i -= this.n) > 0;) {
10       System.arraycopy(dest, 0, dest, i, this.n);
11     }
12 // now randomly shuffle the array: create a random sequence
13     RandomUtils.shuffle(random, dest, 0, dest.length);
14   }
15 }
```

The `apply` method of our implemented operator creates one random point in the JSSP 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>
    extends Metaheuristic0<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 the nullary operator: Fill data structure with a random
// but valid point from the search space.
    this.nullary.apply(x, process.getRandom());

// Evaluate the point: process.evaluate automatically applies the
// representation mapping and calls objective function. The
// objective value is ignored here (not stored anywhere), but
// "process" will remember the best solution. Thus, 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.

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 roughly 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 operations.

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
swv15	2885	5935	6600	6563	346	0s	1
yn4	929	1754	2036	2039	125	0s	1

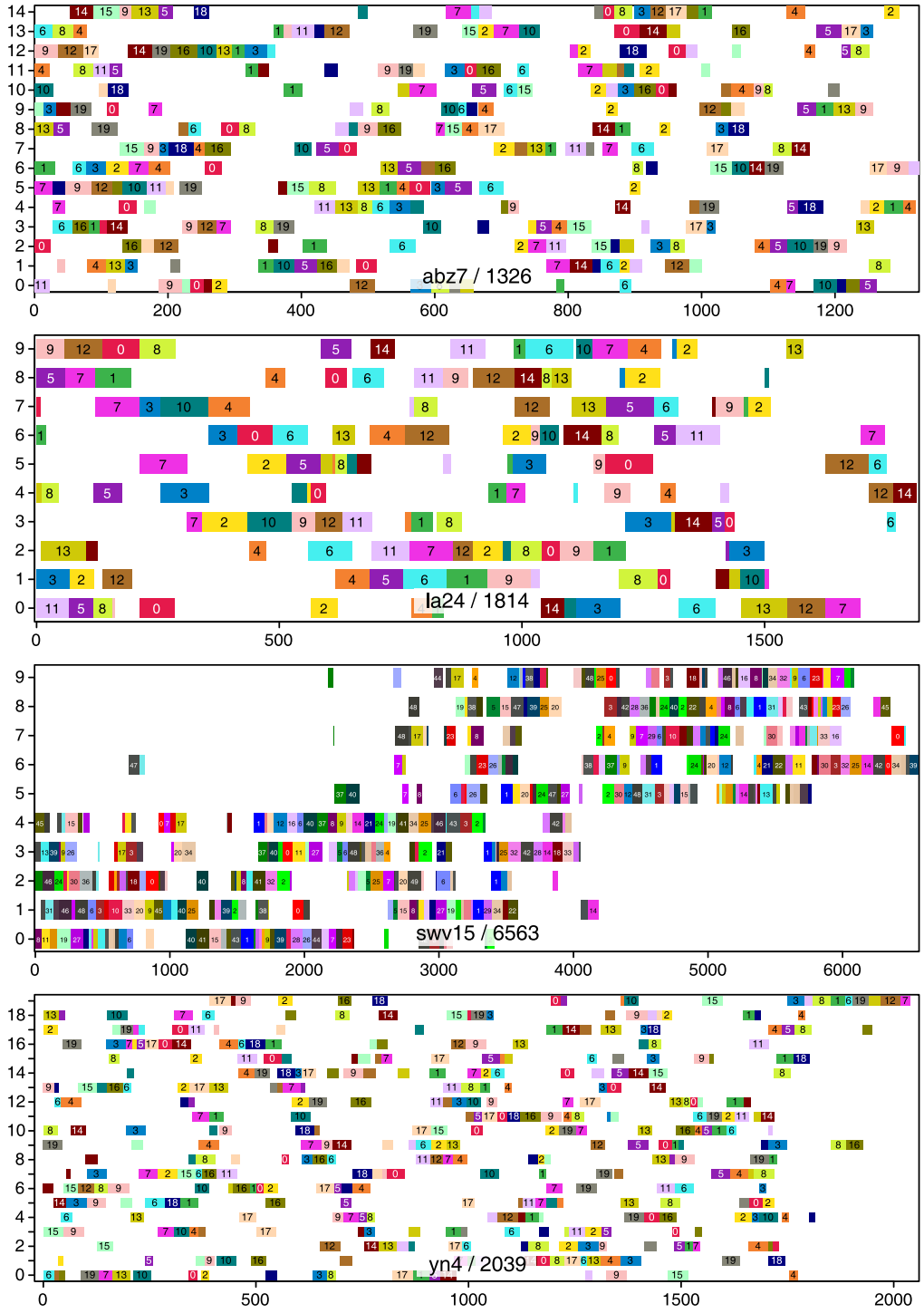

Figure 3.2: The Gantt charts of the median solutions obtained by the `1rs` algorithm. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

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 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 [188]. 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.

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      extends Metaheuristic0<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      Random random = process.getRandom();// get random gen
7
8      do { // Repeat until budget is exhausted.
9        this.nullary.apply(x, random); // Create random point in X.
10 // Evaluate the point: process.evaluate applies the
11 // representation mapping and calls objective function. It
12 // remembers the best solution, so the caller can obtain it.
13       process.evaluate(x);
14     } while (!process.shouldTerminate()); // do until time is up
15   }
16 }
```

The algorithm can be described as follows:

1. Set the best-so-far objective value z to $+\infty$ and the best-so-far candidate solution y to NULL.
2. Create random point x' in search space $\mathbb{X}$ by 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 z' of y' by invoking the objective function $z' = f(y')$.
5. If z' is better than best-so-far-objective value z, i.e., $z' < z$, then
   a. store z' in z and
   b. store y' in y.
6. If the termination criterion is not met, return to *step 2*.
7. Return the best-so-far objective value z and the best solution y to the user.

In actual program code, *steps 3 to 5* can again 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 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	**0s**	**1**
		rs	**895**	**947**	**949**	**12**	85s	6’512’505
la24	935	1rs	1487	1842	1814	165	**0s**	**1**
		rs	**1153**	**1206**	**1208**	**15**	82s	15’902’911
swv15	2885	1rs	5935	6600	6563	346	**0s**	**1**
		rs	**4988**	**5166**	**5172**	**50**	87s	5’559’124
yn4	929	1rs	1754	2036	2039	125	**0s**	**1**
		rs	**1460**	**1498**	**1499**	**15**	76s	4’814’914

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 operations 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.

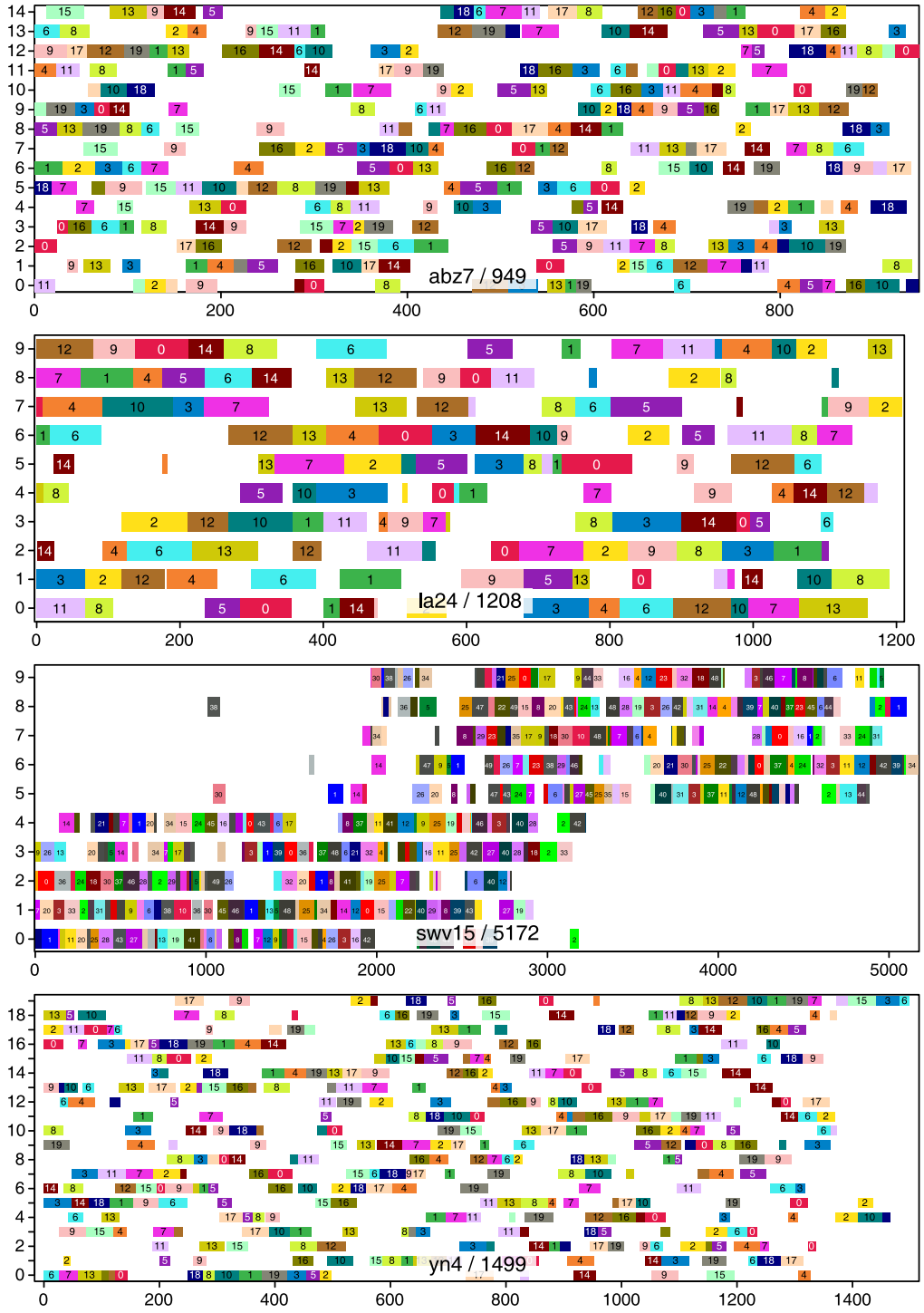

Figure 3.3: The Gantt charts of the median solutions obtained by the `rs` algorithm. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

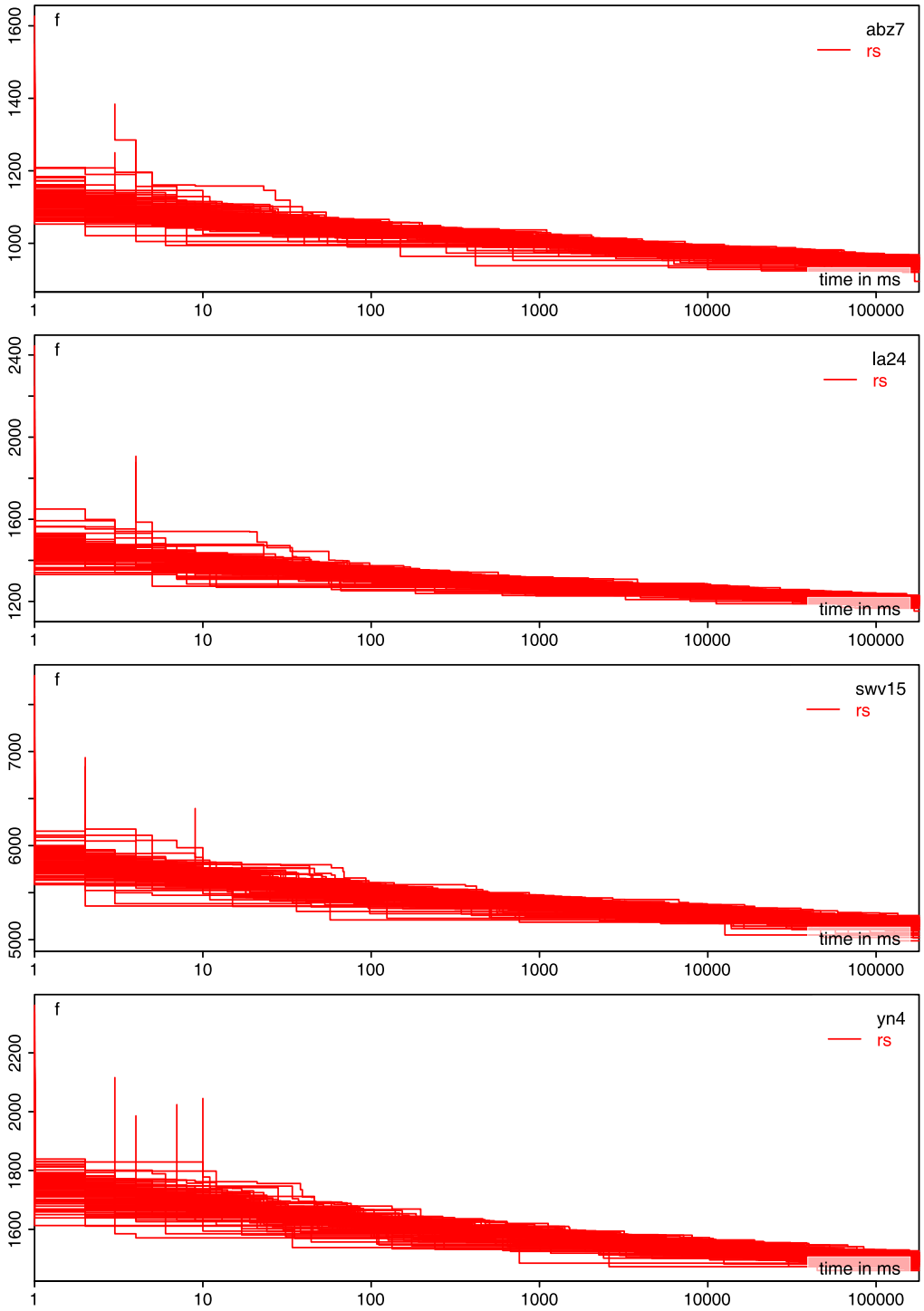

Figure 3.4: The progress of the `rs` algorithm over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis).

3.2.3.3 Progress over Time and the Law of Diminishing Returns

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 algorithm steadily and nicely improves 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, the algorithm improves the solution quality 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* [177] 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 the 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 the amount of time we need to invest to get any improvement would keep to increase. The fact that random sampling can be parallelized perfectly does not help much here, as we would need to provide an exponentially increasing number of processors to keep improving the solution quality.

3.2.4 Summary

With random sampling, we now have a very primitive way to tackle optimization problems. In each step, the algorithm generates a new, entirely random candidate solution. It remembers the best solution that it encounters and, after its computational budget is exhausted, returns it to the user.

Obviously, this algorithm cannot be very efficient. But we already also learned one method to improve the result quality and reliability of optimization methods: restarts. Restarting an optimization can be beneficial if the following conditions are met:

1. If we have a budget limitation, then most of the improvements made by the algorithm must happen early during the run. If the algorithm already uses its budget well can keep improving even close to its end, then it makes no sense to stop and restart. The budget must be large enough so that multiple runs of the algorithm can complete or at least deliver reasonable results.
2. Different runs of the algorithm must generate results of different quality. A restarted algorithm is still *the same* algorithm. It just exploits this variance, i.e., we will get something close to the best result of multiple runs. If the different runs deliver bad results anyway, doing multiple runs will not solve the problem.

Above we said that random sampling is not a very efficient algorithm. This is true in most reasonable scenarios. In problems where information about existing good solutions does not help us in any way to find new good solutions, we cannot really do better than random sampling. In most reasonable problems that one may try to solve, however, such information is helpful. Random sampling then is also a basic yardstick: An optimization algorithm that does not significantly outperform random sampling is useless.

3.3 Hill Climbing

Our first algorithm, random sampling, is 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. If the problem that we try to solve is entirely without structure, then this is already the best we can do. But our JSSP problem is not without structure. For example, we can assume that if we swap the last two jobs in a schedule, the makespan of the resulting new schedule should still be somewhat similar to the one of the original plan. Actually, most reasonable optimization problems have a lot of structure. We therefore should try to somehow make use of the information gained from sampling candidate solutions.

Local search algorithms [112,205] offer one idea for how to do that. They remember one point x^m in the search space $\mathbb{X}$. In every step, a local search algorithm investigates $g \geq 1$ points x_i which are

derived from and are similar to x^m. From the joined set of these g points and x^m, only one point is chosen as the (new or old) x^m. All the other points are 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 which is called *causality* [168,169,208,216]. If we have two points in the search space that only differ a little bit, then they likely map to similar schedules, which, in turn, likely have similar makespans. This means 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

For now, let us limit ourselves to local searches creating $g = 1$ new point in each iteration. The question arises how we can create a candidate solution which is similar to, but also slightly different from, one that 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.
5. Swap the values at indexes i and j in x'.
6. Return the now modified copy x' of x.

Point 4 is important since swapping the same values makes no sense, as we would then get $x' = x$. Then, also the mappings $\gamma(x)$ and $\gamma(x')$ would be the same, i.e., we would actually not make a "move" and just waste time.

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)

```
public class JSSPUnaryOperator1Swap
    implements IUnarySearchOperator<int[]> {
  public void apply(int[] x, int[] dest,
      Random random) {
// copy the source point in search space to the dest
    System.arraycopy(x, 0, dest, 0, x.length);

// choose the index of the first operation to swap
    int i = random.nextInt(dest.length);
    int jobI = dest[i]; // remember job id

    for (;;) { // try to find a location j with a different job
      int j = random.nextInt(dest.length);
      int jobJ = dest[j];
      if (jobI != jobJ) { // we found two locations with two
        dest[i] = jobJ; // different values
        dest[j] = jobI; // then we swap the values
        return; // and are done
      }
    }
  }
}
```

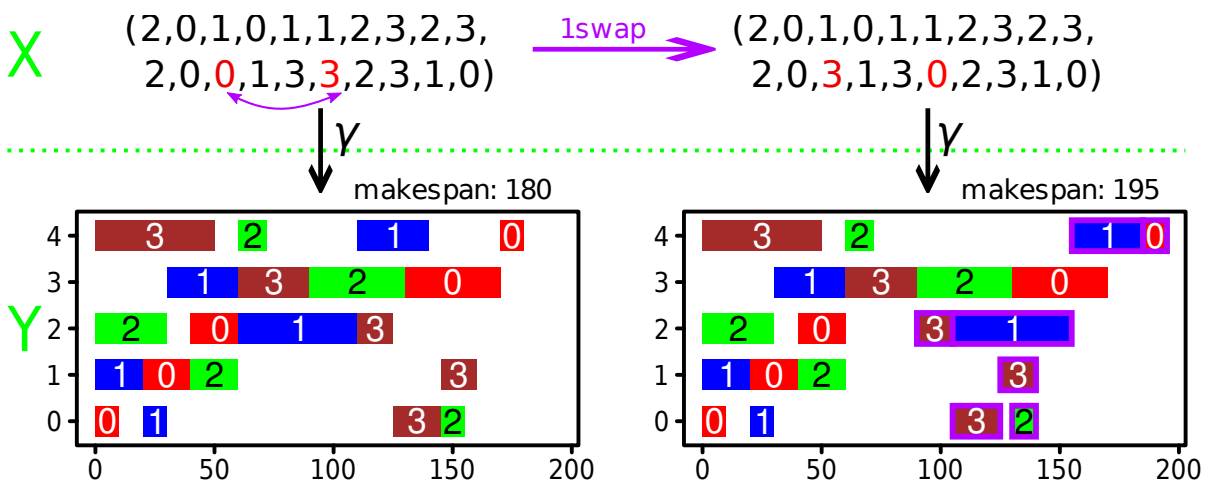

Figure 3.5: An example for the application of `1swap` to an existing point in the search space (top-left) for the `demo` JSSP instance. It yields a slightly modified copy (top-right) with two jobs swapped. If we map these to the solution space (bottom) using the representation mapping γ, the changes 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 operations 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 operations 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 operation to be executed on machine 4. This pushes back the last operation 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 in average and the number of even better solutions in the neighborhood tends to get smaller. However, if we would have been at x' instead, an application of `1swap` could well have resulted in x. In summary, we can hope that the chance to find a really good solution by iteratively sampling the neighborhoods of good solutions is higher compared to 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 [173,187,205] is the simplest implementation of local search. It is also sometimes called localized random search [188]. It proceeds as follows:

1. Create one random point x in the 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. Repeat until the termination criterion is met:
   a. Apply the unary search operator to x to get a 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 $z' < z$, then store x' in x, store y' in y, and store z' in z.
5. Return the best encountered objective value z and the best encountered solution y to the user.

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)

```
public class HillClimber<X, Y>
    extends Metaheuristic1<X, Y> {
  public void solve(IBlackBoxProcess<X, Y> process) {
// initialize local variables xCur, xBest, random
// Create starting point: a random point in the search space.
    this.nullary.apply(xBest, random); // xBest = random point
    double fBest = process.evaluate(xBest); // map & evaluate

    while (!process.shouldTerminate()) {
// Create a slightly modified copy of xBest and store in xCur.
      this.unary.apply(xBest, xCur, random);
// Map xCur from X to Y and evaluate candidate solution.
      double fCur = process.evaluate(xCur);
      if (fCur < fBest) { // we found a better solution
// Remember best objective value and copy xCur to xBest.
        fBest = fCur;
        process.getSearchSpace().copy(xCur, xBest);
      } // Otherwise, i.e., fCur >= fBest: Just forget xCur.
    } // Repeat until computational budget is exhausted.
  } // `process` has remembered the best candidate solution.
}
```

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

If you are wondering what would happen if we would accept the new solution x' also if $z' = z$, i.e., replace the $z' < z$ with an $z' \leq z$ in *point 4.d* of the algorithm definition: This algorithm is called $(1+1)$ EA and discussed later in Section 3.4.6.1.

3.3.2.2 Results on the JSSP

We now plug our unary operator 1swap into our hc algorithm and apply it 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	1rs	1131	1334	1326	106	**0s**	**1**
		rs	895	947	949	**12**	85s	6’512’505
		hc_1swap	**717**	**800**	**798**	28	0s	16’978
la24	935	1rs	1487	1842	1814	165	**0s**	**1**
		rs	1153	1206	1208	**15**	82s	15’902’911
		hc_1swap	**999**	**1095**	**1086**	56	0s	6’612
swv15	2885	1rs	5935	6600	6563	346	**0s**	**1**
		rs	4988	5166	5172	**50**	87s	5’559’124
		hc_1swap	**3837**	**4108**	**4108**	137	1s	104’598
yn4	929	1rs	1754	2036	2039	125	**0s**	**1**
		rs	1460	1498	1499	**15**	76s	4’814’914
		hc_1swap	**1109**	**1222**	**1220**	48	0s	31’789

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 median time med(t) consumed until the algorithm converges is not more than one seconds. The median number of consumed FEs med(FEs) to find the best solutions per run is between 7000 and 105’000, i.e., between one 50^{th} and one 2500^{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, even though it is very far from being useful. For instance, on instance abz7, it delivers better best and mean results than all four Genetic Algorithms (GAs) presented in [122]. 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 five of them. In Section 3.4, we will later introduce (better-performing!) Evolutionary Algorithms, to

which GAs belong.

The Gantt charts of the median solutions of `hc_1swap` are illustrated in Figure 3.6. They are more compact than those discovered by `rs` and illustrated 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.

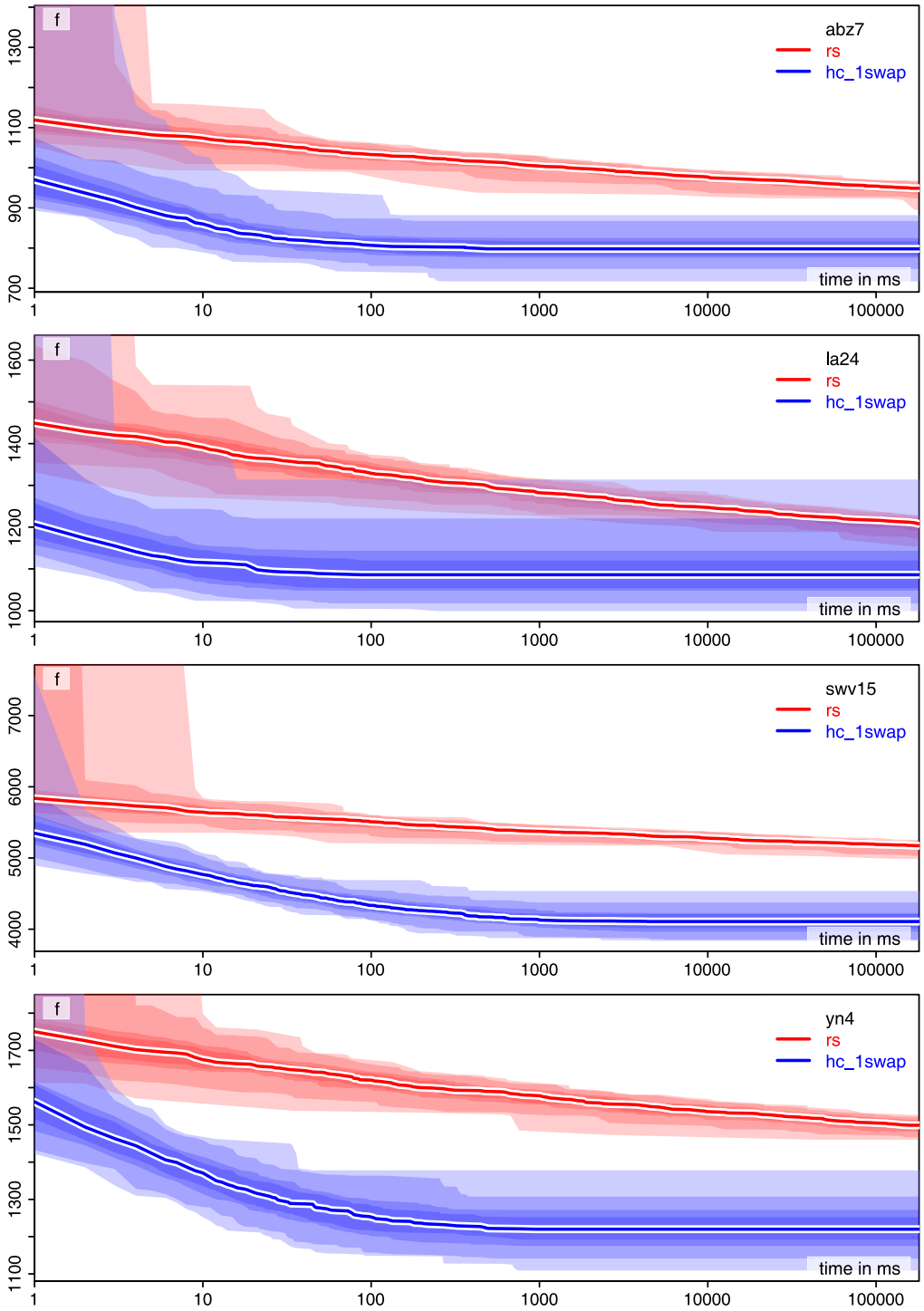

Figure 3.7: The median of the progress of the `hc_1swap` and `rs` algorithm over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis). Different from Figure 3.4, we do not plot the single runs but only shade areas between quantiles.

Figure 3.7 shows how both hc_1swap and rs progress over time. In Figure 3.4, we plotted every individual run. This time, we plot the median (see Section 4.4.2) of achieved quality at each time step as thick line. In the background, we plot the whole range of the values as semi-transparent region. The more runs fall into one region, the stronger we plot the color: In the outer borders of the lightest color shade mark the range between the best and worst results of any run at a given time. The next stronger-shaded region contains about 95% of the runs. Then follow by 68% of the runs while the strongest-shaded region holds half of the runs.

It should be noted that I designed the experiments in such a way that there were 101 different random seeds per instance. For each instance, all algorithms use the same random seeds, i.e., the hill climber and random sampling start with the same initial solutions. Still, the shaded regions of the two 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, usually before 1000 milliseconds have been consumed. This fits well to the values med(t) given in Table 3.3. With the exception of instance la24, where two runs of the hill climber performed exceptionally bad, 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

Upon close inspection of the results, we notice that we are again 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 (1 second vs. 3 minutes). 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 solution, only that these eventually do not mark improvements anymore. Then, 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. [208,216]

Due to the black-box nature of our basic hill climber algorithm, it is not really possible to know when the complete neighborhood of the current solution has already been tested. We thus cannot know whether or not the algorithm is trapped in a local optimum and has *prematurely converged*. However, we can try to guess it: If there has not been any improvement for a high number L of steps, then the

current point x in the search space 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-so-far* candidate solution in a special variable y_b 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.
2. Set the best-so-far objective value z_b to $+\infty$ and the best-so-far candidate solution y_b to `NULL`.
3. Create a random point x in the 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. If $z < z_b$, then store z in z_b and store y in y_b.
7. Repeat until the termination criterion is met:
   a. Apply the unary search operator to x 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 $z' < z$, then
      i. store z' in z and store x' in x and
      ii. set C to 0.
      iii. If $z' < z_b$, then store z' in z_b and store y' in y_b.

      otherwise to *step 7d*, i.e., if $z' \geq z$, then
      iv. increment C by 1.
      v. If $C \geq L$, then perform a restart by going back to *step 3*.
8. Return best encountered objective value z_b and the best encountered solution y_b to the user.

Now this algorithm – implemented in Listing 3.11 – is a bit more elaborate. Basically, we embed the original hill climber into a loop. This inner hill climber will stop after a certain number L of unsuccessful search steps, which then leads to a new round in the outer loop. In combination with the `1swap` operator, we refer to this algorithm as `hcr_L_1swap`, where `L` is to be replaced with the actual value of the parameter L.

3.3.3.2 The Right Setup

We now realize that we do not know which value of L is good. If we pick it too low, then the algorithm will restart before it actually converges to a local optimum, i.e., stop while it could still improve. If we pick it too high, we waste runtime and do fewer restarts than what we could do.

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      extends Metaheuristic1<X, Y> {
3    public void solve(IBlackBoxProcess<X, Y> process) {
4  // omitted for brevity initialize local variables xCur, xBest,
5  // random, failsBeforeRestart, and failCounter=0
6      while (!(process.shouldTerminate())) { // outer loop: restart
7        this.nullary.apply(xBest, random); // start=random solution
8        double fBest = process.evaluate(xBest); // evaluate it
9        long failCounter = 0L; // initialize counters
10
11       while (!(process.shouldTerminate())) { // inner loop
12         this.unary.apply(xBest, xCur, random); // try to improve
13         double fCur = process.evaluate(xCur); // evaluate
14
15         if (fCur < fBest) { // we found a better solution
16           fBest = fCur; // remember best quality
17           process.getSearchSpace().copy(xCur, xBest); // copy
18           failCounter = 0L; // reset number of unsuccessful steps
19         } else { // ok, we did not find an improvement
20           if ((++failCounter) >= this.failsBeforeRestart) {
21             break; // jump back to outer loop for restart
22           } // increase fail counter
23         } // failure
24       } // inner loop
25     } // outer loop
26   } // process has stored best-so-far result
27 }
```

If we do not know which value for a parameter is reasonable, we can always do an experiment to investigate. Since the order of magnitude of the proper value for L is not yet clear, it makes sense to test exponentially increasing numbers. Here, we test the powers of two from $2^7 = 128$ to $2^{18} = 262'144$. For each value, we plot the scaled median result quality over the 101 runs in Figure 3.8. In this diagram, the horizontal axis is logarithmically scaled.

From the plot, we can confirm our expectations: Small numbers of L perform bad and high numbers of L cannot really improve above the basic hill climber. Actually, if we would set L to a number larger than the overall budget, then we would obtain exactly the original hill climber, as it would never perform any restart. For different problem instances, different values of L perform good, but $L \approx 2^{14} = 16'384$ seems to be a reasonable choice for three of the four instances.

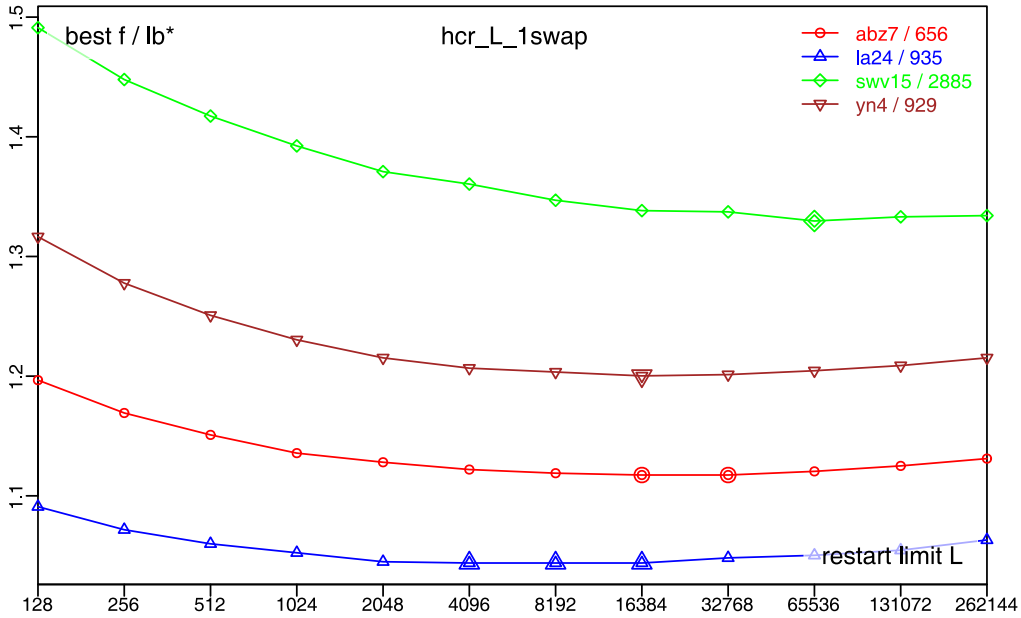

Figure 3.8: The median result quality of the `hcr_1swap` algorithm, divided by the lower bound $\mathrm{lb}(f)^\star$ from Table 2.2 over different values of the restart limit parameter L. The best values of L on each instance are marked with bold symbols.

3.3.3.3 Results on the JSSP

The performance indicators of three settings of our hill climber with restarts in comparison with the plain hill climber are listed in Table 3.4. We know that $L = 2^{14}$ seems a reasonable setting. Additionally, we also list the adjacent setups, i.e., give the results for $L \in \{2^{13}, 2^{14}, 2^{15}\}$.

Table 3.4: The results of the hill climber `hcr_L_1swap` with restarts for values of L from 213, 2^{14}, and 2^{15}. `hcr_L_1swap` restarts after L unsuccessful search moves. 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	**0s**	**16’978**
		`hcr_8192_1swap`	719	734	734	**5**	83s	17’711’879
		`hcr_16384_1swap`	**714**	732	**733**	6	91s	18’423’530
		`hcr_32768_1swap`	716	**732**	**733**	6	73s	15’707’437

$\mathcal{I}$	$\mathrm{lb}f$	setup	best	mean	med	sd	med(t)	med(FEs)
la24	935	hc_1swap	999	1095	1086	56	**0s**	**6'612**
		hcr_8192_1swap	956	**975**	**976**	**6**	84s	35'242'182
		hcr_16384_1swap	953	976	**976**	7	80s	34'437'999
		hcr_32768_1swap	**951**	979	980	8	90s	36'493'494
swv15	2885	hc_1swap	3837	4108	4108	137	**1s**	**104'598**
		hcr_8192_1swap	3745	3881	3886	**37**	91s	12'892'041
		hcr_16384_1swap	3752	3859	3861	42	92s	11'756'497
		hcr_32768_1swap	**3677**	**3853**	**3858**	37	89s	11'562'962
yn4	929	hc_1swap	1109	1222	1220	48	**0s**	**31'789**
		hcr_8192_1swap	1084	1117	1118	11	89s	13'258'408
		hcr_16384_1swap	1081	1115	**1115**	**11**	91s	14'804'358
		hcr_32768_1swap	**1075**	**1114**	1116	13	85s	13'126'688

Table 3.4 shows us that the restarted algorithms hcr_L_1swap almost always provide better best, mean, and median solutions than hc_1swap. Only the overall best result of hcr_8192_1swap on abz7 is worse than for hc_1swap – on all other instances and for all other quality metrics, hc_1swap loses.

The standard deviations of the end results of the variants with restarts are also always smaller, meaning that these algorithms perform more reliably. Their median time until they converge is now higher, which means that we make better use of our computational budget.

As a side note, the median and mean result of the three listed setups of our very basic hcr_L_1swap algorithms for instance la24 are already better than the best result (982) delivered by the Grey Wolf Optimization algorithm proposed in [118]. In other words, even with such a simple algorithm we can already achieve results not very far from recent publications…

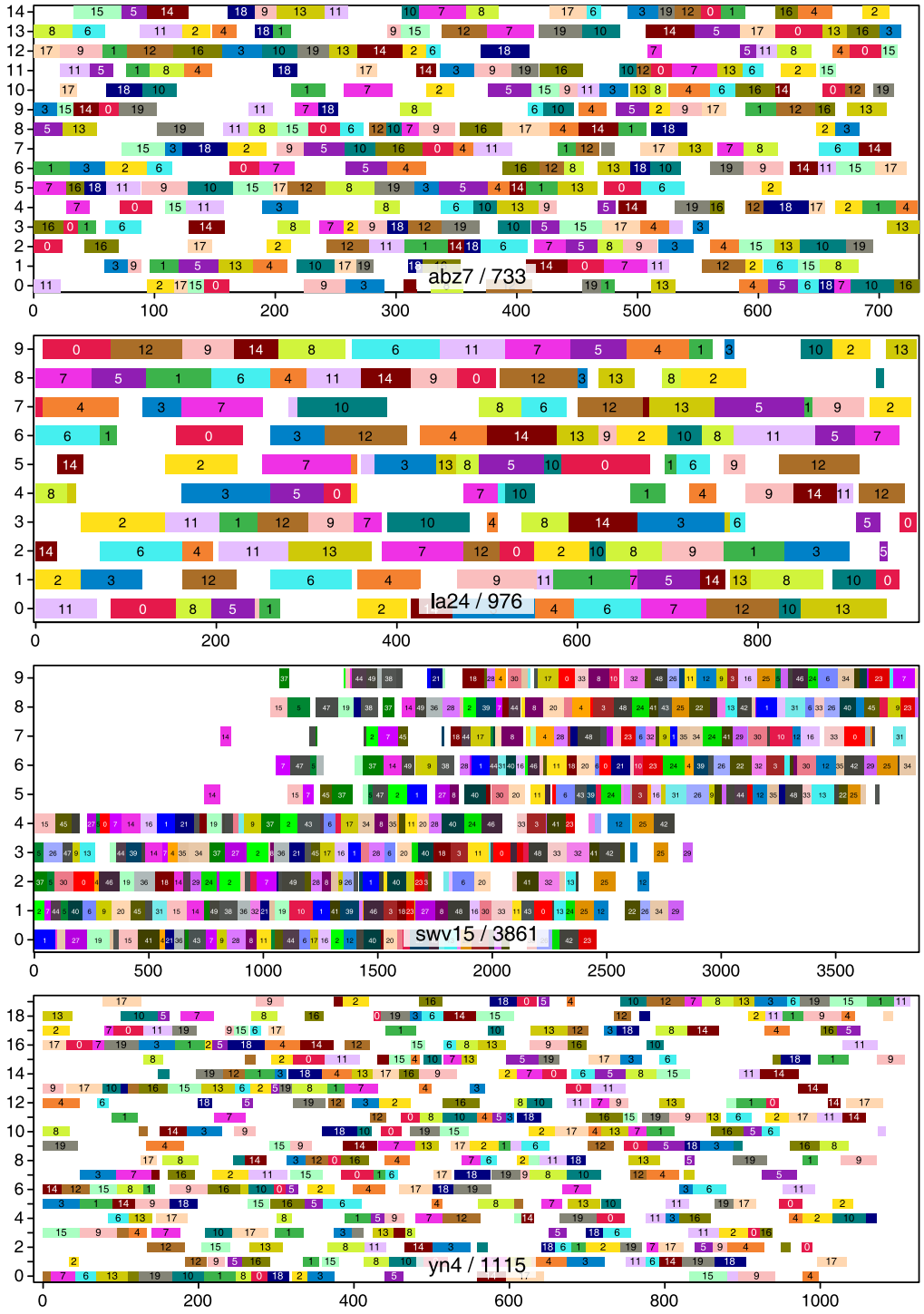

Figure 3.9: The Gantt charts of the median solutions obtained by the `hcr_16384_1swap` algorithm. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

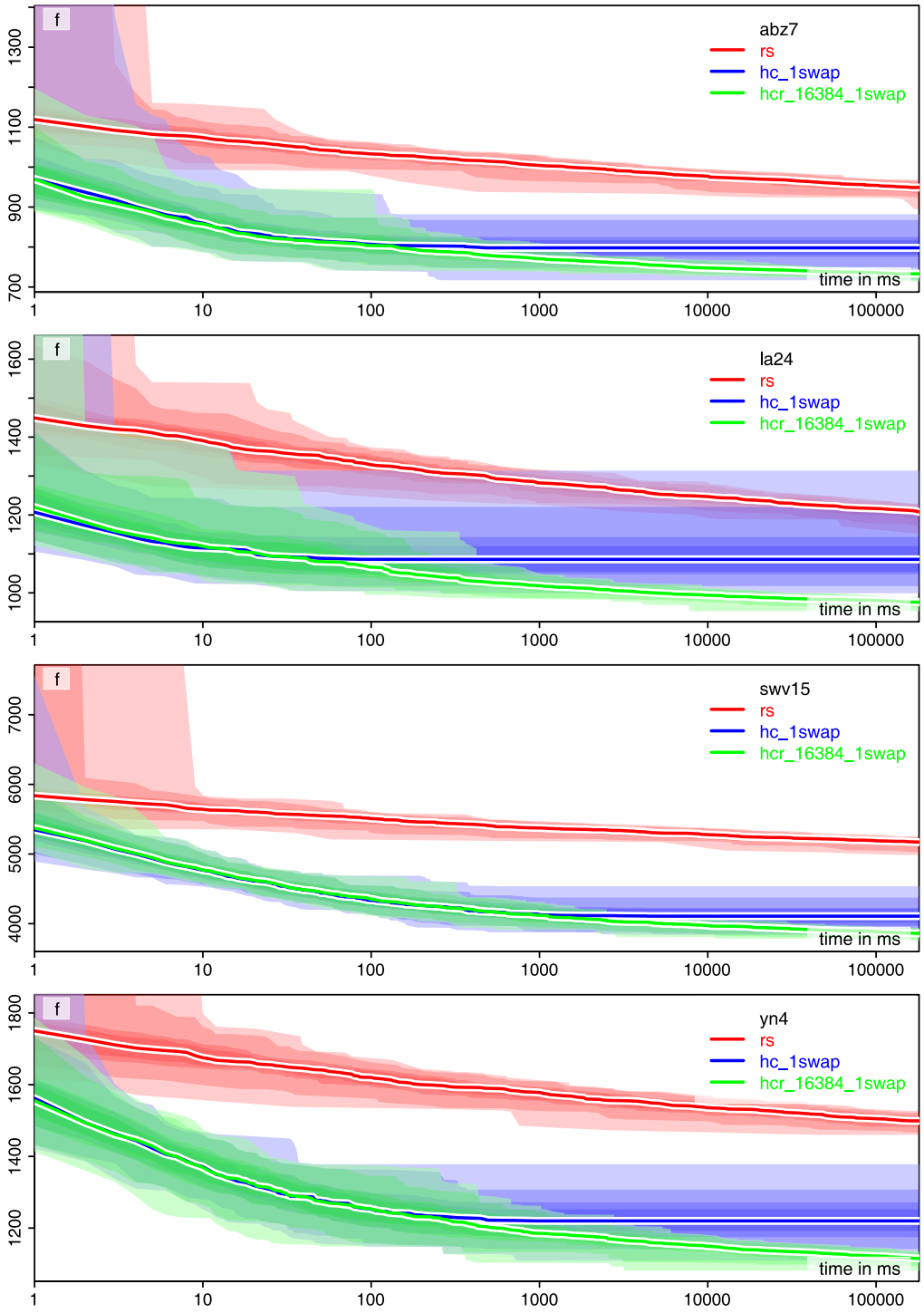

Figure 3.10: The median of the progress of the algorithms `rs`, `hc_1swap`, and `hcr_16384_1swap` over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis). The color of the areas is more intense if more runs fall in a given area.

The median solutions discovered by hcr_16384_1swap, illustrated in Figure 3.9, again show less wasted time. The scheduled jobs again move a bit closer together.

From the progress diagrams plotted in Figure 3.10, we can see that the algorithm version with restart initially behave very similar to the "original" hill climber. Its median quality is almost exactly the same during the first 100 ms. This makes sense, because until its first restart, hcr_16384_1swap is identical to hc_1swap. However, when hc_1swap has converged and stops making improvements, hcr_16384_1swap still continues to make progress.

Of course, our way of finding the right value for the restart parameter L was rather crude. Most likely, the right choice could be determined per instance based on the number m of machines and the number n of jobs. But even with such a coarse way of algorithm configuration, we managed to get rather good results.

3.3.3.4 Drawbacks of the Idea of Restarts

With our restart method, we 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 for its cause.

Basically, a restarted algorithm is still the same algorithm – we just restart it again and again. If there are many more local optima than global optima, every restart will probably end again in a local optimum. If there are many more bad'' local optima thangood'' local optima, every restart will probably end in a "bad'' local optimum. While restarts improve the chance to find better solutions, they cannot solve the intrinsic shortcomings of an algorithm.

Another problem is: With every restart we throw away all accumulated knowledge and information of the current run. Restarts are therefore also somewhat wasteful.

3.3.4 Hill Climbing with a Different Unary Operator

3.3.4.1 Small vs. Large Neighborhoods – and Uniform vs. Non-Uniform Sampling

One of issues limiting the performance of our restarted hill climber is the design of the 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 $m * n$ possible choices when picking the index of the first job to be swapped. 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, because we will only use a second index that points to a different job ID. If we think about the size of the neighborhood spanned by 1swap, we can also ignore equivalent swaps: Exchanging the jobs at indexes $(10, 5)$ and $(5, 10)$, for example, would result in the same

outcome. In total, from any given point in the search space, 1swap may reach $0.5 * (m * n) * [m * (n-1)] = 0.5 * m^2(n^2 - n)$ different other points. Some of these points might still actually encode the same candidate solutions, i.e., identical schedules. In other words, the neighborhood spanned by our 1swap operator equals only a tiny fraction 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. Finding it could, for instance, require swapping three or four jobs – but the hc_1swap algorithm will never find it, because it can only swap two jobs. If the search operator would permit such moves, then even the plain hill climber may discover this better solution.

So let us try to think about how we could define a new unary operator which can access a larger neighborhood. As we should always do, we first consider the extreme cases.

On the one hand, we have 1swap which samples from a relatively small neighborhood. Because the neighborhood is small, the stochastic hill climber will eventually have visited all of the solutions it contains. If none of them is better than the current best solution, it will not be able to depart from it.

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. Then, each point $x \in \mathbb{X}$ would have the whole $\mathbb{X}$ as the neighborhood. Using such a unary operator would turn the hill climber into random sampling (and we do not want that).

From this thought experiment we know that unary operators which indiscriminately sample from very large neighborhoods are probably 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.

Using an operator that creates larger neighborhoods than 1swap, which are still smaller than $\mathbb{X}$ would be one idea. For example, we could always swap three jobs instead of two. The more jobs we swap in each application, the larger the neighborhood gets. Then we will be less likely to get trapped in local optima (as there will be fewer local optima). But we will also make less and less use of the causality property, i.e., the solutions we derive from the current best one will be more and more different from it. Where should we draw the line? How many jobs should we swap?

Well, there is one more aspect of the operators that we did not think about yet. An operator does not just span a neighborhood, but it also defines a *probability distribution* over it. So far, our 1swap unary

operator samples *uniformly* from the neighborhood of its input. In other words, all of the $0.5*m^2(n^2-n)$ new points that it could create in one step have exactly the same probability, the same chance to be chosen.

But we do not need to do it like that. We could construct 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. This operator could have a huge neighborhood – but sample it non-uniformly.

3.3.4.2 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.

Here, the idea 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 where different job-ids are stored, as l must be different from f (*point c* and *d*). We would then swap the jobs at these indices (*points f*, *g*, and *7*). In the case of exactly one iteration of the main loop, this operator behaves 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 theoretically – being able to escape from any local optimum.

3.3.4.3 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 laptop computer in Table 3.5.

Table 3.5: The results of the hill climbers `hc_1swap` and `hc_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	hc_1swap	717	800	798	28	**0s**	**16’978**
		hcr_16384_1swap	**714**	**732**	**733**	**6**	91s	18’423’530
		hc_nswap	724	758	758	17	35s	7’781’762
la24	935	hc_1swap	999	1095	1086	56	**0s**	**6’612**
		hcr_16384_1swap	953	**976**	**976**	**7**	80s	34’437’999
		hc_nswap	**945**	1018	1016	29	25s	9’072’935
swv15	2885	hc_1swap	3837	4108	4108	137	**1s**	**104’598**
		hcr_16384_1swap	3752	**3859**	**3861**	**42**	92s	11’756’497
		hc_nswap	**3602**	3880	3872	112	70s	8’351’112
yn4	929	hc_1swap	1109	1222	1220	48	**0s**	**31’789**

$\mathcal{I}$	$\mathrm{lb}f$	setup	best	mean	med	sd	med(t)	med(FEs)
		hcr_16384_1swap	**1081**	**1115**	**1115**	**11**	91s	14'804'358
		hc_nswap	1095	1162	1160	34	71s	11'016'757

From Table 3.5, we find that hc_nswap performs almost always better than hc_1swap. Only on instance abz7, hc_1swap finds the better best solution. For all other instances, hc_nswap has better best, mean, and median results. It also converges much later and often performs 7 to 15 million function evaluations and consumes 14% to 25% of the three minute budget before it cannot improve anymore. Still, the hill climber hcr_16384_1swap using the 1swap operator with restarts tends to outperform hc_nswap.

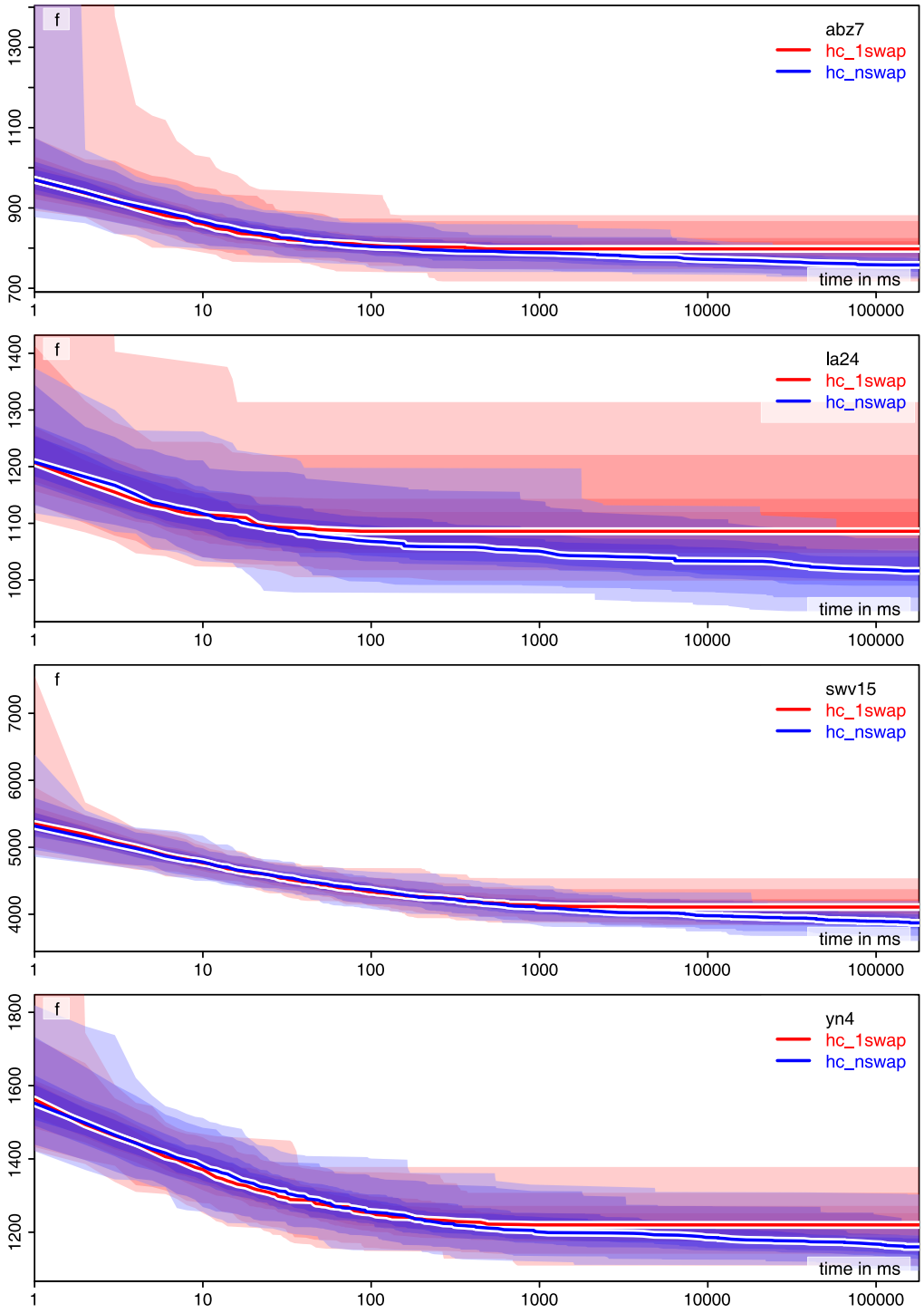

Figure 3.11: Median of the progress of the hill climbers with the 1swap and nswap operators over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis). The color of the areas is more intense if more runs fall in a given area.

Figure 3.11 illustrates the progress of the hill climbers with the 1swap and nswap operators. Initially, both algorithms behave very similar in median. This may be because half of the time, hc_nswap also performs single-swap moves. However, while hc_1swap stops improving even before one second has elapsed, hc_nswap can still improve after ten seconds. Still, these late improvements tend to be small and occur infrequently. It may be that the algorithm arrives in local optima from which it can only escape with complicated multi-swap moves, which are harder to discover by chance.

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 operation to swap
9      int i = random.nextInt(dest.length);
10     int first = dest[i];
11     int last = first; // last stores the job id to "swap in"
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 jobJ = dest[j];
19         if ((last != jobJ) && // don't swap job with itself
20             (hasNext || (first != jobJ))) { // also not at end
21           dest[i] = jobJ; // overwrite job at index i with jobJ
22           i = j; // remember index j: we will overwrite it next
23           last = jobJ; // but not with the same value jobJ...
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 }
```

3.3.5 Combining Bigger Neighborhood with Restarts

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. The fact that hc_nswap improves more and more slowly towards the end of the computational budget means that it could be interesting to try to combine both ideas, restarts and larger neighborhoods.

We plug the nswap operator into the hill climber with restarts and obtain algorithm hcr_L_nswap. We perform the same experiment to find the right setting for the restart limit L as for the hcr_L_1swap algorithm and illustrate the results in Figure 3.12.

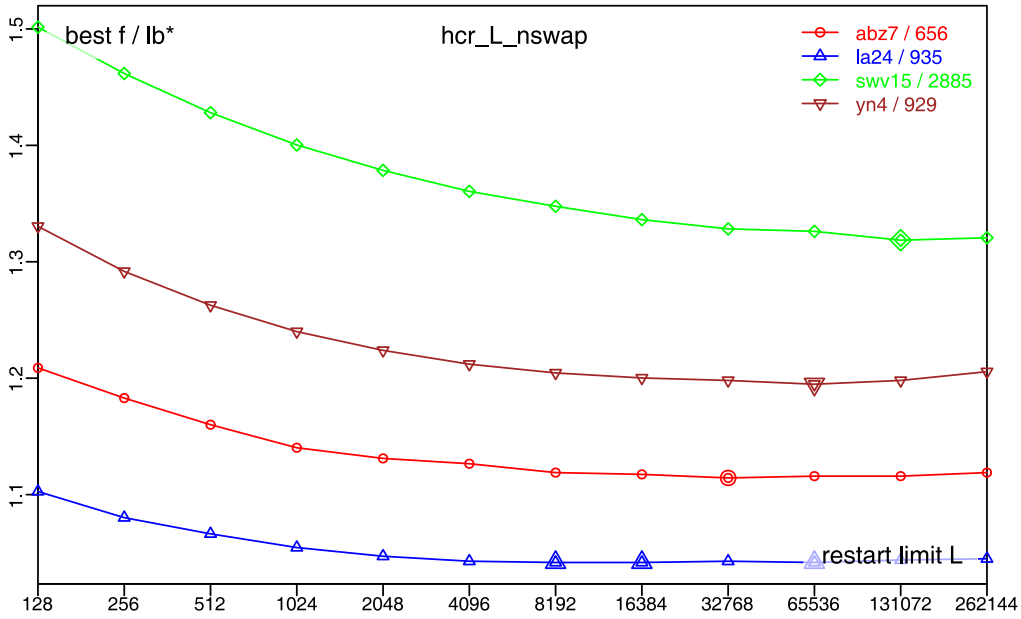

Figure 3.12: The median result quality of the hcr_nswap algorithm, divided by the lower bound $\mathrm{lb}(f)^\star$ from Table 2.2 over different values of the restart limit parameter L. The best values of L on each instance are marked with bold symbols.

The "sweet spot" for the number of unsuccessful FEs before a restart has increased compared to before. This makes sense, because we already know that nswap can keep improving longer.

3.3.5.1 Results on the JSSP

Table 3.6: The results of the hill climber `hcr_L_nswap` with 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	hcr_16384_1swap	714	732	733	**6**	91s	18’423’530
		hc_nswap	724	758	758	17	**35s**	**7’781’762**
		hcr_32768_nswap	**711**	**730**	**731**	6	79s	16’172’407
		hcr_65536_nswap	712	731	732	6	96s	21’189’358
la24	935	hcr_16384_1swap	953	976	976	7	80s	34’437’999
		hc_nswap	945	1018	1016	29	**25s**	**9’072’935**
		hcr_32768_nswap	959	974	975	**6**	92s	38’202’616
		hcr_65536_nswap	**942**	**973**	**974**	8	71s	31’466’420
swv15	2885	hcr_16384_1swap	3752	3859	3861	42	92s	11’756’497
		hc_nswap	**3602**	3880	3872	112	**70s**	**8’351’112**
		hcr_32768_nswap	3703	3830	3832	37	87s	11’288’261
		hcr_65536_nswap	3740	**3818**	**3826**	**35**	89s	10’783’296
yn4	929	hcr_16384_1swap	1081	1115	1115	**11**	91s	14’804’358
		hc_nswap	1095	1162	1160	34	**71s**	**11’016’757**
		hcr_32768_nswap	1081	1114	1113	11	84s	12’742’795
		hcr_65536_nswap	**1068**	**1109**	**1110**	12	78s	18’756’636

From Table 3.6, where we print the results of `hcr_32768_nswap` and `hcr_65536_nswap`, we can find that the algorithm version with restarts performs better in average than the one without. However, it does not always find the best solution, as can be seen on instance `swv15`, where `hc_nswap` finds a schedule of length 3602. The differences between `hcr_16384_1swap` and `hcr_L_nswap`, however, are quite small. If we compare the progress over time of `hcr_16384_1swap` and `hcr_65536_nswap`, then the latter seems to have a slight edge over the former – but only by about half of a percent. This small difference is almost indistinguishable in the progress diagram Figure 3.13.

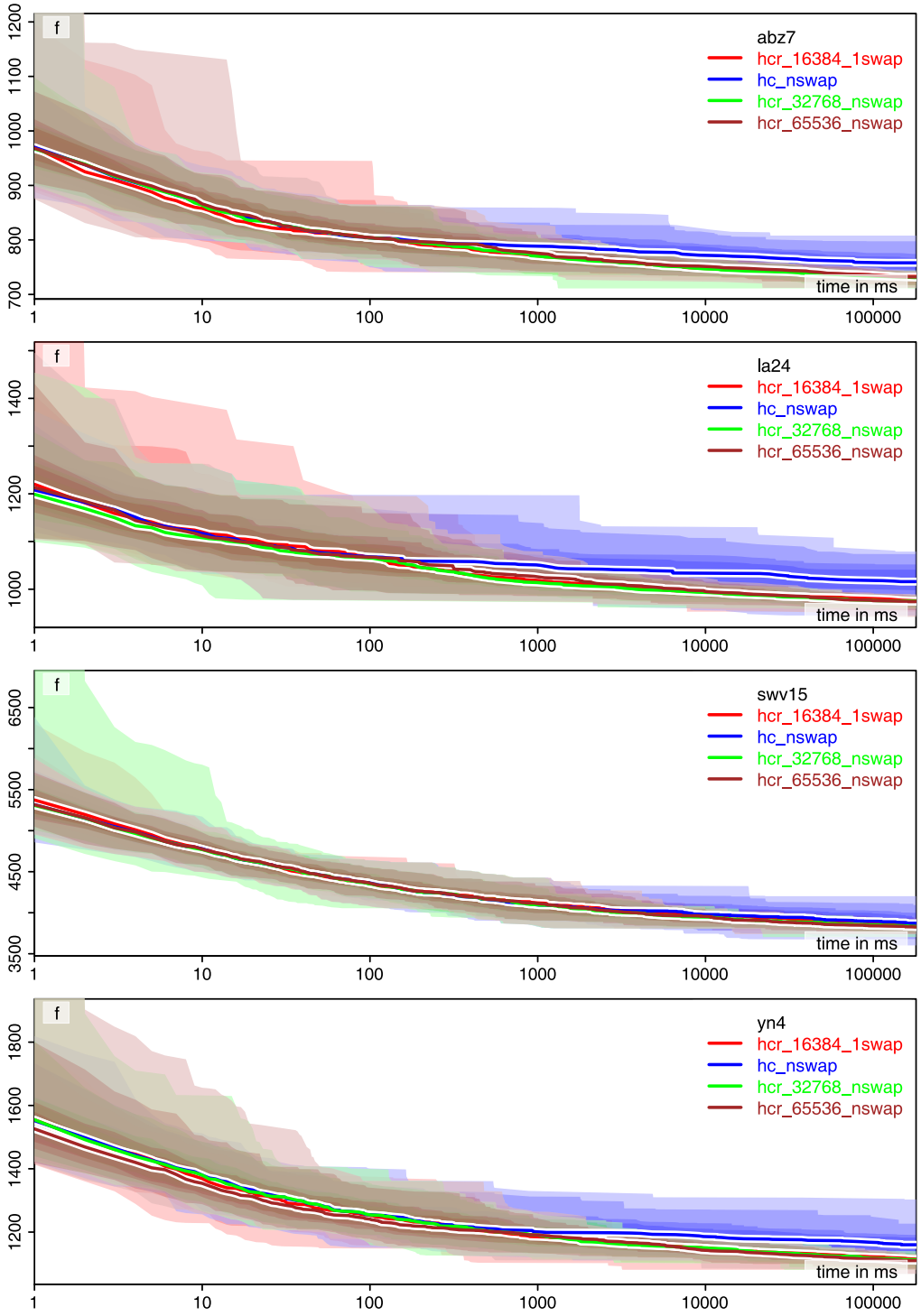

Figure 3.13: Median of the progress of the hill climbers with restarts with the 1swap and nswap operators over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis). The color of the areas is more intense if more runs fall in a given area.

3.3.5.2 Testing for Significance

Still, even small improvements can have a big economical impact. Saving 0.5% of 10'000'000 RMB is still 50'000 RMB. The problem is knowing whether such small improvements are true improvements *or* artifacts of the randomness in the algorithm.

In order to understand the latter 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 another such 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.

So how do we know whether or not the sources of A and B are truly different? Well, we cannot really know for sure. But we can we can make a statement which is wrong with at most a given probability. This is called statistical testing, and we discuss it in detail in Section 4.5. Thus, in order to see whether the observed small performance difference of the `hcr` setups is indeed "real" or just random jitter, we compare their 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.

Table 3.7: The end results of `hcr_16384_1swap` and `hcr_65536_nswap` compared with a Mann-Whitney U test with Bonferroni correction and significance level $\alpha = 0.02$ on the four JSSP instances. The columns indicate the p-values and the verdict (? for insignificant).

Mann-Whitney U $\alpha' = 7.14 \cdot 10^{-4}$	abz7	la24	swv15	yn4
`hcr_16384_1swap` vs. `hcr_65536_nswap`	$3.04 \cdot 10^{-1}$?	$3.57 \cdot 10^{-3}$?	$9.35 \cdot 10^{-11}$ >	$5.19 \cdot 10^{-4}$ >

From Table 3.7 we know that if we would claim "`hcr_16384_1swap` tends to produce results with large makspan than `hcr_65536_nswap` on `abz7`," then, from our experimental data, we can estimate our chance to be wrong to be about 30%. In other words, making that claim would be quite a gamble and we can conclude that here, the differences we observed in the experiment are not statistically significant (marked with ? in the table). However, if we would claim the same for `swv15`, our

chance to be wrong is about $1 \cdot 10^{-10}$, i.e., very small. So on swv15, we find that hcr_65536_nswap very likely performs better.

In summary, Table 3.7 tells us that it seems that the hill climber with restarts using the nswap operator can outperform the one using the 1swap operator on swv15 and yn4. For abz7, there certainly is no significant difference. The p-value of $3.57 \cdot 10^{-3}$ on la24 is fairly small but still above the Bonferroni-corrected $\alpha' = 7.14 \cdot 10^{-4}$, so we cannot make any statement here.

We found that the nswap operator tends to be better then the 1swap operator when plugged into the hill climber. We found that restarts are also beneficial to improve the performance of our hill climber. While we did not find that combining these two methods will yield a convincing improvement, we can at least conclude that preferring hcr_65536_nswap over hcr_16384_1swap seems reasonable. It *might* sometimes give better performance while it is unlikely to perform worse.

3.3.6 Summary

In this section, we have learned about our first "reasonable" optimization method. The stochastic hill climbing algorithm always remembers the best-so-far point in the search space. In each step, it applies the unary operator to obtain a similar but slightly different point. If it is better, then it becomes the new best-so-far point. Otherwise, it is forgotten.

The performance of hill climbing depends very much on the unary search operator. If the operator samples from a very small neighborhood only, like our 1swap operator does, then the hill climber might quickly get trapped in a local optimum. A local optimum here is a point in the search space which is surrounded by a neighborhood that does not contain any better solution. If this is the case, the two conditions for doing efficient restarts may be fulfilled: quick convergence and variance of result quality.

The question when to restart then arises, as we usually cannot find out if we are actually trapped in a local optimum or whether the improving move (application of the unary operator) just has not been discovered yet. The most primitive solution is to simply set a limit L for the maximum number of moves without improvement that are permitted.

Our hcr_L_1swap was born. We configured L in a small experiment and found that $L = 16384$ seemed to be reasonable. The setup hcr_16384_1swap performed much better than hc_1swap. It should be noted that our experiment used for configuration was not very thorough, but it should suffice at this stage. We can also note that it showed that different settings of L are better for different instances. This is probably related to the corresponding search space size – but we will not investigate this any further here.

A second idea to improve the hill climber was to use a unary operator spanning a larger neighborhood, but which still most often sampled solutions similar to current one. The nswap operator gave better

results than than the `1swap` operator in the basic hill climber. The take-away message is that different search operators may (well, obviously) deliver different performance and thus, testing some different operators can always be a good idea.

Finally, we tried to combine our two improvements, restarts and better operator, into the `hcr_L_nswap` algorithm. Here we learned the lesson that performance improvements do not necessarily add up. If we have a method that can deliver an improvement of 10% of solution quality and combine it with another one delivering 15%, we may not get an overall 25% improvement. Indeed, our `hcr_65536_nswap` algorithm only performed a little bit better than `hcr_16384_1swap`.

From this chapter, we also learned one more lesson: Many optimization algorithms have parameters. Our hill climber had two: the unary operator and the restart limit L. Configuring these parameters well can lead to significant improvements.

3.4 Evolutionary Algorithms

We now already have one 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 single 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 many different kinds of optimization problems. Now we will look at a third concept to prevent premature convergence: Instead of just remembering and utilizing only one single point from the search space in each iteration of our algorithm, we will now work on an array of points!

3.4.1 Evolutionary Algorithm without Recombination

Today, there exists a wide variety of different Evolutionary Algorithms (EAs) [17,41,54,84,145,146,188,205]. We will here discuss a 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] For now, we will discuss EAs in a form without recombination. Wait for the binary recombination operator until Section 3.4.3.

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 $\mu + \lambda$ of instances of I.
3. For index i ranging from 0 to $\mu + \lambda - 1$ do
   a. Create a random point from the search space using the nullary search operator and store it 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 in ascending order according to the objective values, i.e., such that the records r with better associated objective value $r.z$ are located at smaller indices.
   e. Shuffle the first μ elements of P randomly.
   f. Set the source index $p = -1$.
   g. For index i ranging from μ to $\mu + \lambda - 1$ do
      i. Set the source index p to $p = (p + 1) \bmod \mu$.
      ii. Apply unary search operator to the point stored at index p and store result at index i, i.e., set $P_i.x = \mathrm{searchOp}_1(P_p.x)$.
      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 (i.e., the best-ever encountered solution) to the user.

This algorithm is implemented in Listing 3.13. There, we make use of instances of the utility class `Record<X>`, which holds one point `x` in the search space along with their corresponding objective values stored in the field `quality`.

Basically, the algorithm 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 with newly sampled points.

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. [17,26,36,146,205]

Selection can be done by sorting the array P (*point 4d*). 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 selected points in an Evolutionary Algorithm are called *parents*.

Definition 28. The points in an Evolutionary Algorithm that are sampled from the neighborhood of the parents by applying search operators are called *offspring*.

Definition 29. 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. The index p in *steps 4f to 4g* 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 perform a modulo division with μ in *step 4g.i*, i.e., set p to the remainder of the division with μ, which makes sure that p will be in $0 \dots (\mu - 1)$.

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 4e*). This does not change the fact that they have been selected.

Since our algorithm will never prefer a worse solution over a better one, it will also never lose the best-so-far solution. We therefore can simply pick the best element from the population once the algorithm has converged. It should be mentioned that there are selection methods in EAs which might reject the best-so-far solution. In this case, we would need to remember it in a special variable like we did in case of the hill climber with restarts. Here, we do not consider such methods, as we want to investigate a plain EA.

3.4.1.2 The Right Setup

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. The question arises which values for μ and λ are reasonable. Without

investigating whether this is the best idea, let us set $\mu = \lambda$ here, so we only have two parameters to worry about: μ and the unary search operator. We already have two unary search operators. Let us call our algorithms of this type `ea_mu_unary`, where `mu` will stand for the value of μ and λ and `unary` can be either `1swap` or `nswap`. We no therefore do a similar experiment as in Section 3.3.3.2 in order to find the right parameters.

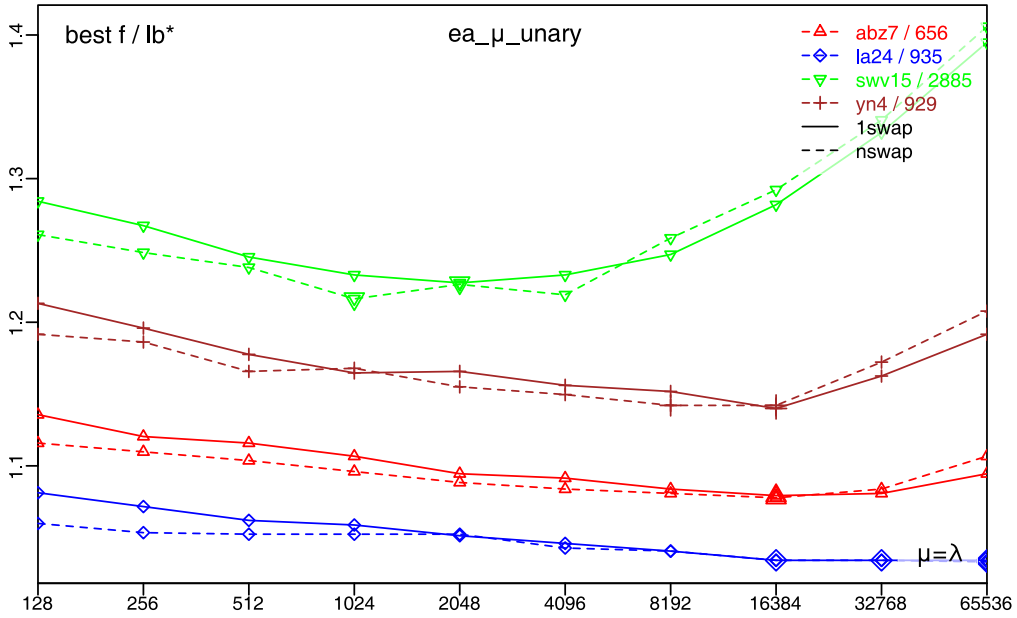

Figure 3.14: The median result quality of the `ea_mu_unary` algorithm, divided by the lower bound $\mathrm{lb}(f)^\star$ from Table 2.2 over different values of the population size parameter $\mu = \lambda$ and the two unary search operators `1swap` and `nswap`. The best values of μ for each operator and instance are marked with bold symbols.

In Figure 3.14, we illustrate this experiment. Regarding μ and λ, we observe the same situation as with the restarts parameters hill climber. There is a "sweet spot" somewhere between small and large population sizes. For small values of μ, the algorithm may end up in a local optimum, whereas for large values, it may not be able to perform sufficiently many generations to arrive at a good solution. Nevertheless, compared to the hill climber with restart and with the exception for instance `swv15`, the EA performs quite stable: There are only relatively little differences in result quality for both unary operators and over many scales of μ. This generally a nice feature, as we would like to have algorithms that are not too sensitive to parameter settings.

The setting $\mu = \lambda = 16'384$ seems to work well for instances `abz7`, `la25`, and `yn4`. Interestingly, instance `swv15` behaves different: here, the setting $\mu = \lambda = 1024$ works best. It is quite common in optimization that different problem instances may require different setups to achieve the best

performance. Here we see this quite pronounced. This is generally a feature that we do not like. We would ideally like to have algorithms where a good parameter setting performs well on many instances as opposed to such where each instance requires a totally different setup. Luckily, this here only concerns one of our example problem instances.

Regarding the choice of the unary search operator: With the exception of problem `swv15`, both operators provide the same median result quality. In the other setups, if one of the two is better, it is most of the time `nswap`.

Therefore, we will consider the two setups `ea_16384_nswap` and `ea_1024_nswap` when evaluating the performance of our Evolutionary Algorithms.

3.4.1.3 Results on the JSSP

Let us now compare the results of the best two EA setups with those that we obtained with the hill climber.

Table 3.8: The results of the Evolutionary Algorithm `ea_mu_nswap` without recombination in comparison with the best hill climber `hc_nswap` and the best hill climber with restarts `hcr_65536_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	hc_nswap	724	758	758	17	35s	7'781'762
		hcr_65536_nswap	712	731	732	**6**	96s	21'189'358
		ea_16384_nswap	**691**	**707**	**707**	8	151s	25'293'859
		ea_1024_nswap	696	719	719	9	**14s**	**4'703'601**
la24	935	hc_nswap	945	1018	1016	29	25s	9'072'935
		hcr_65536_nswap	942	973	974	**8**	71s	31'466'420
		ea_16384_nswap	945	**968**	**967**	12	39s	10'161'119
		ea_1024_nswap	**941**	983	984	19	**2s**	**971'842**
swv15	2885	hc_nswap	3602	3880	3872	112	**70s**	**8'351'112**
		hcr_65536_nswap	3740	3818	3826	**35**	89s	10'783'296

$\mathcal{I}$	$\mathrm{lb}f$	setup	best	mean	med	sd	med(t)	med(FEs)
		ea_16384_nswap	3577	3723	3728	50	178s	18'897'833
		ea_1024_nswap	**3375**	**3525**	**3509**	85	87s	16'979'178
yn4	929	hc_nswap	1095	1162	1160	34	71s	11'016'757
		hcr_65536_nswap	1068	1109	1110	**12**	78s	18'756'636
		ea_16384_nswap	**1022**	**1063**	**1061**	16	168s	26'699'633
		ea_1024_nswap	1035	1083	1085	20	**31s**	**4'656'472**

Table 3.8 shows us that we can improve the best, mean, and median solution quality that we can get within three minutes of runtime when using our either of the two 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 and has a better best solution than ea_16384_nswap. Here, the best solution encountered now has a makespan which is only 0.7% longer than what is theoretically possible. Nevertheless, we find quite a tangible improvement in case swv15 on ea_1024_nswap.

Our ea_16384_nswap outperforms the four Evolutionary Algorithms from [122] both in terms of mean and best result quality on abz7 and la24. It does the same for HIMGA-Mutation, the worst of the four HIMGA variants introduced in [130], for abz7, la24, and yn4. It obtains better results than the PABC from [175] on swv15. On la24, both in terms of mean and best result, it outperforms also all six EAs from [2], both variants of the EA in [157], and the LSGA from [158]. The best solution quality for abz7 delivered by ea_16384_nswap is better than the best result found by the old Fast Simulated Annealing algorithm which was improved in [4].

The Gantt charts of the median solutions of ea_16384_nswap are illustrated in Figure 3.15. They appear only a bit denser than those in Figure 3.9.

More interesting are the progress diagrams of the ea_16384_nswap, ea_1024_nswap, hcr_65536_nswap, and hc_nswap algorithms given in Figure 3.16. Here we find big visual differences between the way the EAs and hill climbers proceed. The EAs spend the first 10 to 1000 ms to discover some basins of attraction of local optima before speeding up. The larger the population, the longer it takes them until this happens: The difference between ea_16384_nswap, is very obvious ea_1024_nswap in this respect. 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 performance, 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.

We also notice that – for the first time in our endeavor to solve the JSSP – runtime is the limiting factor. If we would have 20 minutes instead of three, then we could not expect much improvement from the hill climbers. Even with restarts, they already improve very slowly towards the end of the computational budget. Tuning their parameter, e.g., increasing the time until a restart is performance, would probably not help then either. However, we can clearly see that `ea_16384_nswap` has not fully converged on neither `abz7`, `swv15`, nor on `yn4` after the three minutes. Its median curve is still clearly pointing downwards. And even if it had converged: If we had a larger computational budget, we could increase the population size. The reason why the performance for larger populations in Figure 3.14 gets worse is very likely the limited time budget.

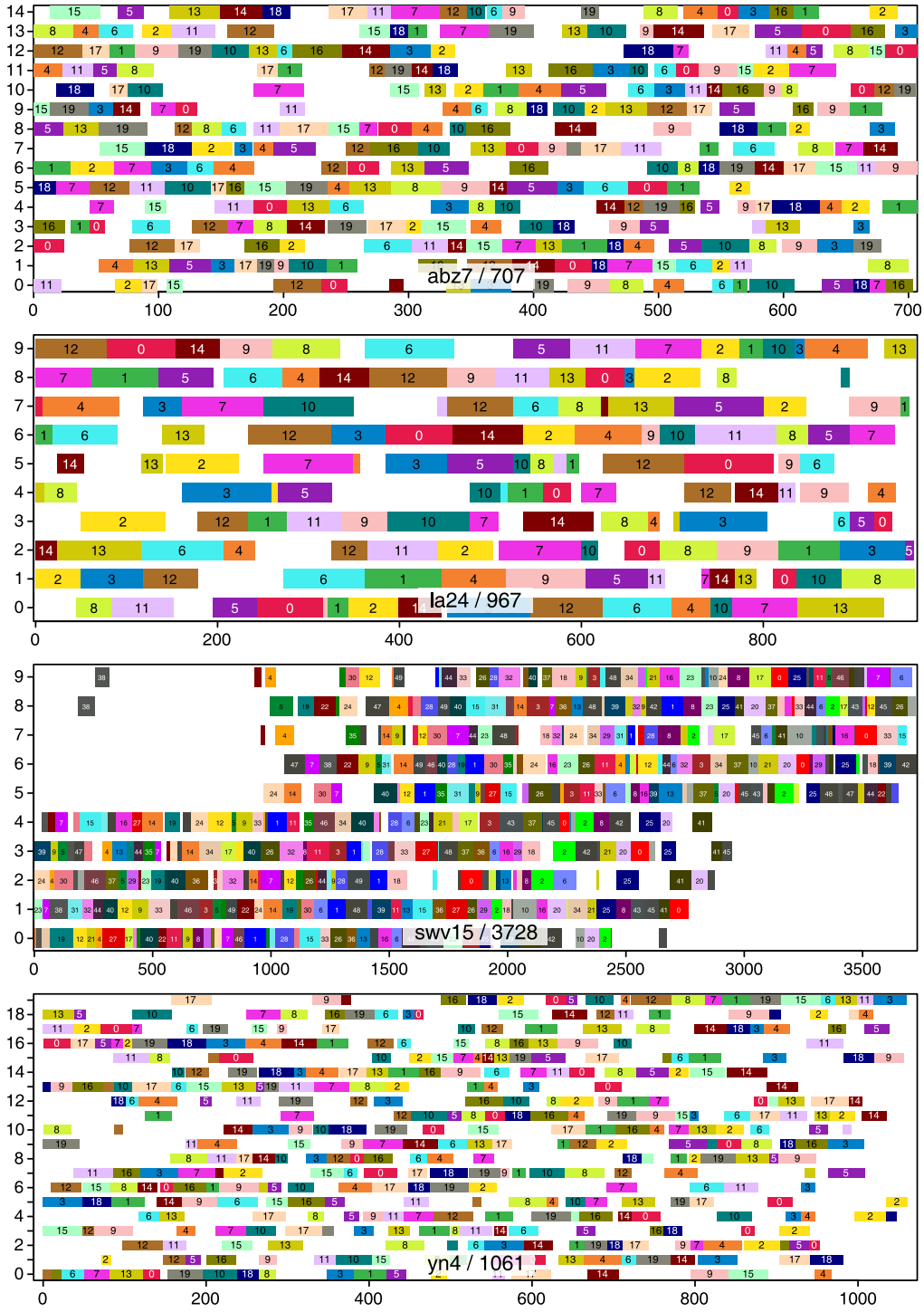

Figure 3.15: The Gantt charts of the median solutions obtained by the ea_16384_nswap setup. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

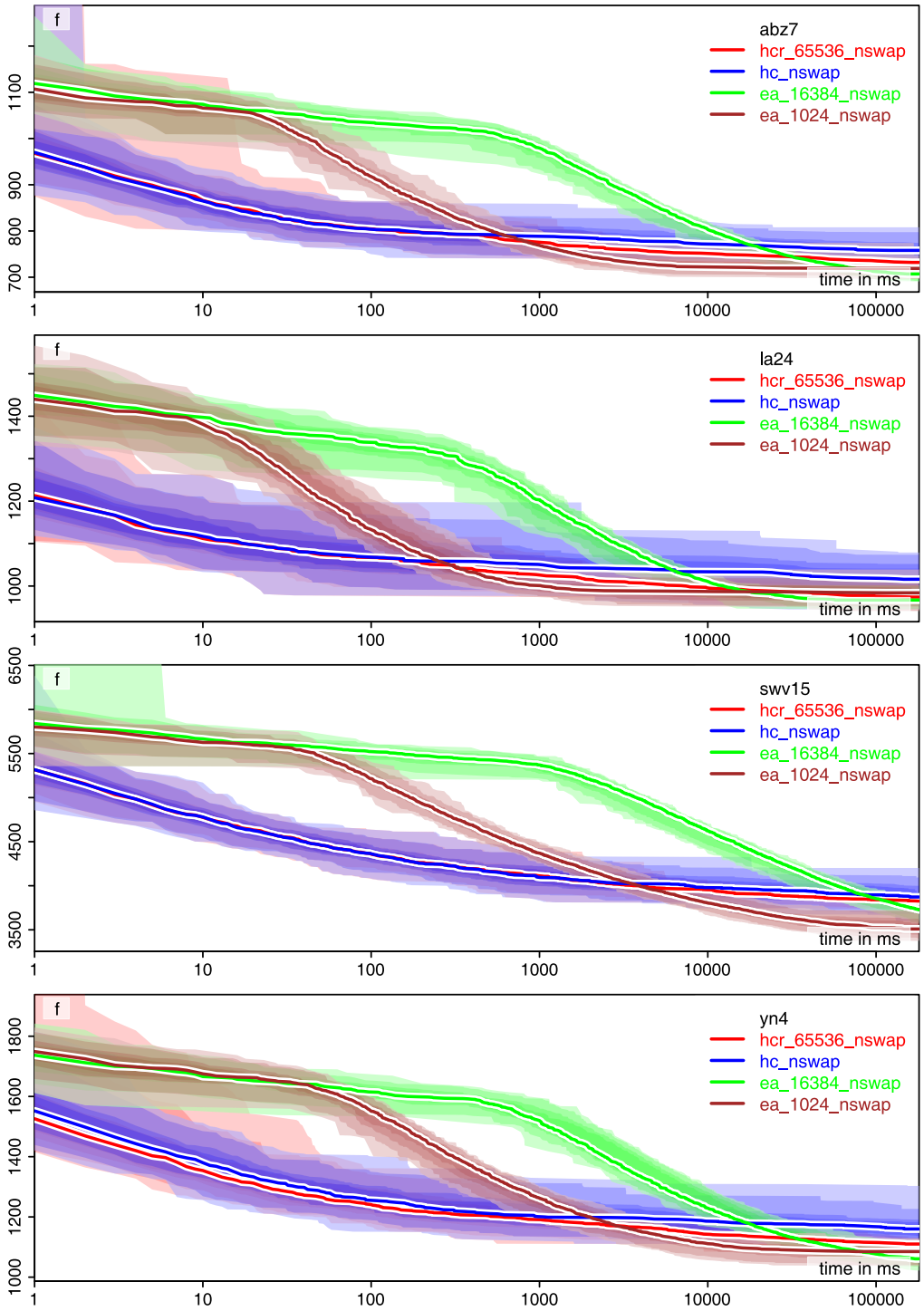

Figure 3.16: The median of the progress of the `ea_16384_nswap`, `ea_1024_nswap`, `hcr_65536_nswap`, and `hc_nswap` algorithms over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis). The color of the areas is more intense if more runs fall in a given area.

3.4.1.4 Exploration versus Exploitation

Naturally, when discussing EAs, we may ask why the population is helpful in the search. While we now have some anecdotal evidence that this is the case, we may also think about this more philosophically. Let us therefore do a thought experiment. If we would set $\mu = 1$ and $\lambda = 1$, then the EA would always remember the best solution we had so far and, in each generation, derive one new solution from it. If it is better, it will replace the remembered solution. Such an EA is actually a hill climber.

Now imagine what would happen if we would set μ to infinity instead. We would not even complete one single generation. Instead, if $\mu \to \infty$, it would also take infinitely long to finish creating the first population of random solutions. This does not even require infinity μ – μ just needs to be large enough so that the complete computational budget (in our case, three minutes) is consumed before creating the initial, random candidate solutions is completed. In other words, the EA would then equal random sampling.

The parameter μ basically allows us to "tune" between these two behaviors [214]! If we pick it small, our algorithm becomes more "greedy". It will spend more time investigating (*exploiting*) the neighborhood of the current best solutions. 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.

The question on which of the two to focus is known as the dilemma of "Exploration versus Exploitation" [48,70,205,208,216]. To make matters worse, theorists have proofed that there are scenarios where only a small population can perform well, while there are other scenarios where only a large population works well [45]. In other words, if we apply an EA, we always need to do at least some rudimentary tuning of μ and λ.

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

```
public class EA<X, Y> extends Metaheuristic2<X, Y> {
  public void solve(IBlackBoxProcess<X, Y> process) {
// omitted: initialize local variables random, searchSpace, and
// the array P of length mu+lambda
// first generation: fill population with random solutions
    for (int i = P.length; (--i) >= 0;) {
      X x = searchSpace.create(); // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
    }

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

// overwrite 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
        }

        Record<X> dest = P[index];
        p1 = (p1 + 1) % this.mu; // step the parent 1 index
        Record<X> sel = P[p1];
// create modified copy of parent using unary operator
          this.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 `ea_16384_nswap` (Figure 3.15) and `hcr_16384_1swap` (Figure 3.9), you can see some good solutions for the same problem instances. These solutions 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 [53,109]. If we are lucky and that works, then ideally such good characteristics could aggregate over time [84,147].

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 operation should be stored in x' to $i = 0$.
3. Repeat
    a. Randomly choose one of the input points $x1$ or $x2$ with equal probability as source x.
    b. Select the first (at the lowest index) operation 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 operations 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 by 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 jobs to machines according to the order in which their IDs appear. It is

a natural idea to design a binary operator that works in a similar way. Our sequence recombination 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 operation, 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 operations in exactly the order in which they appear in $x2$.

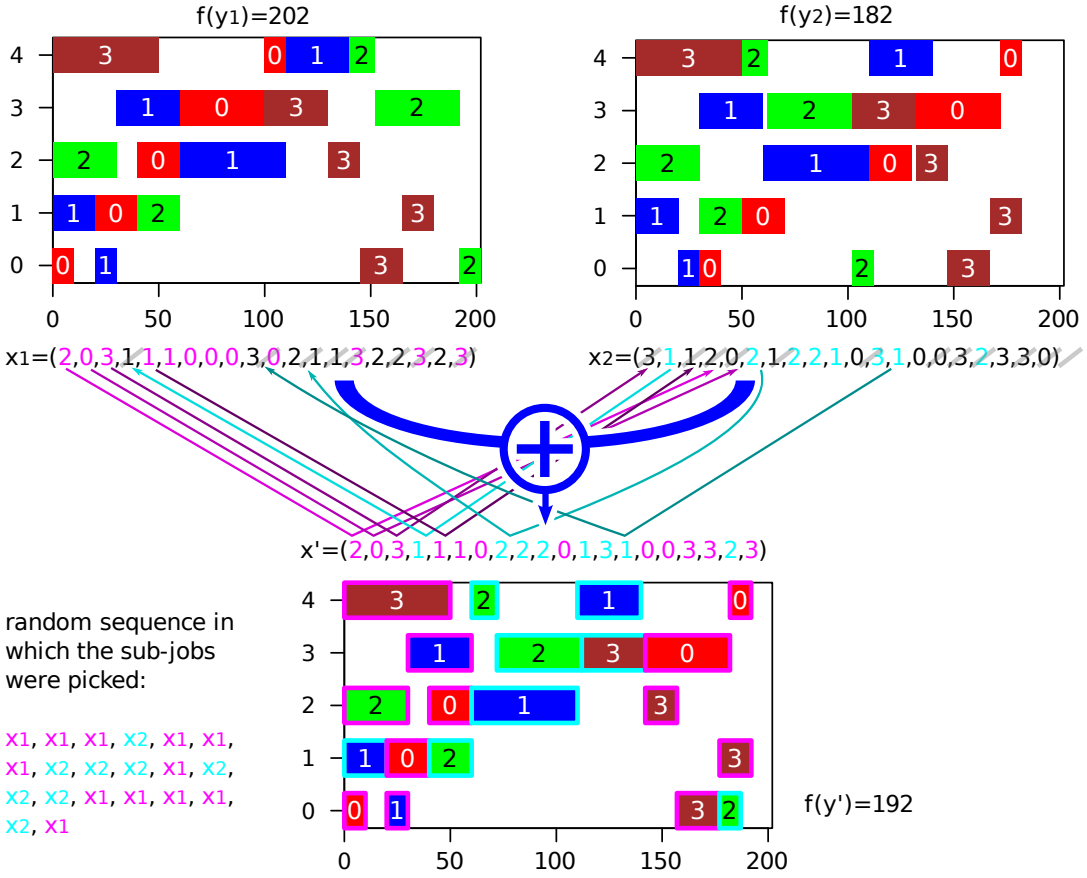

Figure 3.17: An example application of our sequence recombination operator to two points $x1$ and $x2$ in the search space of the demo instance, resulting in a new point x'. We mark the selected job IDs with pink and cyan color, while crossing out those IDs which were not chosen because of their received 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 3.17. 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 operation for the new point x'. The first job ID in $x1$ is 2, which is placed as first operation 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 operation. 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 operation 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 operation, $x2$ is chosen. The first unmarked operation 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)

```
public class JSSPBinaryOperatorSequence
    implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1,
      int[] dest, Random random) {
// omitted: initialization of arrays doneX0 and doneX1 (that
// remember the already-assigned operations from x0 and x1) of
// length=m*n to all false; and indices desti, x0i, x10 to 0
    for (;;) { // repeat until dest is filled, i.e., desti=length
// randomly chose a source point and pick next operation from it
      int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
      dest[desti++] = add; // we picked a operation and added it
      if (desti >= length) { // if desti==length, we are finished
        return; // in this case, desti is filled and we can exit
      }

      for (int i = x0i;; i++) { // mark operation as done in x0
        if ((x0[i] == add) && (!doneX0[i])) { // find added job
          doneX0[i] = true;// found it and marked it
          break; // quit operation finding loop
        }
      }
      while (doneX0[x0i]) { // now we move the index x0i to the
        x0i++; // next, not-yet completed operation in x0
      }

      for (int i = x1i;; i++) { // mark operation as done in x1
        if ((x1[i] == add) && (!doneX1[i])) { // find added job
          doneX1[i] = true; // found it and marked it
          break; // quit operation finding loop
        }
      }
      while (doneX1[x1i]) { // now we move the index x1i to the
        x1i++; // next, not-yet completed operation in x0
      }
    } // loop back to main loop and to add next operation
  } // end of function
}
```

3.4.3 Evolutionary Algorithm with Recombination

We now want to utilize this new operator in our EA. The algorithm now has two ways to create new offspring solutions: either via the unary operator (mutation, in EA-speak) or via the binary operator (recombination in EA-speak). We modify the original EA as follows.

3.4.3.1 The Algorithm (with Recombination)

We introduce a new parameter $cr \in [0,1]$, 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 otherwise apply the unary operator, i.e., 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 $\mu+\lambda$ of instances of I.
3. For index i ranging from 0 to $\mu+\lambda-1$ do
   a. Create a random point from the search space using the nullary search operator and store it 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 in ascending order according to the objective values, i.e., such that the records r with better associated objective value $r.z$ are located at smaller indices.
   e. Shuffle the first μ elements of P randomly.
   f. Set the first source index $p1 = -1$.
   g. For index i ranging from μ to $\mu+\lambda-1$ do
      i. Set the first source index $p1$ to $p1 = (p1+1) \bmod \mu$.
      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 a second index $p2$ from $0 \ldots (\mu-1)$ such that $p2 \neq p1$. B. Apply binary search operator to the points stored at index $p1$ and $p2$ and store result at index i, i.e., set $P_i.x = \mathrm{searchOp}_2(P_{p1}.x, P_{p2}.x)$.
      iv. otherwise to *step 4g.iii*, i.e., if $c \geq cr$, then we apply the unary operator: C. Apply unary search operator to the point stored at index $p1$ and store result at index i, i.e., set $P_i.x = \mathrm{searchOp}_1(P_{p1}.x)$.

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 variant 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 4g.iii.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 The Right Setup

Unfortunately, with $cr \in [0, 1]$, a new algorithm parameter has emerged. It is not really clear whether a large or a small crossover rate is good. We already tested $cr = 0$: The EA without recombination. Similar to our other small tuning experiments, let us compare the performance of different settings of cr. We investigate the crossover rates $cr \in \{0, 0.05, 0.3, 0.98\}$. We will stick with `nswap` as unary operator and keep $\mu = \lambda$.

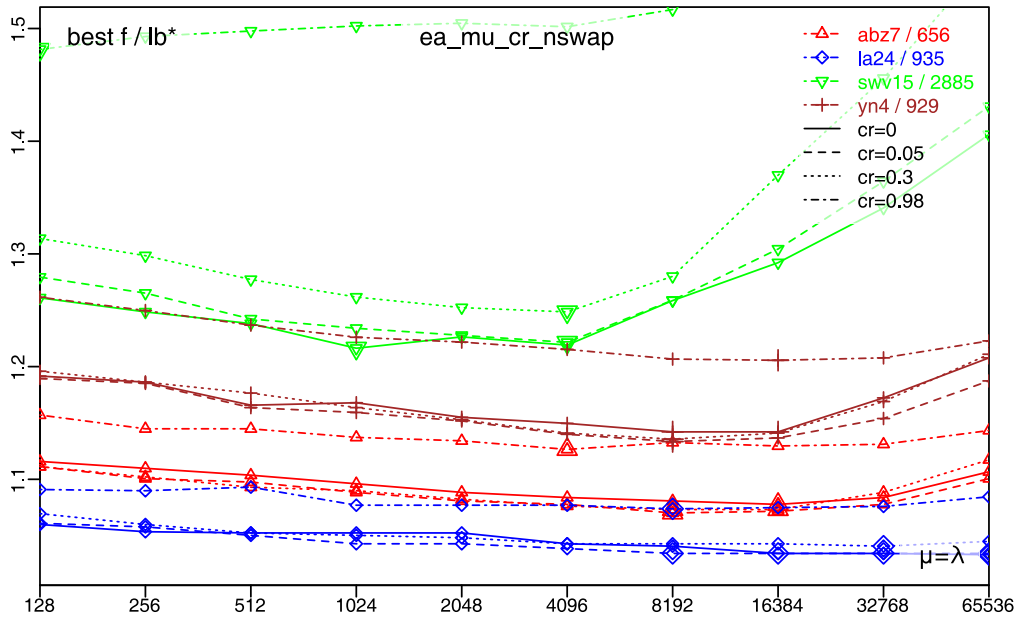

Figure 3.18: The median result quality of the `ea_mu_cr_nswap` algorithm, divided by the lower bound $\mathrm{lb}(f)^\star$ from Table 2.2 over different values of the population size parameter $\mu = \lambda$ and the crossover rates in $\{0, 0.05, 0.3, 0.98\}$. The best values of μ for each crossover rate and instance are marked with bold symbols.

From Figure 3.18, we immediately find that the large crossover rate $cr = 0.98$ is performing much worse than using no crossover at all ($cr = 0$) on all instances. Smaller rates $cr \in \{0.05, 0.3\}$ tend to be sometimes better and sometimes worse than $cr = 0$, but there is no big improvement. On instance swv15, the binary operator does not really help. On instance la24, $cr = 0.05$ performs best on mid-sized populations, while there is no distinguishable difference between $cr = 0.05$ and $cr = 0$ for large populations. On abz7 and yn4, $cr = 0.05$ always seems to be a good choice. On these three instances, the population size $\mu = \lambda = 8192$ in combination with $cr = 0.05$ looks promising. We will call this setup ea_8192_5%_nswap in the following, where 5% stands for the crossover rate of 0.05 using the binary sequence operator.

3.4.3.3 Results on the JSSP

We can now investigate whether our results have somewhat improved.

Table 3.9: The results of the Evolutionary Algorithm ea_8192_5%_nswap in comparison two the same population size without recombination (ea_8192_nswap) and the two EAs from Table 3.8. 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	ea_16384_nswap	691	707	707	**8**	151s	25’293’859
		ea_1024_nswap	696	719	719	9	**14s**	**4’703’601**
		ea_8192_nswap	687	709	709	9	75s	13’347’232
		ea_8192_5%_nswap	**684**	**703**	**702**	8	54s	10’688’314
la24	935	ea_16384_nswap	945	968	**967**	12	39s	10’161’119
		ea_1024_nswap	**941**	983	984	19	**2s**	**971’842**
		ea_8192_nswap	**941**	973	973	15	16s	4’923’721
		ea_8192_5%_nswap	943	**967**	**967**	**11**	18s	4’990’002
swv15	2885	ea_16384_nswap	3577	3723	3728	**50**	178s	18’897’833
		ea_1024_nswap	**3375**	**3525**	**3509**	85	**87s**	**16’979’178**
		ea_8192_nswap	3497	3630	3631	54	178s	18’863’017
		ea_8192_5%_nswap	3498	3631	3632	65	178s	17’747’983

$\mathcal{I}$	$\mathrm{lb} f$	setup	best	mean	med	sd	med(t)	med(FEs)
yn4	929	ea_16384_nswap	**1022**	1063	1061	**16**	168s	26'699'633
		ea_1024_nswap	1035	1083	1085	20	**31s**	**4'656'472**
		ea_8192_nswap	1027	1061	1061	17	138s	17'882'160
		ea_8192_5%_nswap	1026	**1056**	**1053**	17	114s	13'206'552

The results in Table 3.9 show that a moderate crossover rate of 0.05 can indeed improve our algorithm's performance – a little bit. For swv15, we already know that recombination was not helpful and this is confirmed in the table. On the three other instances, ea_8192_5%_nswap has better mean and median results than ea_1024_nswap, ea_8192_nswap, and ea_16384_nswap. On abz7, it can also slightly improve on the best result we got so far.

Interestingly, on swv15, the ea_1024_nswap stops improving in median after about 87 seconds, whereas all other EAs keep finding improvements even very close to the end of the 180 seconds budget. This probably means that they would also perform better than ea_1024_nswap if only we had more time. 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 would be 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*. We therefore apply the same statistical approach as already used in Section 3.3.5.2 and later discussed in detail in Section 4.5.

Table 3.10: The end results of ea_8192_nswap and ea_8192_5%_nswap compared with a Mann-Whitney U test with Bonferroni correction and significance level $\alpha = 0.02$ on the four JSSP instances. The columns indicate the p-values and the verdict (? for insignificant).

Mann-Whitney U $\alpha' = 7.14 \cdot 10^{-4}$	abz7	la24	swv15	yn4
ea_8192_nswap vs. ea_8192_5%_nswap	$2.32 \cdot 10^{-6}$ >	$2.73 \cdot 10^{-3}$?	$9.78 \cdot 10^{-1}$?	$4.32 \cdot 10^{-2}$?

In Table 3.10, we find that that EA using the binary sequence operator to generate 5% of the offspring solutions leads to significantly better results on abz7. It never performs significantly worse, not even on swv15, and the p-values are below α but above α' on the other two instances. Overall, this is not a very convincing result. Not enough for us to claim that our particular recombination operator is a very good invention – but enough to convince us that using it may sometimes be good and won't hurt too much otherwise.

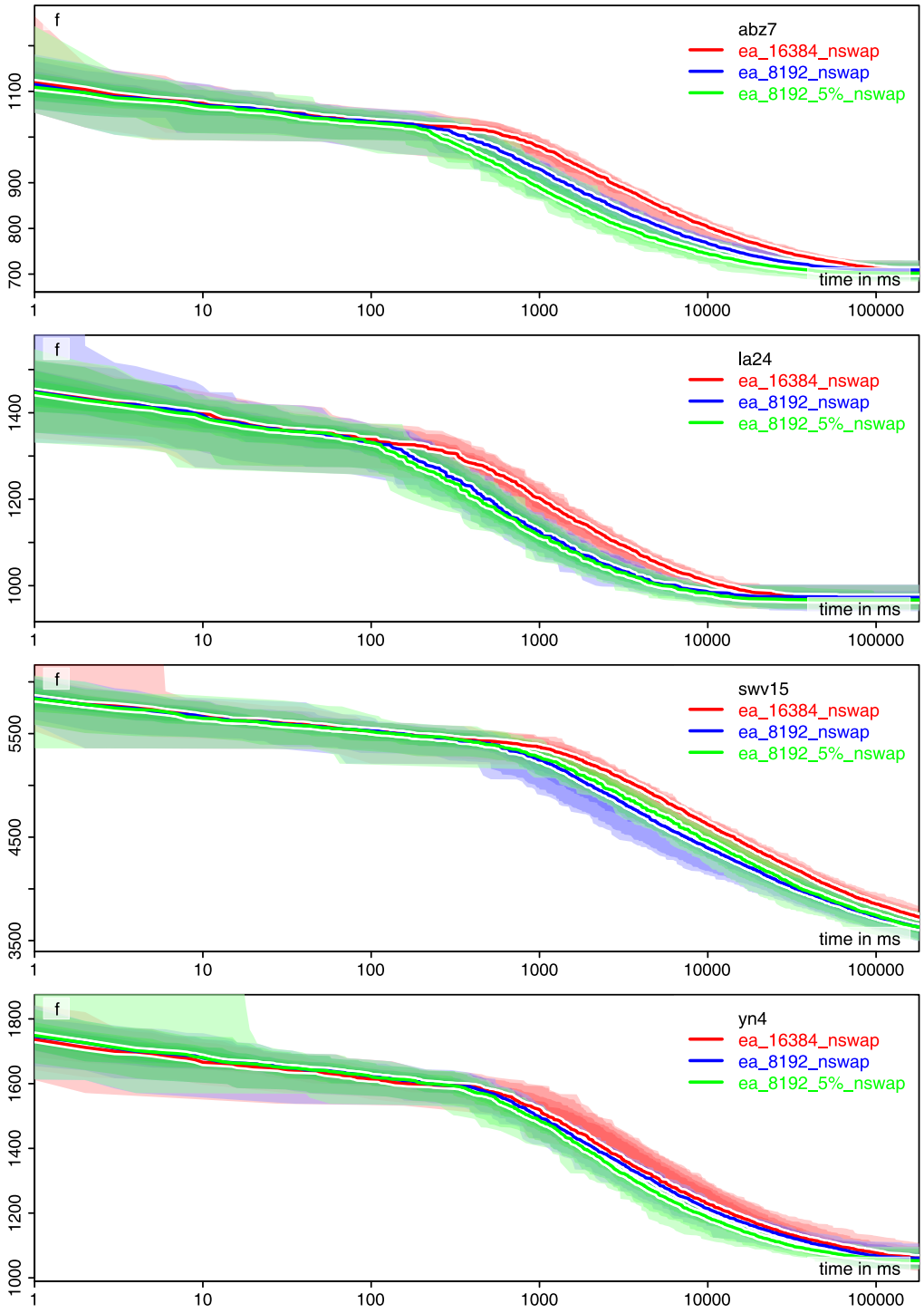

Figure 3.19: The median of the progress of the `ea_8192_5%_nswap`, `ea_8192_nswap`, and `ea_16384_nswap` algorithms over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis). The color of the areas is more intense if more runs fall in a given area.

If we look at Figure 3.19, we can confirm that using the binary `sequence` operator at the low 5% rate does make some difference in how the median solution quality over time changes, although only a small one. On `abz7`, `ea_8192_5%_nswap` improves faster than the setup without recombination. On `la24` and `yn4`, there may be a small advantage during the phase when the algorithm improves the fastest, but this could also be caused by the randomness of the search. On `swv15`, there is a similarly small disadvantage of `ea_8192_5%_nswap`.

In summary, it seems that using our binary operator is reasonable. Different from what we may have hoped for (and which would have been very nice for this book...), it does not improve the results by much. We now could try to design a different recombination operator in the hope to get better results, similar to what we did with the unary operator by moving from `1swap` to `nswap`. We will not do this here – the interested reader is invited to do that by herself as an exercise.

As the end of this section, let me point out that binary search operators are a hot and important research topic right now. On one of the most well-known classical optimization problems, the Traveling Salesman Problem mentioned already back in the introduction in Section 1.1.2, they are part of the most efficient algorithms [153,178,197,219]. It also has theoretically been proven that a binary operator can speed-up optimization on some problems [61].

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

```
public class EA<X, Y> extends Metaheuristic2<X, Y> {
  public void solve(IBlackBoxProcess<X, Y> process) {
// omitted: initialize local variables random, searchSpace, and
// array P of length mu+lambda
// first generation: fill population with random solutions
    for (int i = P.length; (--i) >= 0;) {
      X x = searchSpace.create(); // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
    }

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

// overwrite 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
        }

        Record<X> dest = P[index];
        p1 = (p1 + 1) % this.mu; // step the parent 1 index
        Record<X> sel = P[p1];
        if (random.nextDouble() <= this.cr) { // crossover!
          do { // find a second, different record
            p2 = random.nextInt(this.mu);
          } while (p2 == p1); // repeat until p1 != p2
// perform recombination of the two selected records
          this.binary.apply(sel.x, P[p2].x, dest.x, random);
        } else {
// create modified copy of parent using unary operator
          this.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 Ingredient: Diversity Preservation

In Section 3.4.1.4, we asked why a population is helpful for optimization. Our answer was that there are two opposing goals in metaheuristic optimization: On the one hand, we want to get results *quickly* and, hence, want that the algorithms quickly trace down to the bottom of the basins around optima. On the other hand, we want to get *good* results, i.e., better local optima, preferably global optima. A smaller population is good for the former and forsters exploitation. A larger population is good for the latter, as it invests more time on exploration.

Well. Not necessarily. Imagine we discover a good local optimum, a solution better than everything else we have in the population. Great. It will survive selection and we will derive offspring solutions from it. Since it is a local optimum, these will probably be worse. They might also encode the same solution as the parent, which is entirely possible in our JSSP scenario. But even if they are worse, they maybe just good enough to survive the next round of selection. Then, their (better) parent will also survive. We will thus get more offspring from this parent. But also offsprings from its surviving offsprings. And some of these may again be the same as the parent. If this process keeps continuing, the population may slowly be filling with copies of that very good local optimum. The larger our population, the longer it will take, of course. But unless we somehow encounter a different, similarly good or even better solution, it will probably happen eventually.

What does this mean? Recombination of two identical points in the search space should yield the very same point as output, i.e., the binary operator will become useless. This would leave only the unary operator as possible source of randomness. We then practical have one point in the search space to which only the unary operator is applied. Our EA has become a weird hill climber.

If we would want that, then we would have implemented a hill climber, i.e., a local search, instead. In order to enable global exploration and to allow for the binary search operators to work, it makes sense to try to preserve the *diversity* in the population [193].

Now there exist quite a few ideas how to do that [48,184,189] and we also discuss some concepts later in Section 5.1.2.4. Many of them are focused on penalizing candidate solutions which are too similar to others in the selection step. Similarity could be measured via computing the distance in the search space, the distance in the solution space, the difference of the objective values. The penalty could be achieved by using a so-called *fitness* as basis for selection instead of the objective value. In our original EA, the fitness would be the same as the objective value. In a diversity-preserving EA, we could add a penalty value to this base fitness for each solution based on the distance to the other solutions.

3.4.5 Evolutionary Algorithm with Clearing in the Objective Space

Let us now test whether a diversity preserving strategy can be helpful in an EA. We will only investigate one very simple approach: Avoiding objective value duplicates [78,184]. In the rest of this section, we

will call this method *clearing*, as it can be viewed as the strictest possible variant of the clearing [161,162] applied in the objective space.

Put simply, we will ensure that all records that "survive" selection have different objective values. If two good solutions have the same objective value, we will discard one of them. This way, we will ensure that our population remains diverse. No single candidate solution can take over the population.

3.4.5.1 The Algorithm (with Recombination and Clearing)

We can easily extend our $(\mu + \lambda)$ EA with recombination from Section 3.4.3.1 to remove duplicates of the objective value. We need to consider that a full population of $\mu + \lambda$ records may contain less than μ different objective values. Thus, in the selection step, we may obtain $1 \leq u \leq \mu$ elements, where u can be different in each generation. If $u = 1$, we cannot apply the binary operator regardless of the crossover rate cr.

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 $\mu + \lambda$ of instances of I.
3. For index i ranging from 0 to $\mu + \lambda - 1$ do
   a. Create a random point from the search space using the nullary search operator and store it 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 in ascending order according to the objective values, i.e., such that the records r with better associated objective value $r.z$ are located at smaller indices.
   e. Iterate over P from front to end and delete all records with an objective value already seen in this iteration. The number of remaining records be w. Set the number u of selected records to $u = \min\{w, \mu\}$.
   f. Shuffle the first u elements of P randomly.
   g. Set the first source index $p1 = -1$.
   h. For index i ranging from u to $\mu + \lambda - 1$ do
      i. Set the first source index $p1$ to $p = (p1 + 1) \bmod u$.
      ii. Draw a random number c uniformly distributed in $[0, 1)$.

iii. If $u > 1$ and $c < cr$, then we apply the binary operator: A. Randomly choose another index $p2$ from $0 \ldots (u-1)$ such that $p2 \neq p1$. B. Apply binary search operator to the points stored at index $p1$ and $p2$ and store result at index i, i.e., set $P_i.x = \mathrm{searchOp}_2(P_{p1}.x, P_{p2}.x)$.
iv. otherwise to *step 4h.iii, i.e., if $c \geq cr$ or $u = 1$, we apply the unary search operator to the point stored at index $p1$ and store result at index i, i.e., set $P_i.x = \mathrm{searchOp}_1(P_{p1}.x)$.
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 3.16 and using the routine given in Listing 3.17 differs from the variant in Section 3.4.3.1 mainly in *step 4e*. There, the sorted population P is processed from beginning to end. Whenever an objective value is found in a record which has already been encountered during this processing step, the record is removed. Since P is sorted, this means that the record at (zero-based) index k is deleted if and only if $k > 0$ and $P_k.z = P_{k-1}.z$. As a result, the number u of selected records with unique objective value may be less than μ (while always being greater or equal to 1). Therefore, we need to adjust the parts of the algorithm where parent solutions are selected for generating offsprings. Also, we generate $\mu + \lambda - u$ offspring, to again obtain a total of $\mu + \lambda$ elements.

In the actual implementation in Listing 3.16, we do not delete the records but move them to the end of the list, so we can re-use them later. We also stop processing P as soon as we have μ unique records, as it does not really matter whether un-selected records are unique. This is slightly more efficient, but would be harder to write in pseudo-code.

We will name setups of this algorithm in the same manner as those of the original EA, except that we start the names with the prefix `eac_` instead of `ea_`.

3.4.5.2 The Right Setup

With the simple diversity-preservation mechanism in place, we may wonder which population sizes are good. It is easy to see that findings from Section 3.4.1.2, where we found that $\mu = \lambda = 16'384$ is reasonable, may longer hold: We know from Section 2.5.3 of the makespan for any solution on the JSSP instance `abz7` is 656. It can be doubted whether it is even possible to generate $16'384$ schedules with different makespans. If not, then the number u of selected records would always be less than μ, which would make choosing a large μ useless.

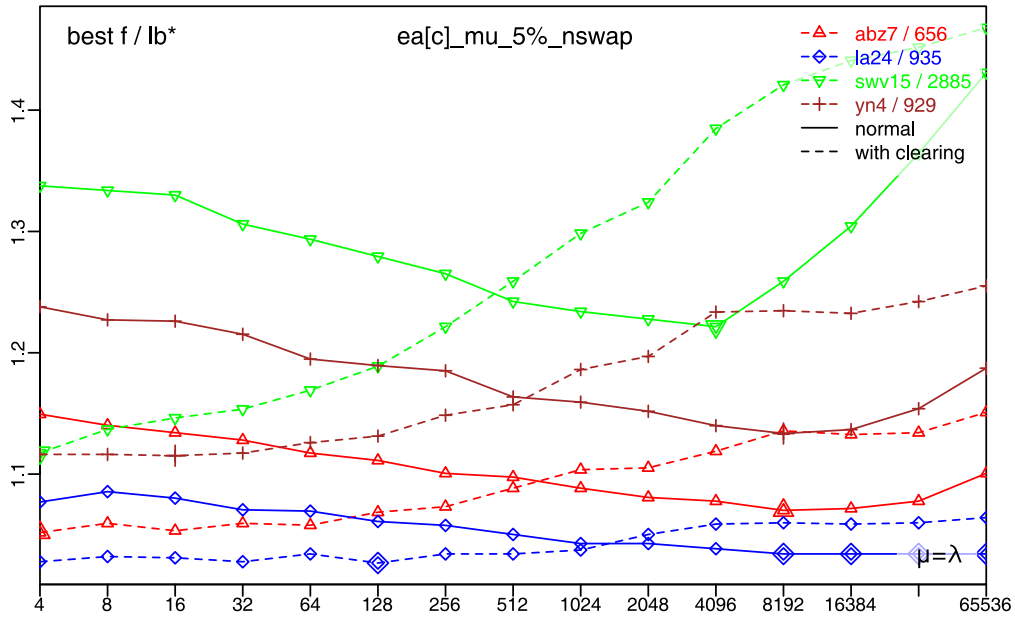

Figure 3.20: The median result quality of the `ea[c]_mu_5%_nswap` algorithm, divided by the lower bound $\mathrm{lb}(f)^\star$ from Table 2.2 over different values of the population size parameter $\mu = \lambda$, with and without clearing. The best values of μ for each operator and instance are marked with bold symbols.

Since this time smaller population sizes may be interesting, we investigate all powers of 2 for $\mu = \lambda$ from 4 to 65'536. In Figure 3.20, we find that the `eac_mu_5%_nswap` behave entirely different from those of `ea_mu_5%_nswap`. If clearing is applied, the smallest investigated setting, $\mu = \lambda = 4$, seems to be the right choice. This setup has the best performance on `abz7` and `swv15`, while being only a tiny bit worse than the best choices on `la24` and `yn4`. Larger populations lead to worse results at the end of the computational budget of three minutes.

Why could this be? One possible reason could be that maybe not very many different candidate solutions are required. If only few are needed and we can maintain sufficiently many in a small population, then the advantage of the small population is that we can do many more iterations within the same computational budget.

This very small population size means that an EA with clearing in the objective space should be configured quite similar to a hill climber!

3.4.5.3 Results on the JSSP

We can now investigate whether our results have somewhat improved.

Table 3.11: The results of the Evolutionary Algorithm `eac_4_5%_nswap` in comparison to `ea_8192_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	ea_8192_5%_nswap	684	703	702	**8**	**54s**	**10'688'314**
		eac_4_5%_nswap	**672**	**690**	**690**	9	68s	12'474'571
la24	935	ea_8192_5%_nswap	943	967	967	**11**	**18s**	**4'990'002**
		eac_4_5%_nswap	**935**	**963**	**961**	16	30s	9'175'579
swv15	2885	ea_8192_5%_nswap	3498	3631	3632	**65**	178s	**17'747'983**
		eac_4_5%_nswap	**3102**	**3220**	**3224**	65	**168s**	18'245'534
yn4	929	ea_8192_5%_nswap	1026	1056	1053	**17**	**114s**	**13'206'552**
		eac_4_5%_nswap	**1000**	**1038**	**1037**	18	118s	15'382'072

Table 3.11 shows that our EA with recombination and clearing in the objective space outperforms the EA without clearing on every single instance in terms of best, mean, and median result quality. Especially on instance swv15, the new algorithm performs much better than `ea_8192_5%_nswap`. Most remarkable is that we can even solve instance la24 to optimality once. The median solutions of our new algorithm variant are illustrated in Figure 3.21. Compared to Figure 3.15, the result on swv15 has improved: especially the upper-left and lower-right corners of the Gantt chart, where only few jobs are scheduled, have visibly become smaller.

Figure 3.21: The Gantt charts of the median solutions obtained by the eac_4_5%_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.

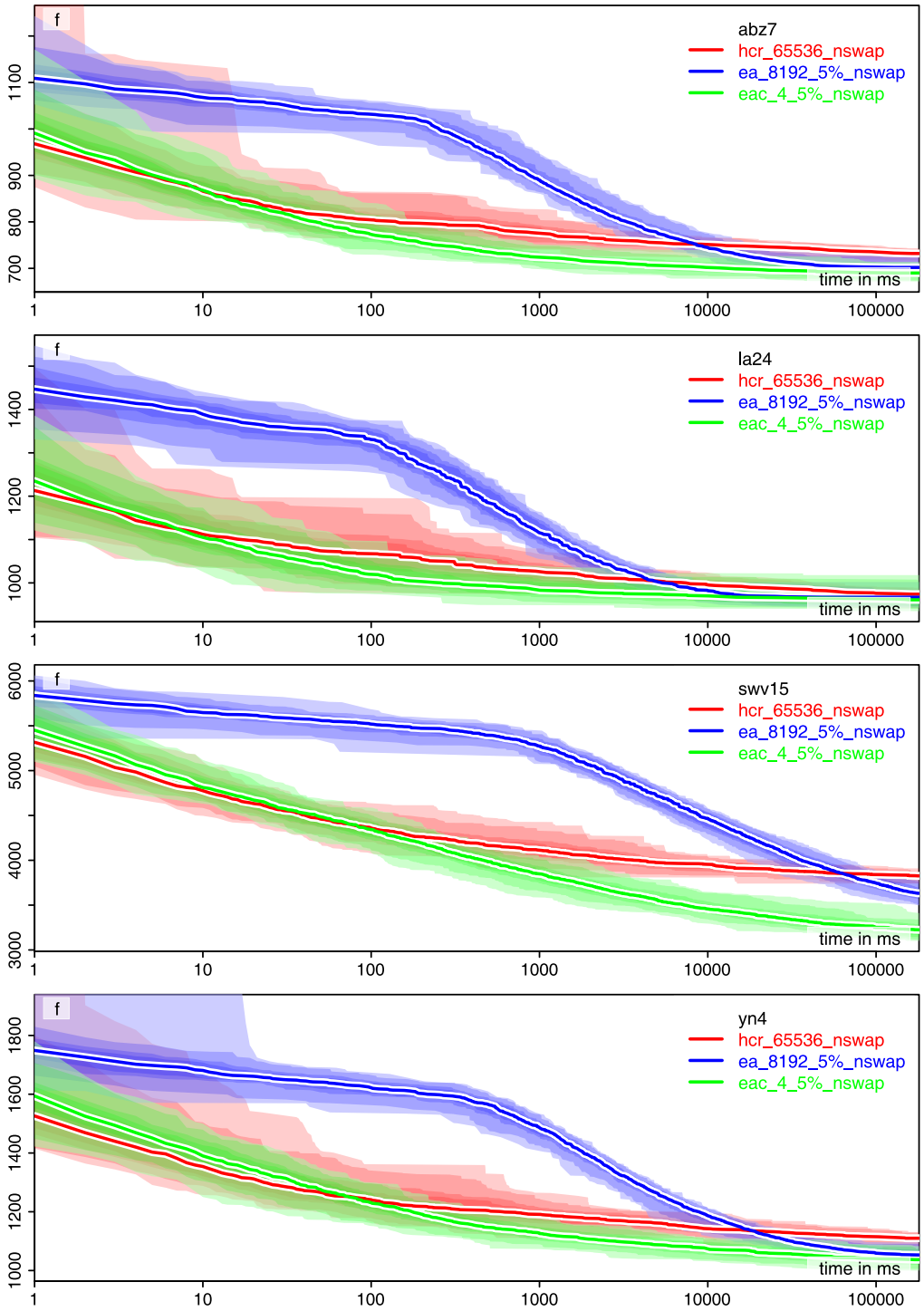

Figure 3.22: The median of the progress of the `eac_4_5%_nswap` in comparison to the `ea_8192_5%_nswap` and `hcr_65536_nswap` over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis). The color of the areas is more intense if more runs fall in a given area.

We plot the discovered optimal solution for `la24` in Figure 3.23. Comparing it with the median solution for `la24` in Figure 3.21, time was saved, e.g., by arranging the jobs in the top-left corner in a tighter pattern.

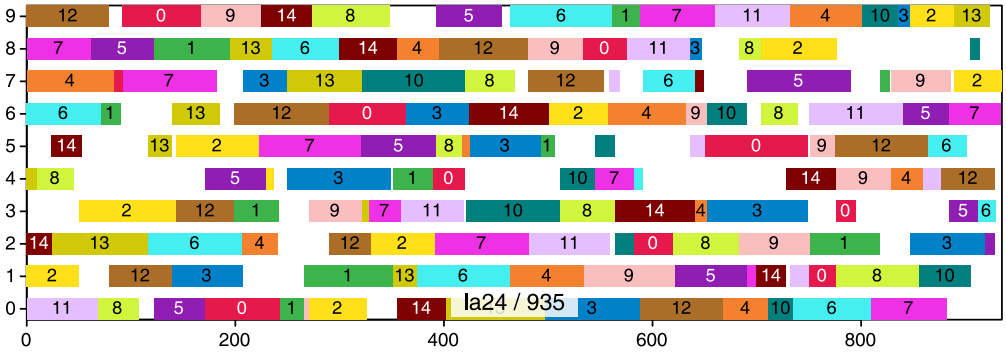

Figure 3.23: One *optimal* Gantt charts for instance `la24`, discovered by the `eac_4_5%_nswap` setup. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

From the progress charts plotted in Figure 3.22, we can confirm that `eac_4_5%_nswap` indeed behaves more similar to the hill climber `hcr_65536_nswap` than to the other EA `ea_8192_5%_nswap`. This is due to its small population size. However, unlike the hill climber, its progress curve keeps going down for a longer time. After only 0.1 seconds, it keeps producing better results in median.

We can confirm that even the simple and rather crude pruning in the objective space can significantly improve the performance of our EA. We did not test any sophisticated diversity preservation method. We also did not re-evaluate which crossover rate cr and which unary operator (`1swap` or `nswap`) works best in this scenario. This means that we might be able to squeeze out more performance, but we will leave it at this. The small populations working so well make us curious, however.

Listing 3.16 An excerpt of the implementation of the Evolutionary Algorithm algorithm with and clearing. (src)

```
public class EAWithClearing<X, Y>
    extends Metaheuristic2<X, Y> {
  public void solve(IBlackBoxProcess<X, Y> process) {
// Omitted: Initialize local variables random, searchSpace, set
// arrays P of length mu+lambda, and array T to null. Fill P with
// random solutions + evaluate.
    for (int i = P.length; (--i) >= 0;) {
      X x = searchSpace.create();
      this.nullary.apply(x, random);
      P[i] = new Record<>(x, process.evaluate(x));
      if (process.shouldTerminate()) { // we return
        return; // best solution is stored in process
      }
    }

    while (!process.shouldTerminate()) { // main loop
      RandomUtils.shuffle(random, P, 0, P.length); // make fair
      int u = Utils.qualityBasedClearing(P, this.mu);
// Now we have 1 <= u <= mu unique solutions.
      RandomUtils.shuffle(random, P, 0, u); // for fairness
      int p1 = -1; // index to iterate over first parent
// Overwrite the worse (mu + lambda - u) solutions.
      for (int index = P.length; (--index) >= u;) {
// Omitted: Quit loop if process.shouldTerminate()
        Record<X> dest = P[index]; // offspring
        p1 = (p1 + 1) % u; // parent 1 index
        Record<X> sel = P[p1]; // parent 1
        if ((u >= 2) && (random.nextDouble() <= this.cr)) {
          do { // find a second, different record
            p2 = random.nextInt(u);
          } while (p2 == p1); // Of course, can't be p1.
          this.binary.apply(sel.x, P[p2].x, dest.x, random);
        } else { // Otherwise: Mutation.
          this.unary.apply(sel.x, dest.x, random);
        }
        dest.quality = process.evaluate(dest.x);
      } // the end of the offspring generation
    } // the end of the main loop
  }
}
```

Listing 3.17 The implementation of the objective-value based clearing routine. (src)

```
1  public static int qualityBasedClearing(Record<?>[] array,
2      int max) {
3
4    Arrays.sort(array, Record.BY_QUALITY);// best -> first
5
6    int unique = 0;
7    double lastQuality = Double.NEGATIVE_INFINITY; // impossible
8
9    for (int index = 0; index < array.length; index++) {
10     Record<?> current = array[index];
11     double currentQuality = current.quality;
12     if (currentQuality > lastQuality) { // unique so-far
13       if (index > unique) { // need to move forward?
14         Record<?> other = array[unique];
15         array[unique] = current; // swap with first non-unique
16         array[index] = other;
17       }
18       lastQuality = currentQuality; // update new quality
19       if ((++unique) >= max) { // are we finished?
20         return unique; // then quit: unique == max
21       }
22     }
23   }
24
25   return unique; // return number of unique: 1<=unique<=max
26 }
```

3.4.6 $(1+1)$ EA

The $(1+1)$ EA is a special case of the $(\mu+\lambda)$ EA with $\mu=1$ and $\lambda=1$. Since the number of parents is $\mu=1$, it does not apply the binary recombination operator. We explicitly discuss it here because it is usually defined slightly differently from what we did in Section 3.4.1.1 and implemented as Listing 3.13.

3.4.6.1 The Algorithm

The $(1+1)$ EA works like a hill climber (see Section 3.3), with the difference that the new solution replaces the current one if it is better *or equally good*, instead of just replacing it if it is better.

1. Create one random point x in the 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. Repeat until the termination criterion is met:
   a. Apply the unary search operator to x to get a 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 $z'\leq z$, then store x' in x, store y' in y, and store z' in z.
5. Return the best encountered objective value z and the best encountered solution y to the user.

This algorithm is implemented in Listing 3.18. The *only* difference to the hill climber in Section 3.3.2.1 and Listing 3.10 is the $z'\leq z$ in *point 4.d* instead of $z'<z$: the new solution does not need to be strictly better to win over the old one, it is sufficient if it is not worse. In the $(\mu+\lambda)$ EA implementation that we had before, we could also set $\mu=1$ and $\lambda=1$. The algorithm would be slightly different from the $(1+1)$ EA: In the $(1+1)$ EA as defined here, the new solution y' will always win against the current solution y if $z'=z$, whereas in our shuffle-and-sort method of the population as implemented in Listing 3.13, either one could win with probability 0.5.

3.4.6.2 Results on the JSSP

We can now apply our $(1+1)$ EA to the four JSSP instances either using the `1swap` operator (`ea_1+1_1swap`) or the `nswap` operator (`ea_1+1_nswap`). Astonishingly, Table 3.12 reveals that it performs *better* than our best EA so far, namely `eac_4_5%_nswap`. Both $(1+1)$ EA setups also perform much better than our hill climber. The log-scaled Figure 3.25 shows that the two EAs without

Listing 3.18 An excerpt of the implementation of the $(1+1)$ EA. (src)

```
public class EA1p1<X, Y> extends Metaheuristic1<X, Y> {
  public void solve(IBlackBoxProcess<X, Y> process) {
// initialize local variables xCur, xBest, random
    X xCur = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();// get random gen

// create starting point: a random point in the search space
    this.nullary.apply(xBest, random); // xBest=random point
    double fBest = process.evaluate(xBest); // map & evaluate

    while (!process.shouldTerminate()) {
// create a slightly modified copy of xBest and store in xCur
      this.unary.apply(xBest, xCur, random);
// map xCur from X to Y and evaluate candidate solution
      double fCur = process.evaluate(xCur);
      if (fCur <= fBest) { // we found a not-worse solution
// remember best objective value and copy xCur to xBest
        fBest = fCur;
        process.getSearchSpace().copy(xCur, xBest);
      } // otherwise, i.e., fCur > fBest: just forget xCur
    } // until time is up
  } // process will have remembered the best candidate solution
}
```

population have better median solution almost always during the runs. And the Gantt charts of the median solutions of `ea_1+1_1swap`, illustrated in Figure 3.24, again appear denser.

Table 3.12: The results of the $(1+1)$ EA with the `nswap` and the `1swap` operator, in comparison to `eac_4_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	ea_1+1_1swap	**664**	678	**677**	**7**	**35s**	**7'727'896**
		ea_1+1_nswap	**664**	**677**	**677**	7	40s	15'323'407
		eac_4_5%_nswap	672	690	690	9	68s	12'474'571
la24	935	ea_1+1_1swap	941	957	955	**14**	4s	1'701'938

$\mathcal{I}$	$\mathrm{lb} f$	setup	best	mean	med	sd	med(t)	med(FEs)
		ea_1+1_nswap	938	**956**	**951**	14	**4s**	**1'650'810**
		eac_4_5%_nswap	**935**	963	961	16	30s	9'175'579
swv15	2885	ea_1+1_1swap	2958	**3041**	**3039**	**35**	**130s**	**16'644'658**
		ea_1+1_nswap	**2954**	3045	3047	39	132s	29'179'630
		eac_4_5%_nswap	3102	3220	3224	65	168s	18'245'534
yn4	929	ea_1+1_1swap	981	**1005**	**1003**	12	**48s**	**15'040'152**
		ea_1+1_nswap	**973**	1006	1005	**11**	61s	24'500'204
		eac_4_5%_nswap	1000	1038	1037	18	118s	15'382'072

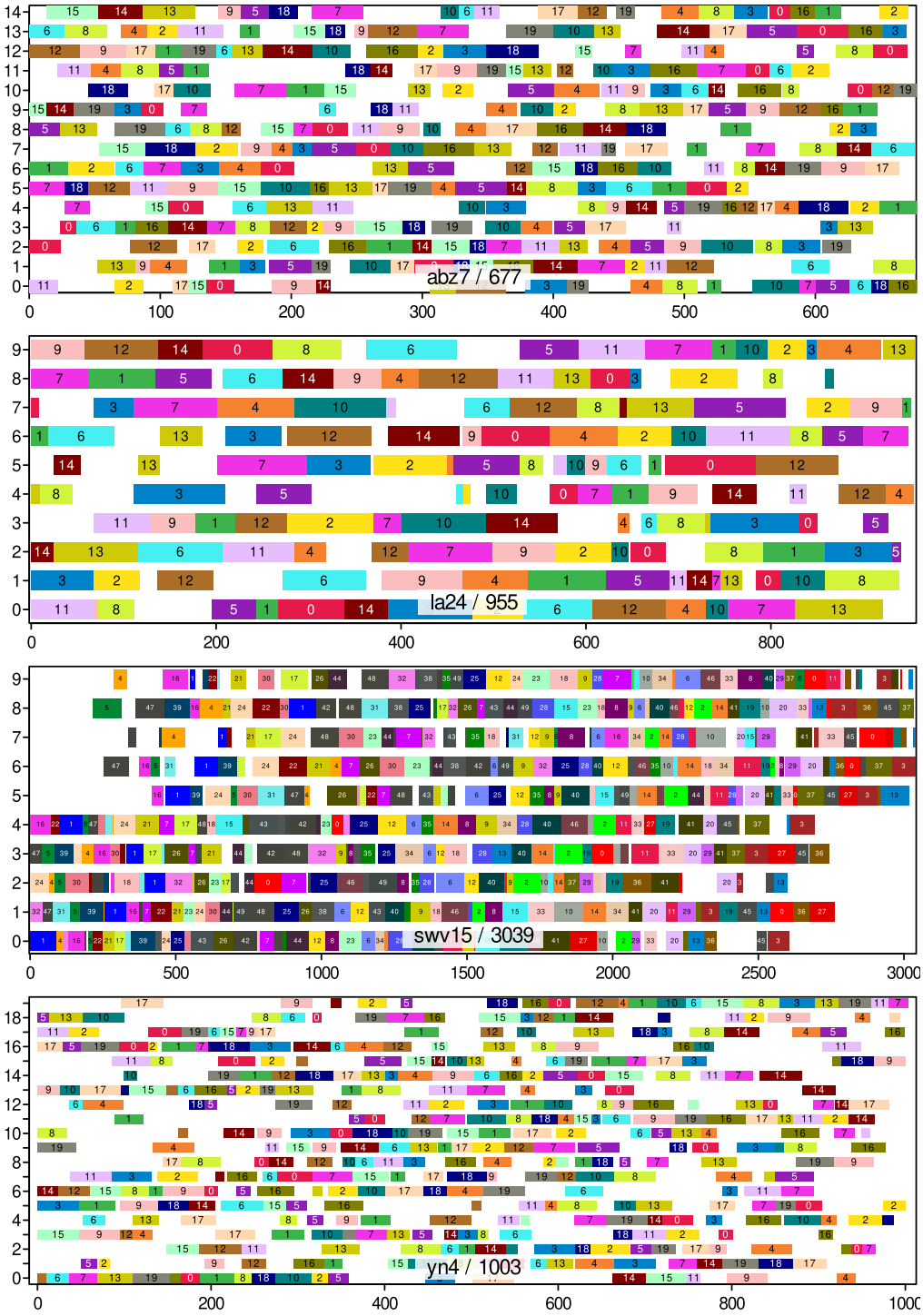

Figure 3.24: The Gantt charts of the median solutions obtained by the ea_1+1_1swap setup. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

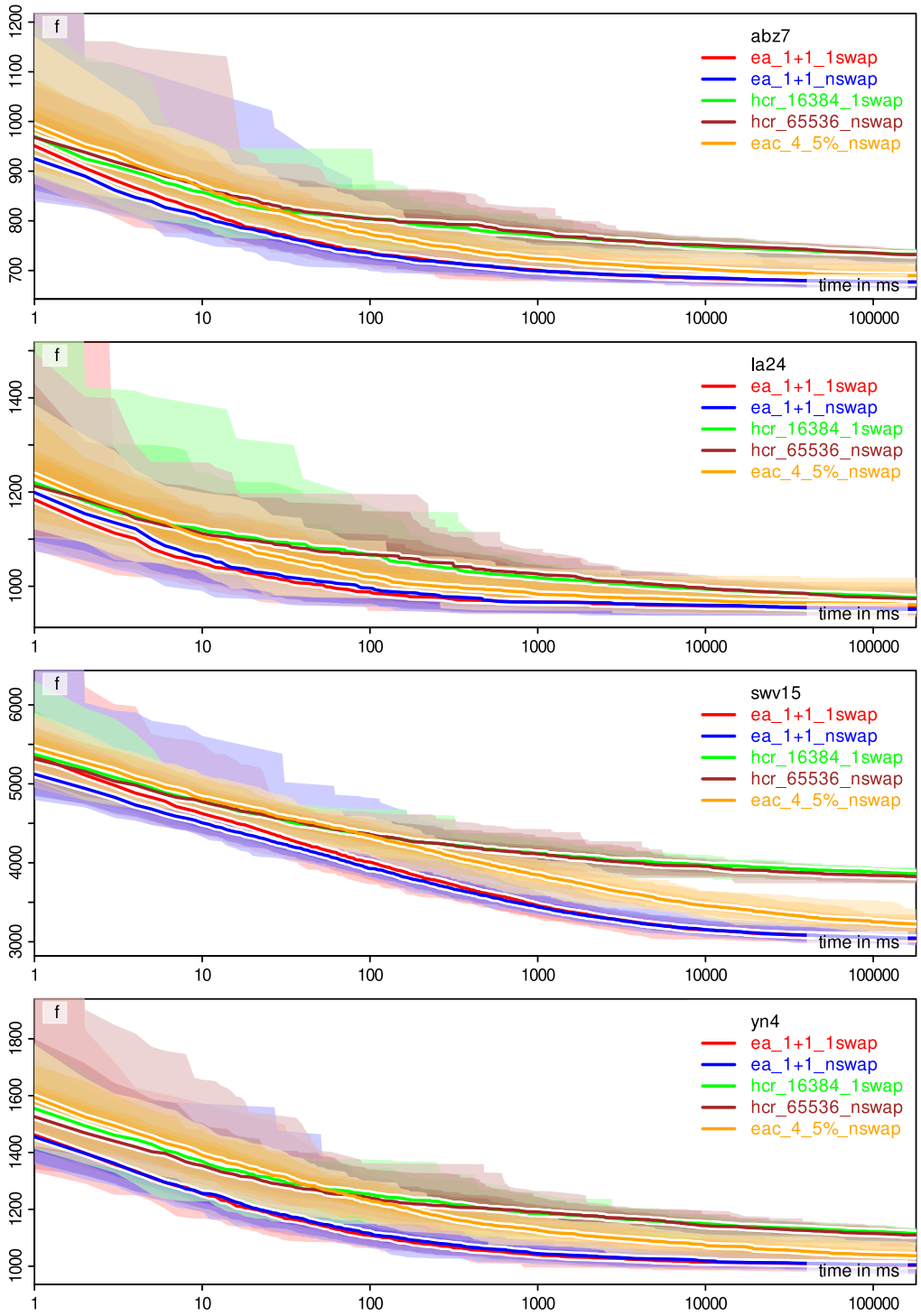

Figure 3.25: The median of the progress of the `ea_1+1_1swap` and `ea_1+1_nswap` compared to `eac_4_5%_nswap` and the two hill climbers `hcr_16384_nswap` and `hcr_65536_nswap` over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis). The color of the areas is more intense if more runs fall in a given area.

3.4.6.3 Discussion

The big advantage of EAs is that their population guards against premature convergence. This can bring better end results, but comes at a trade-off of slower convergence [214]. The fact that the EA without population performs best in our experimental setup is annoying for the author. It does not mean that this is always the case, though. But why is it the case here:

The answer consists, most likely, of two parts: First, as we know, our search space $\mathbb{X}$ is much larger than the solution space $\mathbb{Y}$, i.e., $|\mathbb{X}| \gg |\mathbb{Y}$. If we have an integer string $x_1 \in \mathbb{X}$ representing a Gantt chart $y_1 = \gamma(x_1)$, then we can swap two jobs and get a new string $x_2 \in \mathbb{X}$ with $x_2 \neq x_2$, but this does not necessarily mean that this new string maps to a different solution. It could well be that $\gamma(x_1) = \gamma(x_2)$. Then, we have made a *neutral* move. Of course, our move could also be neutral if $\gamma(x_1) \neq \gamma(x_2)$ but $f(\gamma(x_1)) = f(\gamma(x_2))$ Our $(1+1)$ EA can drift along a network of points in the search space which all map to solutions with the same makespan. This means that the $(1+1)$ EA can explore far beyond the neighborhood visible to the hill climber. By drifting over the network, it may eventually reach a point which maps to a better solution. This reasoning is supported by the fact that our $(1+1)$ EA perform much better than the hill climbers. You can find a more detailed discussion on neutrality in Section 5.4.1.

The second reason is probably that the three minutes of runtime are simply not enough for the EAs with the really big populations to reap the benefit of being resilient against premature convergence.

3.4.7 Summary

In this chapter, we have introduced Evolutionary Algorithms as methods for global optimization. We have learned the two key concepts that distinguish them from local search: the use of a *population of solutions* and of a *binary search operator*. We found that even the former alone can already outperform the simple hill climber with restarts. While we were a bit unlucky with our choice of the binary operator, our idea did work at least a bit.

We then noticed that populations are only useful if they are *diverse*. If all the elements in the population are very similar, then the binary operator stops working and the EA has converged. From our experiments with the hill climber, we already know premature convergence to a local optimum as something that should be avoided.
It therefore makes sense to try adding a method for enforcing diversity as another ingredient into the algorithm. Our experiments with a very crude diversity enhancing method – only allowing one solution per unique objective value in the population – confirmed that this can lead to better results.

We also found that the strict criterion to only accept solutions which are *strictly better* than the current one, as practiced in our hill climbers Section 3.3 may not be a good idea. Accepting solutions which are *not worse* than the current one allows for drift in the search space, which can yield better results.

3.5 Simulated Annealing

So far, we have only discussed two variants of local search: the hill climbing algorithm and the $(1+1)$ EA. 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). The $(1 + 1)$ EA may additionally utilize drift over solutions of the same quality, but there is no reason to assume that it cannot get stuck in local optima as well.

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) [47,116,125,164] is a local search which provides another approach to escape local optima [188,205]. The algorithm is inspired by the idea of simulating the thermodynamic process of *annealing* using statistical mechanics, hence the naming [144]. Instead of restarting the algorithm when reaching a local optimum, it tries to preserve the parts of the current 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 their objective values to the current solution.
3. The probability also decreases with the number of performed search steps.

These three principles are "injected" into the main loop of the hill climber. This is realized as follows.

Let us assume that $x \in \mathbb{X}$ is the "current" point that our local search maintains. $x' \in \mathbb{X}$ is the "newly sampled" point, i.e., the result of the application of the unary search operator to x. Then, ΔE be the difference between the objective value corresponding to x' and x. In other words, if γ is the

representation mapping and f the objective function, then:

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

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 point x' is actually better (or, at least, not worse) than the current point x, i.e., $\Delta E \leq 0$, then we will definitely accept it. (This is exactly the acceptance criterion used in the $(1+1)$ EA.) However, the new solution is not necessarily rejected otherwise: If the new point x' is worse ($\Delta E > 0$), then the acceptance probability

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 the exponential term and the two above points follow from $e^{-a} < e^{-b} \forall a > b$. We also have $e^{-a} \in (0,1) \forall a > 0$, so it can be used as probability value.

The temperature will be changed automatically such that it decreases and approaches zero with a rising number τ of algorithm iterations, i.e., the performed objective function evaluations. The optimization process is initially "hot" and T is high. Then, even significantly worse solutions may be accepted Over time, the process "cools" down and T decreases. The search slowly accepts 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." 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 $\tau = 1$. This 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 current 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 3.19 An excerpt of the abstract base class for temperature schedules. (src)

```
1 public abstract class TemperatureSchedule
2     implements ISetupPrintable {
3   public double startTemperature;
4
5   public abstract double temperature(long tau);
6 }
```

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 3.19.

The two most common temperature schedule implementations may be the *exponential* and the *logarithmic* schedule.

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 3.20. Besides the start temperature T_s, it has a parameter $\epsilon \in (0, 1)$ which tunes the speed of the temperature decrease. Higher values of ϵ lead to a faster temperature decline.

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

Listing 3.20 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)));
  }
}
```

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 3.21.. It, too, has the parameters $\epsilon \in (0, \infty)$ and T_s. Larger values of ϵ again lead to a faster temperature decline.

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

Listing 3.21 An excerpt of the logarithmic temperature schedules. (src)

```
public static class Logarithmic
    extends TemperatureSchedule {
  public double epsilon;
  public double temperature(long tau) {
    if (tau >= Long.MAX_VALUE) {
      return 0d;
    }
    return (this.startTemperature
        / Math.log(((tau - 1L) * this.epsilon) + Math.E));
  }
}
```

3.5.3 The Algorithm

Now that we have the blueprints for temperature schedules, we can completely define our SA algorithm and implement it in Listing 3.22.

1. Create random point x in the search space $\mathbb{X}$ by 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, which we will use to preserve the best-so-far results.
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 $z' \leq z$, then
      i. Store x' in x and store z' in z.
      ii. If $z' \leq z_b$, then store y' in y_b and store 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 going to *step 6*.
   h. Set $\Delta E = z' - z$ (see Equation (3.1)).
   i. Compute $P = e^{-\frac{\Delta E}{T}}$ (see Equation (3.2)).
   j. Draw a random number r uniformly distributed in $[0, 1)$.
   k. If $r \leq P$, then store x' in x, store z' in z, and perform next iteration by going to *step 6*.
7. Return best encountered objective value z_b and the best encountered solution z_b to the user.

There exist a several proofs [89,156] 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 [156]. 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 3.22 An excerpt of the implementation of the Simulated Annealing algorithm. (src)

```
public class SimulatedAnnealing<X, Y>
    extends Metaheuristic1<X, Y> {
  public TemperatureSchedule schedule;

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

    do { // repeat until budget exhausted
// create a slightly modified copy of xCur and store in xNew
      this.unary.apply(xCur, xNew, random);
      ++tau; // increase step counter
// map xNew from X to Y and evaluate candidate solution
      double fNew = process.evaluate(xNew);
      if ((fNew <= fCur) || // accept if better solution OR
          (random.nextDouble() < // probability is e^(-dE/T)
              Math.exp((fCur - fNew) / // -dE == -(fNew-fCur)
                  this.schedule.temperature(tau)))) {
// accepted: remember objective value and copy xNew to xCur
        fCur = fNew;
        process.getSearchSpace().copy(xNew, xCur);
      } // otherwise fNew > fCur and not accepted
    } while (!process.shouldTerminate()); // until time is up
  } // process will have remembered the best candidate solution
}
```

3.5.4 The Right Setup

Our algorithm has four parameters:

- the start temperature T_s,
- the parameter ϵ,
- the type of temperature schedule to use (here, logarithmic or exponential), and
- the unary search operator (in our case, we could use `1swap` or `nswap`).

We will only consider `1swap` as choice for the unary operator and focus on the exponential temperature schedule. We have two more parameters to set: T_s and ϵ and thus refer to the settings of this algorithm with the naming scheme `sa_exp_Ts_epsilon_1swap`.

At first glance, it seems entirely unclear how what to do with these parameters. However, we may get some ideas about their rough ranges if we consider Simulated Annealing as an improved hill climber.

Then, we can get some leads from our prior experiments with that algorithm.

Table 3.13: The median of total performed function evaluations and the standard deviation *sd* of the final result qualities of the hill climber `hc_1swap`.

$\mathcal{I}$	med(total FEs)	sd
abz7	35'648'639	28
la24	70'952'285	56
swv15	21'662'286	137
yn4	27'090'511	48
median	31'369'575	52

In Table 3.13, we print the standard deviation *sd* of the final result qualities that our `hc_1swap` algorithm achieved on our four JSSP instances. This tells us something about how far the different local optima at which `hc_1swap` can get stuck are apart in terms of objective value. The value of *sd* ranges from 28 on `abz7` to 137 on `swv15`. The median standard deviation over all four instances is about 50. Thus, accepting a solution which is worse by 50 units of makespan, i.e., with $\Delta E \approx 50$, should be possible at the beginning of the optimization process.

How likely should accepting such a value be? Unfortunately, we are again stuck at making an arbitrary choice – but at least we can make a choice from within a well-defined region: Probabilities must be in $[0, 1]$ and can be understood relatively intuitively, whereas it was completely unclear in what range reasonable "temperatures" would be located. Let us choose that the probability P_{50} to accept a candidate solution that is 50 makespan units worse than the current one, i.e., has $\Delta E = 50$, should be $P_{50} = 0.1$ at the beginning of the search. In other words, there should be a 10% chance to accept such a solution at $\tau = 1$. At $\tau = 1, T(\tau) = T_s$ for both temperature schedules. Of course, at $\tau = 1$ we do not really use the probability formula to decide whether or not to accept a solution, but we can expect that the temperature at $\tau = 2$ would still be very similar to T_s and we are using rough and rounded approximations anyway, so let's not make our life unnecessarily complicated here. We can now solve Equation (3.2) for T_s:

$$\begin{aligned} P_{50} &= e^{-\frac{\Delta E}{T(\tau)}} \\ 0.1 &= e^{-\frac{50}{T_s}} \\ \ln 0.1 &= -\frac{50}{T_s} \\ T_s &= -\frac{50}{\ln 0.1} \\ T_s &\approx 21.7 \end{aligned}$$

A starting temperature T_s of approximately $T_s = 20$ seems to be suitable in our scenario. Of course, we just got there using very simplified and coarse estimates. If we would have chosen $P_{50} = 0.5$, we would have gotten $T_s \approx 70$ and if we additionally went with the maximum standard deviation 137 instead of the median one, we would obtain $T_s \approx 200$. But at least we have a first understanding of the range where we will probably find good values for T_s.

But what about the ϵ parameters? In order to get an idea for how to set it, we first need to know a proper end temperature T_e, i.e., the temperature which should be reached by the end of the run. It cannot be 0, because while both temperature schedules do approach zero for $\tau \to \infty$, they will not actually become 0 for any finite number τ of iterations.

So we are stuck with the task to pick a suitably small value for T_e. Maybe here our previous findings from back when we tried to restart the hill climber can come in handy. In Section 3.3.3.2, we learned that it makes sense to restart the hill climber after $L = 16'384$ unsuccessful search steps. So maybe a terminal state for our Simulated Annealing could be a scenario where the probability P_e of accepting a candidate solution which is $\Delta E = 1$ makespan unit worse than the current one should be $P_e = 1/16'384$? This would mean that the chance to accept a candidate solution being marginally worse than the current one would be about as large as making a complete restart in `hcr_16384_1swap`. Of course, this is again an almost arbitrary choice, but it at least looks like a reasonable terminal state for our Simulated Annealing algorithm. We can now solve Equation (3.2) again to get the end temperature T_e:

$$\begin{aligned} P_e &= e^{-\frac{\Delta E}{T(\tau)}} \\ 1/16'384 &= e^{-\frac{1}{T_e}} \\ \ln\left(1/16'384\right) &= -\frac{1}{T_e} \\ T_e &= -\frac{1}{\ln\left(1/16'384\right)} \\ T_e &\approx 0.103 \end{aligned}$$

We choose a final temperature of $T_e = 0.1$. But when should it be reached? In Table 3.13, we print the total number of function evaluations (FEs) that our `hc_1swap` algorithm performed on the different problem instances. We find that it generated and evaluated between 22 million on `swv15` and 71 million

on `la24` candidate solutions.[2] The overall median is at about 30 million FEs within the 3 minute computational budget. From this, we can conclude that after about 30 million FEs, we should reach approximately $T_e = 0.1$. We can solve Equation (3.3) for ϵ to configure the exponential schedule:

$$\begin{aligned} T(\tau) &= T_s * (1-\epsilon)^{\tau-1} \\ T(30'000'000) &= 20 * (1-\epsilon)^{30'000'000-1} \\ 0.1 &= 20 * (1-\epsilon)^{29'999'999} \\ 0.1/20 &= (1-\epsilon)^{29'999'999} \\ 0.005^{1/29'999'999} &= 1-\epsilon \\ \epsilon &= 1-0.005^{1/29'999'999} \\ \epsilon &\approx 1.776 * 10^{-7} \end{aligned}$$

We can conclude, for an exponential temperature schedule, settings for ϵ somewhat between $1! \cdot 10^{-7}$ and $2 \cdot 10^{-7}$ seem to be a reasonable choice if the start temperature T_s is set to 20.

In Figure 3.26, we illustrate the behavior of the exponential temperature schedule for starting temperature $T_s = 20$ and the six values $5 \cdot 10^{-8}$, $1 \cdot 10^{-7}$, $1.5 \cdot 10^{-7}$, $2 \cdot 10^{-7}$, $4 \cdot 10^{-7}$, and $8 \cdot 10^{-7}$. The sub-figure on top shows how the temperature declines over the performed objective value evaluations. Starting at $T_s = 20$ it reaches close to zero for $\epsilon \geq 1.5 \cdot 10^{-7}$ after about $\tau = 30'000'000$ FEs. For the smaller ϵ values, the temperature would need longer to decline, while for larger values, it declines quicklier. The next three sub-figures show how the probability to accept candidate solutions which are worse by ΔE units of the objective value decreases with τ for $\Delta E \in \{1, 3, 10\}$. This decrease is, of course, the direct result of the temperature decrease. Solutions with larger ΔE clearly have a lower probability of being accepted. The larger ϵ, the earlier and faster does the acceptance probability decrease.

[2] Notice that back in Table 3.3, we printed the median number FEs until the best solution was discovered, not until the algorithm has terminated.

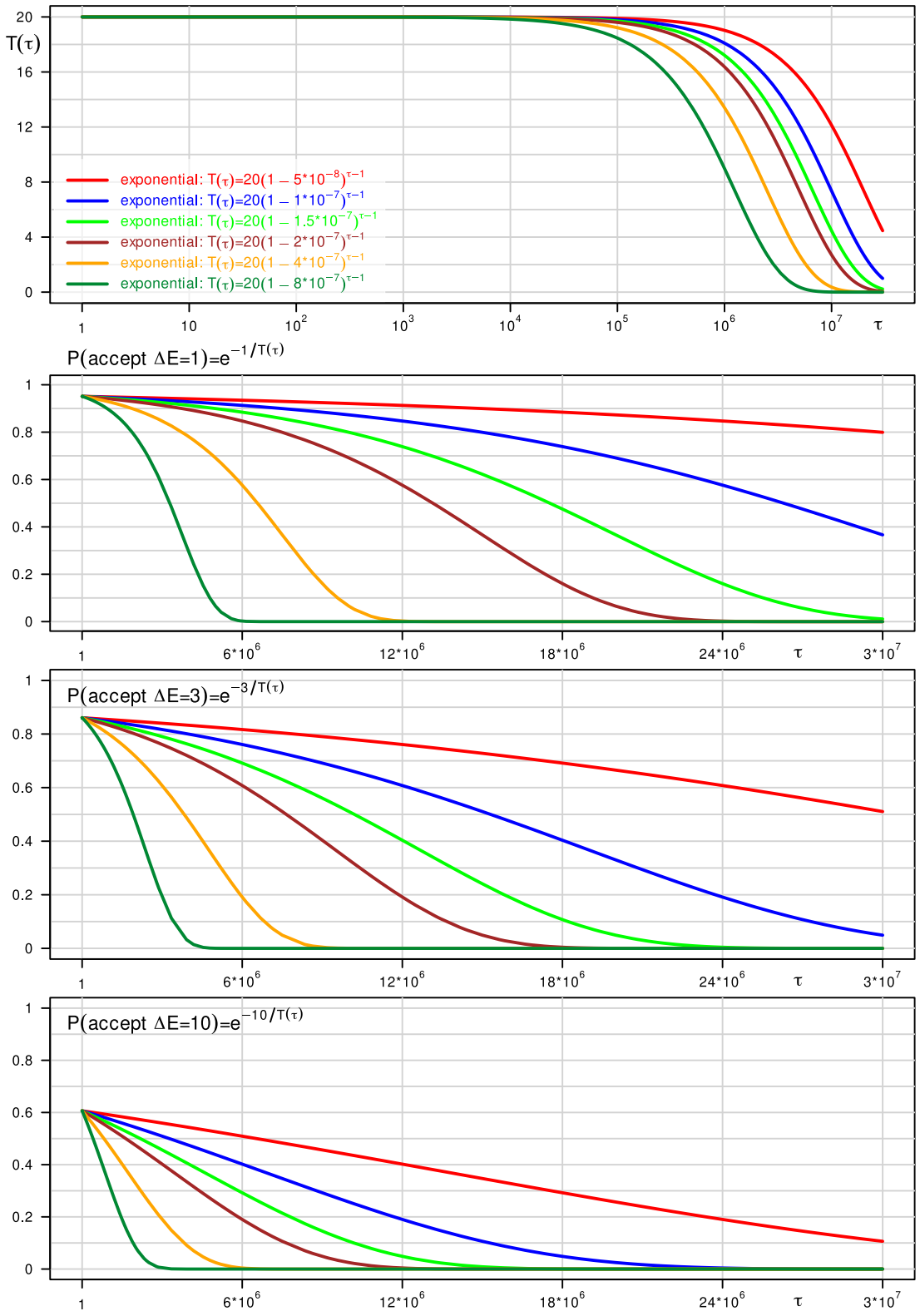

Figure 3.26: The temperature progress of six exponential temperature schedules (top) plus their probabilities to accept solutions with objective values worse by 1, 3, or 10 than the current solution.

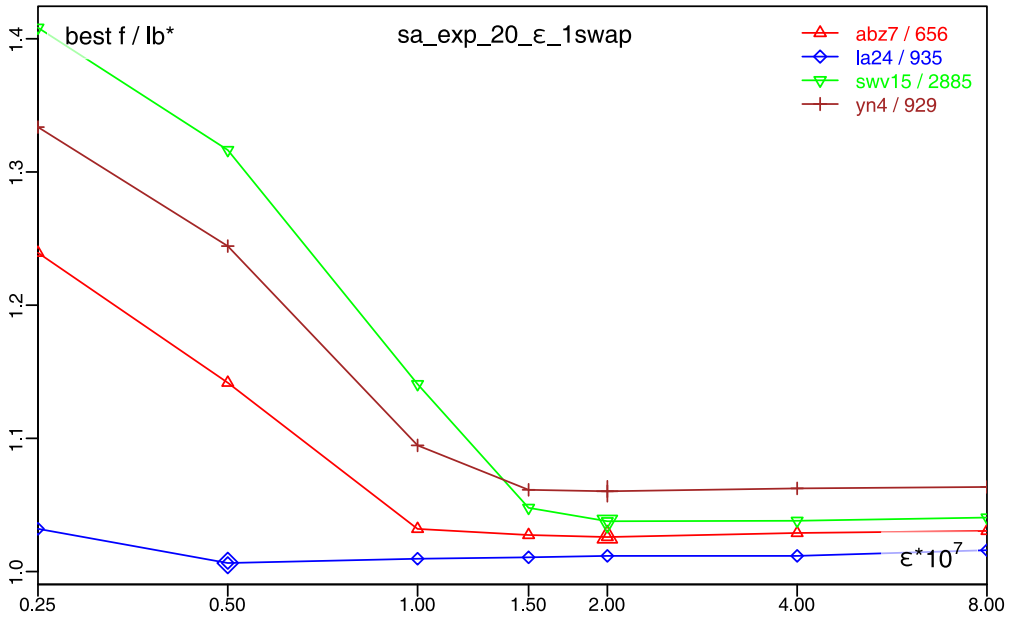

Figure 3.27: The median result quality of the `sa_exp_20_epsilon_1swap` algorithm, divided by the lower bound $\mathrm{lb}(f)^\star$ from Table 2.2 over different values of the parameter ϵ. The best values of L on each instance are marked with bold symbols.

In Figure 3.27, we illustrate the normalized median result quality that can be obtained by Simulated Annealing with starting temperature $T_s = 20$, exponential schedule, and `1swap` operator for different values of the parameter ϵ, including those from Figure 3.26. Interestingly, it turns out that $\epsilon = 2$ is the best choice for the instances `abz7`, `swv15`, and `yn4`, which is quite close to what we could expect from our calculation. Smaller values will make the temperature decrease more slowly and lead to too much exploration and too little exploitation, as we already know from Figure 3.26. They work better on instance `la24`, which is no surprise: From Table 3.13, we know that on this instance, we can conduct more than twice as many objective value evaluations than on the others within the three minute budget: On `la24`, we can do enough steps to let the temperature decrease sufficiently even for smaller ϵ.

3.5.5 Results on the JSSP

We can now evaluate the performance of our Simulated Annealing approach on the JSSP. We choose the setup with exponential temperature schedule, the `1swap` operator, the starting temperature $T_s = 20$, and the parameter $\epsilon = 2 \cdot 10^{-7}$. For simplicity, we refer to it as `sa_exp_20_2_1swap`.

Table 3.14: The results of the Simulated Annealing setup `sa_exp_20_2_1swap` in comparison with the best EA, `eac_4_5%_nswap`, and the hill climber with restarts `hcr_16384_1swap`. 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_16384_1swap	714	732	733	6	91s	18'423'530
		eac_4_5%_nswap	672	690	690	9	**68s**	**12'474'571**
		sa_exp_20_2_1swap	**663**	**673**	**673**	**5**	112s	21'803'600
la24	935	hcr_16384_1swap	953	976	976	**7**	80s	34'437'999
		eac_4_5%_nswap	**935**	963	961	16	**30s**	**9'175'579**
		sa_exp_20_2_1swap	938	**949**	**946**	8	33s	12'358'941
swv15	2885	hcr_16384_1swap	3752	3859	3861	42	**92s**	**11'756'497**
		eac_4_5%_nswap	3102	3220	3224	65	168s	18'245'534
		sa_exp_20_2_1swap	**2936**	**2994**	**2994**	**28**	157s	20'045'507
yn4	929	hcr_16384_1swap	1081	1115	1115	11	**91s**	**14'804'358**
		eac_4_5%_nswap	1000	1038	1037	18	118s	15'382'072
		sa_exp_20_2_1swap	**973**	**985**	**985**	**5**	130s	20'407'559

In Table 3.14, we compare results of the Simulated Annealing setup `sa_exp_20_2_1swap` to those of our best EA, `eac_4_5%_nswap`, and the hill climber with restarts `hcr_16384_1swap`. Simulated Annealing is better than these three algorithms in terms of the best, mean, and median result on almost all instances. Only on `la24`, `eac_4_5%_nswap` can win in terms of the best discovered solution, which already was the optimum. Our SA setup also is more reliable than the other algorithms, its standard deviation and only on `la24`, the standard deviation sd of its final result quality is not the lowest.

We know that on `la24`, $\epsilon = 2 \cdot 10^{-7}$ is not the best choice for SA and smaller values would perform better there. Interestingly, in the experiment, the settings $\epsilon = 4 \cdot 10^{-7}$ and $\epsilon = 8 \cdot 10^{-7}$ (not listed in the table) also each discovered a globally optimal solution on that instance. In Figure 3.29, we illustrate the one found by `sa_exp_20_8_1swap`.

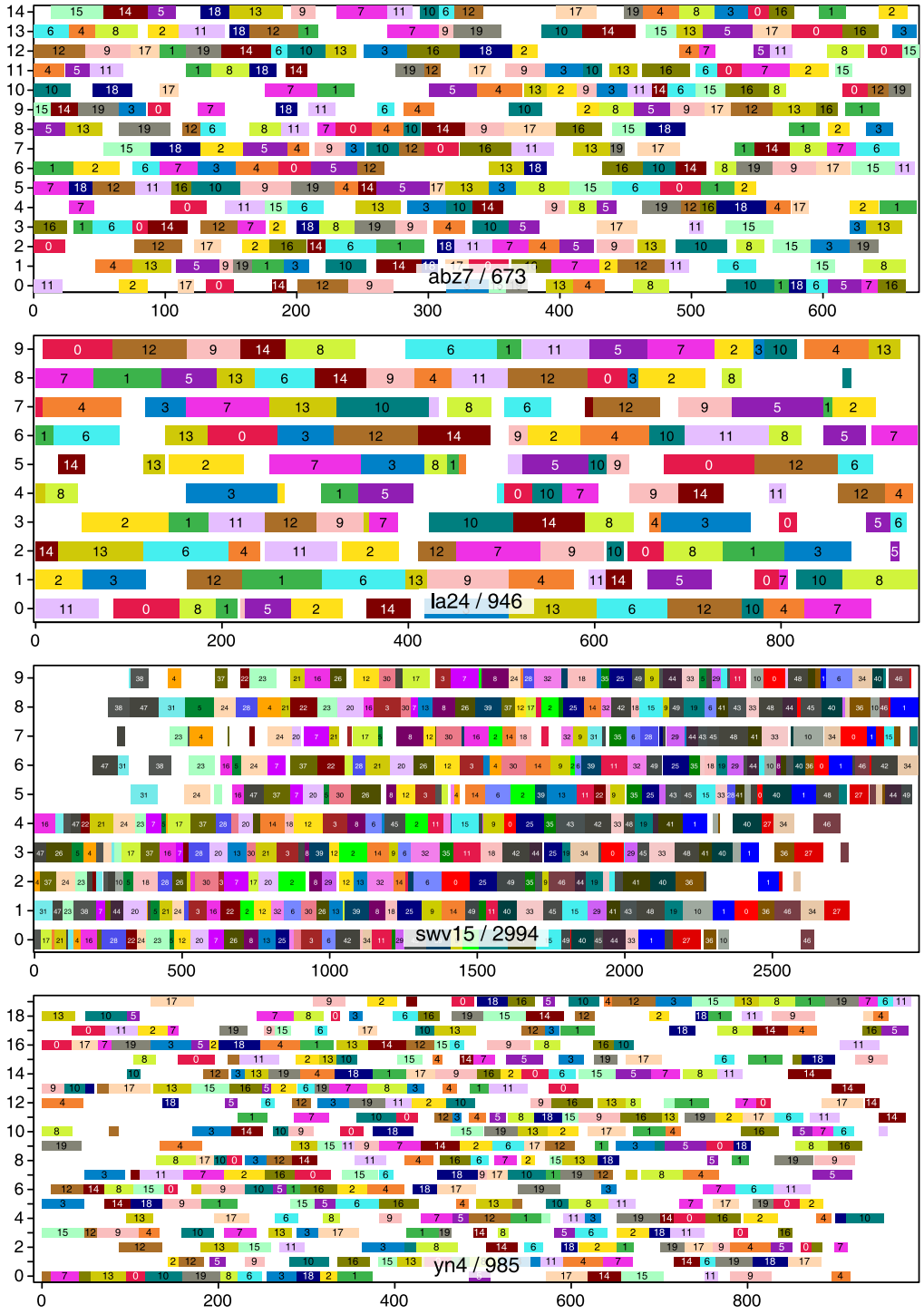

Figure 3.28: The Gantt charts of the median solutions obtained by the `sa_exp_20_2_1swap` algorithm. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of each diagram denote the instance name and makespan.

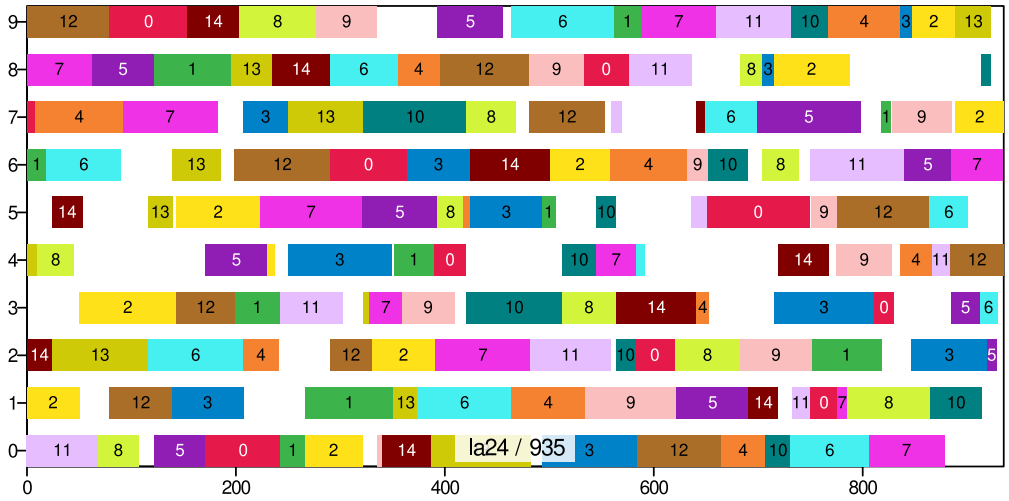

Figure 3.29: The Gantt chart of the best solution obtained by the `sa_exp_20_8_1swap` algorithm on `la24`, which happens to be *optimal*. The x-axes are the time units, the y-axes the machines, and the labels at the center-bottom of the diagram denotes the makespan.

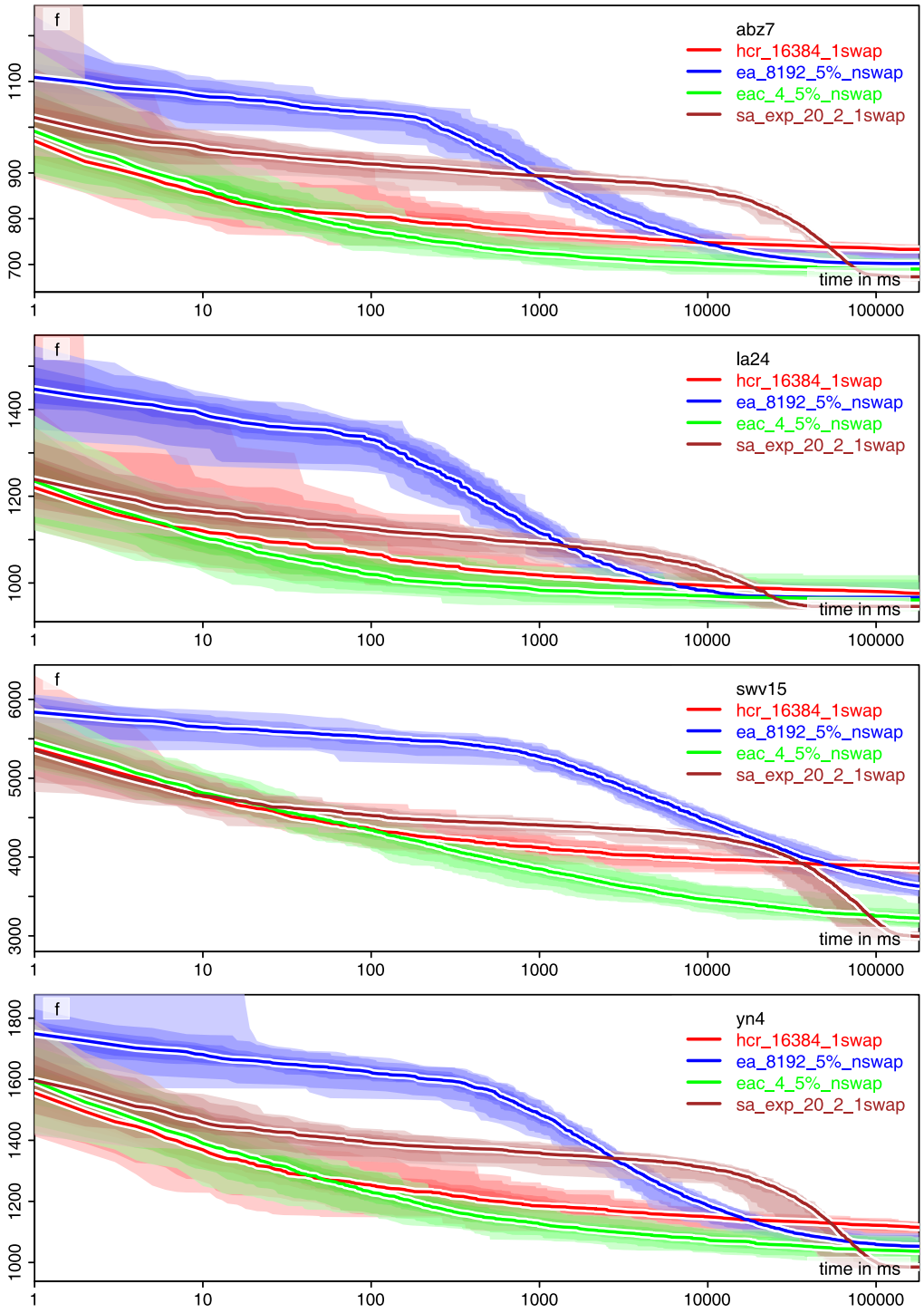

Figure 3.30: The median of the progress of the algorithms `sa_exp_20_2_1swap`, `ea_8192_5%_nswap`, `eac_4_5%_nswap`, and `hcr_16384_1swap` over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis). The color of the areas is more intense if more runs fall in a given area.

If we compare our sa_exp_20_2_1swap with the related work, we find its best and mean solution quality on abz7 surpass those of the original Fast Simulated Annealing algorithm and its improved version HFSAQ from [4]. Its mean and best results of sa_exp_20_2_1swap on la24 outperform the algorithms proposed in [2,10,118,122,157,158]. On yn4, it outperforms all four AntGenSA algorithms (complex hybrids of three algorithms including SA and EAs) from [107] in mean and best result quality. Since this is an educational book, we are not really aiming for solving the JSSP outstandingly well and only use a very small set of instances. Our algorithms are not very complicated, but these comparisons indicate that they are at least somewhat working.

We plot the Gantt charts of the median result of sa_exp_20_2_1swap in Figure 3.28. Especially on instance swv15, changes are visible in comparison to the results produced by eac_4_5%_nswap and illustrated in Figure 3.21.

In Figure 3.30, we plot the progress of sa_exp_20_2_1swap over time in comparison to ea_8192_5%_nswap, eac_4_5%_nswap, and hcr_16384_1swap. Especially for swv15 and yn4, we find that sa_exp_20_2_1swap converges towards end results that are very visibly better than the other algorithms.

We also notice that the median solution quality obtained by sa_exp_20_2_1swap looks very similar to the shape of the temperature curve in Figure 3.26. Under the exponential schedule that we use, both the temperature and acceptance probability remain high for some time until they suddenly drop. Interestingly, the objective value of the best-so-far solution in SA seems to follow that pattern. Its , it first declines slowly, then there is a sudden transition where many improvements are made, before the curve finally becomes flat. This relationship between the temperature and the obtained result quality shows us that configuring the SA algorithm correctly is very important. Had we chosen ϵ too small or the start temperature T_s too high, then the quick decline could have shifted beyond our three minute budget, i.e., would not take place. Then the results of Simulated Annealing would have been worse than those of the other three algorithms. This also explains the worse results for smaller ϵ shown in Figure 3.27.

The behavior of SA is different from the hill climber and small-population EA, which do not exhibit such a transition region. The EA ea_8192_5%_nswap with the larger population also shows a transition from slow to faster decline, but there it takes longer and is much less steep.

3.5.6 Summary

Simulated Annealing is an optimization algorithm which tunes from exploration to exploitation during the optimization process. Its structure is similar to the hill climber, but different from that simple local search, it also sometimes moves to solutions which are worse than the one it currently holds. While it will always accept better solutions, the probability to move towards a worse solution depends on how

much worse that solution is (via ΔE) and on the number of search steps already performed. This later relationship is implemented by a so-called "temperature schedule": At any step τ, the algorithm has a current temperature $T(\tau)$. The acceptance probability of a worse solution is computed based on how bad its objective value is in comparison and based on $T(\tau)$. Two temperature schedules are most common: letting $T(\tau)$ decline either logarithmically or exponentially. Both have two parameters, the start temperature T_s and ϵ.

In our experiments, we only considered the exponential schedule. We then faced the problem of how to set the values of these obscure parameters T_s and ϵ. We did this by using the experience we already gained from our experiments with the simple hill climber: We already know something about how different good solutions can be from each other. This provides us with the knowledge of how "deep" a local optimum may be, i.e., what kind of values we may expect for ΔE. We also know roughly how many algorithm steps we can perform in our computational budget, i.e., have a rough idea of how large τ can become. Finally, we also know roughly the number L of function evaluations after which it made sense to restart the hill climber. By setting the probability to accept a solution with $\Delta E = 1$ to $1/L$, we got a rough idea of temperature that may be good at the end of the runs. The word "rough" appeared quite often in the above text. It is simply not really possible to "compute" the perfect values for T_s and ϵ (and we could not compute the right values for μ, λ, nor cr for the Evolutionary Algorithm either). But the roughly computed values gave us a good idea of suitable settings and we could confirm them in a small experiments. Using them, our Simulated Annealing algorithm performed quite well. The very crude calculations in Section 3.5.4 may serve as rule-of-thumb in other scenarios, too.

3.6 Hill Climbing Revisited

Until now, we have entirely relied on randomness to produce new points in the search space. Our nullary, unary, and binary operators are all randomized. 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, in our hill climber, we cannot know whether or when we have tested the complete neighborhood around the current best point x in the search space. Thus, we also never 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 number L of steps. Thus, the restart might be too early, as there may still be undiscovered

solutions in the neighborhood of x. It also might happen too late: 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 enumerate over them completely. As soon as we encounter an improvement, we can stop the enumeration and accept the newly discovered better point. If we have finished enumerating all possible `1swap` neighbors and none of them corresponds to 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 can know it directly. Also, as detailed in Section 6.1.2, we might even find the improving moves faster in average, because we would never try the same move twice 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 `true`, then the enumeration will be terminated. If `false` is returned, it will continue if there are more points in the neighborhood.

The call-back function provided by the optimization process could internally apply the representation mapping γ to x' and compute the objective value $f(y')$ of the resulting candidate solution $y' = \gamma(x')$. If that solution is better than what we get for x, the call-back function could store it and return `true`. This would stop the enumeration process and would return the control back to the main loop of the optimization algorithm. Otherwise, the call-back function 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 3.23 A the generic interface for unary search operators, now able to enumerate neighborhoods. (src)

```
1  public interface IUnarySearchOperator<X>
2      extends ISetupPrintable {
3    void apply(X x, X dest, Random random);
4    default boolean enumerate(Random random, X x,
5        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 3.23, 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 of the Java 8-interface `Predicate`. The `test` 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 a reasonable way: In the case of the `nswap`, operator, for instance, all other points in the search space could potentially be reached from the current one (just with different probabilities). 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 should only be overwritten by operators that span neighborhoods which are sufficiently small for efficient enumeration. A reasonable limit is neighborhood whose size grows quadratically with the problem scale or at most with the third power of the problem scale.

3.6.2 Ingredient: Neighborhood Enumerating `1swap` Operators for the JSSP

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

3.6.2.1 Enumerating in Deterministic Order

The easiest idea is to just enumerate all index pairs (i, j). If the jobs at two indices i and j are different, we swap them, invoke the call-back function, then swap them back to the original order. Since swapping jobs at indices $i = j$ makes no sense and swapping the jobs at indices (i, j) is the same as swapping at indices (j, i), we only need to investigate $m * n * (m * n - 1)/2$ pairs.

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: A. Store job_i at index j in x'. B. Store job_j at index i in x'. C. 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. D. Store job_i at index i in x'. E. Store job_j at index j in x'.

This simple algorithm is implemented in Listing 3.24, which only shows the new function that was added to our class `JSSPUnaryOperator1Swap` that we had already back in Section 3.3.1.

3.6.2.2 Random Enumeration Order

Our enumerating `1swap` operator has one drawback: The order in which it processes the indices is always the same. We always check swapping jobs at the lower indices first. Swap moves involving two jobs near the end of the arrays x are only checked if all other moves closer to the beginning have been checked. This introduces a bias in the way we search.

We again remember the concept mentioned right at the beginning of this chapter: If you do not know the best choice out of several options, pick a random one. While we can generate all neighbors, the order in which we generate them may be random! In other words, we now design an operator `1swapU` which enumerates the same neighborhood as `1swap`, but does so in a random order.

0. Let S be the list of all index pairs (i, j) with $0 < i < m * n$ and $0 \leq j < i$. It has the length $|S| = (m * n)(m * n - 1)/2$.
1. Make a copy x' of the input point x from the search space.
2. For index u from 0 to $|S| - 1$ do:
   a. Choose an index v from $u \ldots |S| - 1$ uniform at random.
   b. Swap the index pairs at indices u and v in S.
   c. Pick index pair $(i, j) = S[u]$.

Listing 3.24 An excerpt of the `1swap` operator for the JSSP, namely the implementation of the `enumerate` function from the interface `IUnarySearchOpertor` (Listing 3.23). (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 jobI = dest[i]; // remember job id at index i
    for (int j = i; (--j) >= 0;) { // iterate over 0..(i-1)
      int jobJ = dest[j]; // remember job at index j
      if (jobI != jobJ) { // both jobs are different
        dest[i] = jobJ; // then we swap the values
        dest[j] = jobI; // 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] = jobI; // revert the change
        dest[j] = jobJ; // 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
}
```

d. Store the job at index i in x' in variable job_i.
e. Store the job at index j in x' in variable job_j.
f. If $job_i \neq job_j$ then:
   i. Store job_i at index j in x'.
   ii. Store job_j at index i in x'.
   iii. 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.
   iv. Store job_i at index i in x'.
   v. Store job_j at index j in x'.

If this routine completes, then the lines *2a* and *2b* together will have performed a Fisher-Yates shuffle [76,129]. By always randomly choosing an index pair (i, j) from the not-yet-chosen ones, it will enumerate the complete `1swap` neighborhood in a uniformly random fashion. This might incur some small performance loss due to not being very cache-friendly. However, we will see that it can actually increase the search efficiency. We will refer to this algorithm as `1swapU` and implement it in Listing 3.25.

Listing 3.25 An excerpt of the `1swapU` operator for the JSSP, namely the random-order implementation of the `enumerate` function from the interface `IUnarySearchOpertor` (Listing 3.23). (src)

```
  public boolean enumerate(Random random, int[] x,
      int[] dest, Predicate<int[]> visitor) {
// indexes be the flattened list of unique index pairs and
// pairCount their number.
    System.arraycopy(x, 0, dest, 0, dest.length); // copy x

// We move along the index-pair array and shuffle the indices on
// the way with an iterative version of the Fisher-Yates shuffle.
    for (int i = 0, start = -1; i < pairCount; i++) {
// Get "a" and "b": the next, randomly chosen index pair.
      int swapWith = (i + random.nextInt(pairCount - i)) << 1;
      int a = indexes[swapWith];
      indexes[swapWith] = indexes[++start];
      indexes[start] = a;
      int b = indexes[++swapWith];
      indexes[swapWith] = indexes[++start];
      indexes[start] = b;

      int jobI = dest[a];// the job at first index
      int jobJ = dest[b];// the job at second index

      if (jobI != jobJ) {
        dest[a] = jobJ; // then we swap the values
        dest[b] = jobI; // 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[a] = jobI; // revert the change
        dest[b] = jobJ; // and continue
      }
    }
    return false; // we have enumerated the complete neighborhood
  }
```

3.6.3 The Algorithm (with Restarts)

We can now design a new variant of the hill climber which enumerates 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. Of course, it could also happen that the enumeration of the neighborhood is completed without discovering a better solution. In this case, we *know* that z_b is a local optimum. Then, we can simply restart at a new, random point. The general pattern of this algorithm is given below:

1. Set the best-so-far objective value z_b to $+\infty$ and the best-so-far candidate solution y_b to NULL.
2. Create a random point x in the 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. If $z < z_b$, then store z in z_b and store y in y_b.
6. Repeat until the termination criterion is met:
   a. For each point $x' \in \mathbb{X}$ neighboring to x 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 $z' < z$, then A. Store x' in x and store z' in z. B. If $z' < z_b$, then store y' in y_b and store z' in z_b.
      C. Stop the enumeration and go back to *step 6a*.
   b. If we arrive here, the neighborhood of x did not contain any better solution. Hence, we perform a restart by going back to *step 2*.
7. Return best encountered objective value z_b and the best encountered solution $solspel_b$ 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 Listing 3.23, 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 3.26.

Different from our original hill climber with restarts introduced in Section 3.3.3, this new algorithm does not need to count steps to know when to restart. It therefore also does not need a parameter L determining the number of non-improving FEs after which a restart should be performed. Its implementation in Listing 3.26 is therefore also shorter and simpler than the implementation of the original

algorithm variant in Listing 3.11. It should be noted that the new hill climber 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.

Listing 3.26 An excerpt of the implementation of the Hill Climbing algorithm with restarts based on neighborhood enumeration. (src)

```
1  public class HillClimber2WithRestarts<X, Y>
2      extends Metaheuristic1<X, Y> {
3    public void solve(IBlackBoxProcess<X, Y> process) {
4  // initialization of local variables xCur, xBest, random omitted
5  // for brevity
6      while (!process.shouldTerminate()) { // main loop
7  // create starting point: a random point in the search space
8  // put random point in xBest
9        this.nullary.apply(xBest, random);
10       fBest[0] = process.evaluate(xBest); // evaluate
11
12       do { // repeat until budget exhausted or no improving move
13 // enumerate all neighboring solutions of xBest and receive them
14 // one-by-one in parameter x (for which xCur is used)
15         improved = this.unary.enumerate(random, xBest, xCur,
16             x -> {
17 // map x from X to Y and evaluate candidate solution
18               double fCur = process.evaluate(x);
19               if (fCur < fBest[0]) { // found better solution
20 // remember best objective value and copy x to xBest
21                 fBest[0] = fCur;
22                 process.getSearchSpace().copy(x, xBest);
23                 return true; // quit enumerating neighborhood
24               }
25 // no improvement: continue enumeration unless time is up
26               return process.shouldTerminate();
27             });
28 // repeat until time is up or no further improvement possible
29         if (process.shouldTerminate()) {
30           return; // ok, we should exit
31         } // otherwise: continue inner loop as long as we
32       } while (improved); // can find improvements
33     } // outer loop: if we get here, we need to restart
34   } // process will have remembered the best candidate solution
35 }
```

3.6.4 Results on the JSSP

Table 3.15: The results of the new hill climbers hc2r_1swap and hc2r_1swapU in comparison with hcr_16384_1swap. 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_16384_1swap	714	732	733	6	91s	18'423'530
		hc2r_1swap	**705**	736	737	9	95s	17'631'217
		hc2r_1swapU	708	**731**	**731**	**6**	**79s**	**16'413'522**
la24	935	hcr_16384_1swap	953	976	976	**7**	80s	34'437'999
		hc2r_1swap	959	978	978	8	93s	41'131'198
		hc2r_1swapU	**952**	**973**	**974**	7	**78s**	**32'552'884**
swv15	2885	hcr_16384_1swap	3752	3859	3861	42	92s	11'756'497
		hc2r_1swap	**3628**	**3810**	**3807**	74	99s	13'949'301
		hc2r_1swapU	3731	3829	3831	**42**	**89s**	**11'380'041**
yn4	929	hcr_16384_1swap	1081	1115	**1115**	**11**	91s	14'804'358
		hc2r_1swap	1093	1131	1132	19	92s	**12'816'913**
		hc2r_1swapU	**1076**	**1114**	1116	13	**88s**	13'349'708

In Table 3.15, we list the results of new neighborhood-enumerating hill climbers with restarts (prefix hc2r). We compare the version using the deterministic neighborhood enumeration hc2r_1swap to hc2r_1swapU which enumerates the neighborhood in a random order. We further also list the results of hcr_16384_1swap, the stochastic hill climber which restarts after 16'384 unsuccessful steps. This setup was found to perform well in Section 3.3.3.2.

We find that hc2r_1swapU tends to have the edge over hc2r_1swap, except for instance swv15, where it does perform worse. Also, hc2r_1swapU and hcr_16384_1swap deliver very similar results, which also means that it performs worse than our Evolutionary Algorithms or Simulated Annealing.

In Figure 3.31, we plot the progress of the hc2r_1swap, hc2r_1swapU, and hcr_16384_1swap

algorithms over time. It is very surprising to see that the median of best-so-far solution qualities of `hc2r_1swapU` and `hcr_16384_1swap` are almost identical during the whole three minute computational budget and on all four JSSP instances. Both `hc2r_1swapU` and `hcr_16384_1swap` perform random job swaps in each step. `hc2r_1swapU` avoids trying the same move twice and will restart when it has arrived in a local optimum. `hcr_16384_1swap` may try the same move multiple times and performs a restart after $L = 16'384$ unsuccessfuly steps. The fact that both algorithms perform so very similar probably means that the restart setting of $L = 16'384$ for `hcr_16384_1swap` is probably a rather good choice.

It is also clearly visible that `hc2r_1swap` is initially slower than the other two algorithms, although its end result is still similar. This shows that enumerating the neighborhood of a solution in a random fashion is better than doing it always in the same deterministic way. This supports the idea of doing things in a randomized way if no clear advantage of a deterministic approach is visible.

The question arises why we would bother with the `hc2r`-type hill climbers if we seemingly can get the exact same behavior from a stochastic hill climber with restarts. One answer is the fact that we found a method to actually *know* whether a solution is an optimum instead of having to guess. Another answer is that we need one parameter less. We retain the black-box ability of the algorithm but have zero parameters (except the choice of the unary search operator), as opposed to the EA and SA algorithms which each have three (μ, λ, cr and temperature schedule, T_s, ϵ, respectively).

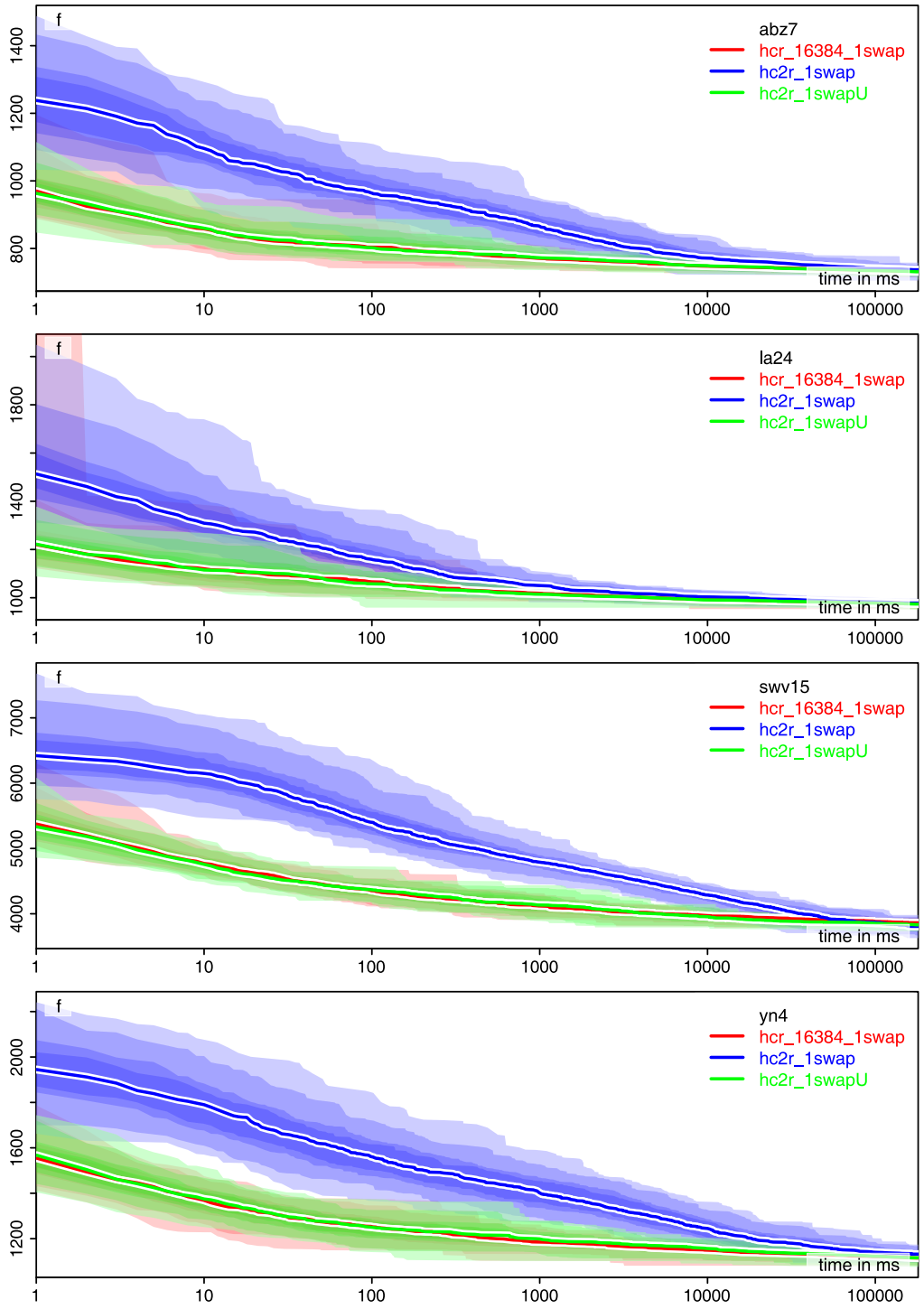

Figure 3.31: The median of the progress of the algorithms `hc2r_1swap`, `hc2r_1swapU`, and `hcr_16384_1swap` over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis). The color of the areas is more intense if more runs fall in a given area.

3.7 Memetic Algorithms: Hybrids of Global and Local Search

Let remember two of the algorithms we have already seen:

1. The hill climbers are local search methods, which can refine and improve one solution quickly but may get stuck at local optima.
2. Evolutionary Algorithms are global optimization methods, 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) [103,148,155] (sometimes also Lamarkian Evolution [220]).

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. From the perspective of a single point in the population, each generation of the EA is somewhat similar to one iteration of a hill climber. However, there are μ points in the surviving population, not just one. As a result, the overall progress made towards a good solution is much slower compared to the hill climber.

Also, 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). They will not contain many good components yet.

Memetic Algorithms try to mitigate both issues with one simple idea: Let each new point, before it enters the population, become the starting point of a local search. The result of this local search then enters the population instead.

3.7.2 Algorithm: EA Hybridized with Neighborhood-Enumerating Hill Climber

We could choose any type of local search for this purpose, but here we will use the iterative neighborhood enumeration as done by our revisited hill climber in Section 3.6.3. As a result, the first generation of the MA behaves exactly the same as our neighborhood-iterating hill climber with restarts (until it has done $\mu + \lambda$ restarts). 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).

In the reproduction phase, an Evolutionary Algorithm applies either a unary or a binary operator. In an MA, it obviously makes no sense to use the same unary operator as in the local search here. We could therefore either use an unary operator that always makes different moves or only use the binary search operator. Here, we will follow the latter choice, simply to spare us the necessity to define yet another operator here (`nswap` would not be suitable, as it most often does single swaps like `1swap`.)

The basic $(\mu + \lambda)$ Memetic Algorithm is given below and implemented in Listing 3.27.

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 $\mu + \lambda$ of instances of I.
3. For index i ranging from 0 to $\mu + \lambda - 1$ do
   a. Create a random point from the search space using the nullary search operator and store it in $P_i.x$.
4. Repeat until the termination criterion is met:
   b. For index i ranging from 0 to $\mu + \lambda - 1$ do
      i. If P_i is already a fully-evaluated solution and a local optimum, continue with the next iteration value of the *loop 4b*.
      ii. 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., $P_i.z = f(y)$.
      iii. *Local Search:* For each point x' in the search space neighboring to $P_i.x$ according to the unary search operator do: A. Map the point x' to a candidate solution y' by applying the representation mapping $y' = \gamma(x')$. B. Compute the objective value z' by invoking the objective function $z' = f(y')$. C. If the termination criterion has been met, jump directly to step 5. D. If $z' < z_b$, then store x' in $P_i.x$, store z' in $P_i.z$, stop the enumeration, and go back to *step 4b.iii*.
   c. Sort the array P in ascending order according to the objective values, i.e., such that the records r with better associated objective value $r.z$ are located at smaller indices.
   d. Shuffle the first μ elements of P randomly.
   e. Set the first source index $p1 = -1$.
   f. For index i ranging from μ to $\mu + \lambda - 1$ do
      iv. Set the first source index $p1$ to $p1 = (p1 + 1) \bmod \mu$.
      v. Randomly choose another index $p2$ from $0 \ldots (\mu - 1)$ such that $p2 \neq p$.

vi. Apply binary search operator to the points stored at index $p1$ and $p2$ and store result at index i, i.e., set $P_i.x = \mathrm{searchOp}_2(P_{p1}.x, P_{p2}.x)$.

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

The condition in *step 4b.i* allows that in the first iteration, all members of the population are refined by local search, whereas in the latter steps, only the λ new offsprings are refined. This makes sense because the μ parents have already been subject to local search and are local optima. Trying to refine them again would just lead to one useless enumeration of the entire neighborhood during which no improvement could be found.

Listing 3.27 An excerpt of the implementation of the Memetic Algorithm algorithm. (src)

```
public class MA<X, Y> extends Metaheuristic2<X, Y> {
  public void solve(IBlackBoxProcess<X, Y> process) {
// the initialization of local variables is omitted for brevity
// first generation: fill population with random solutions
    for (int i = P.length; (--i) >= 0;) {
// set P[i] = random solution (code omitted)
    }

    while (!process.shouldTerminate()) { // main loop
      for (LSRecord<X> ind : P) {
// If ind is not known to be local optimum, refine it with local
// search a la HillClimber2 for a given number of maximum steps
// (code omitted for brevity).
      } // end of 1 ls iteration: we have refined 1 solution
// sort the population: mu best records at front are selected
      Arrays.sort(P, Record.BY_QUALITY);
// 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;) {
        LSRecord<X> dest = P[index];
        LSRecord<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 solutions
        this.binary.apply(sel.x, P[p2].x, dest.x, random);
        dest.quality = process.evaluate(dest.x);
      } // the end of the offspring generation
    } // the end of the main loop
  }
}
```

3.7.3 The Right Setup

We can now evaluate the performance of our Memetic Algorithm variant. As unary operator with enumerable neighborhood, we use the `1swapU` operator. As binary search operator, we apply the `sequence` crossover operator defined in Section 3.4.2 for the EA.

Since we always apply this operator in the reproduction step of our MA (i.e., $cr = 1$), the only parameter we need to worry about is the population size. While a large population size was good for EAs, we need to remember that our budget is limited to 180 seconds and that every individual in the population will be refined using the `hc2r_1swapU`-style local search. This means that the limit for the total population size $\mu + \lambda$ would be the number of restarts that `hc2r_1swapU` can make within three minutes. Any larger size would mean that the first generation would not be completed.

Table 3.16: The median runtime that the neighborhood-enumerating hill climber would consume *without* restarts, i.e., the median time until arriving in a local optimum, as well as how many restarts we could do within our three-minute budget.

$\mathcal{I}$	med(total time) w/o restarts	med(restarts)
abz7	313 ms	575
la24	47 ms	3'830
swv15	1'729 ms	104
yn4	726 ms	248
median	520 ms	412

In Table 3.16, we apply `hc2r_1swapU`, but instead of restarting, we terminate the algorithm when it has arrived in a local optimum. We find that it needs between 47 ms (on `la24`) and 1'729 ms (on `swv15`) to do so. This means that within the 180 s, we can refine between 3'830 and 104 individuals with the local search. If we want to be able to do several generations of the MA, then $\mu + \lambda \ll 104$.

I did some small, preliminary experiments where I found that a population size of $\mu = \lambda = 8$ works well for our three minute budget on all instances except `swv15`. We will call this setup `ma_8_1swapU` and investigate it in the following text.

3.7.4 Results on the JSSP

Table 3.17: The results of the Memetic Algorithm `ma_8_1swapU` in comparison to the two EAs `ea_8192_5%_nswap` and `eac_4_5%_nswap` as well as the hill climber `hc2r_1swapU`. 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	eac_4_5%_nswap	672	690	690	9	68s	12’474’571
		ea_8192_5%_nswap	684	703	702	8	**54s**	**10’688’314**
		hc2r_1swapU	708	731	731	**6**	79s	16’413’522
		ma_8_1swapU	**671**	**689**	**689**	7	115s	23’304’852
la24	935	eac_4_5%_nswap	**935**	963	961	16	30s	9’175’579
		ea_8192_5%_nswap	943	967	967	11	18s	4’990’002
		hc2r_1swapU	952	973	974	**7**	78s	32’552’884
		ma_8_1swapU	939	**958**	**956**	11	**10s**	**3’832’439**
swv15	2885	eac_4_5%_nswap	**3102**	**3220**	**3224**	65	168s	18’245’534
		ea_8192_5%_nswap	3498	3631	3632	65	178s	17’747’983
		hc2r_1swapU	3731	3829	3831	**42**	**89s**	**11’380’041**
		ma_8_1swapU	3405	3602	3599	64	167s	22’837’065
yn4	929	eac_4_5%_nswap	**1000**	1038	1037	18	118s	15’382’072
		ea_8192_5%_nswap	1026	1056	1053	17	114s	**13’206’552**
		hc2r_1swapU	1076	1114	1116	**13**	**88s**	13’349’708
		ma_8_1swapU	1005	**1036**	**1035**	15	159s	25’517’707

Table 3.18: A statistical comparison of the end results of ma_8_1swapU, ea_8192_5%_nswap, and hc2r_1swapU, using the Mann-Whitney U test with Bonferroni correction and significance level $\alpha = 0.02$ on the four JSSP instances. The columns indicate the p-values and the verdict (? for insignificant).

Mann-Whitney U $\alpha' = 3.03 \cdot 10^{-4}$	abz7	la24	swv15	yn4
ea_8192_5%_nswap vs. hc2r_1swapU	$2.62 \cdot 10^{-33}$ <	$2.59 \cdot 10^{-6}$ <	$7.85 \cdot 10^{-34}$ <	$3.65 \cdot 10^{-34}$ <
ea_8192_5%_nswap vs. ma_8_1swapU	$4.52 \cdot 10^{-22}$ >	$4.54 \cdot 10^{-7}$ >	$3.39 \cdot 10^{-3}$?	$1.31 \cdot 10^{-13}$ >
hc2r_1swapU vs. ma_8_1swapU	$1.16 \cdot 10^{-34}$ >	$4.88 \cdot 10^{-19}$ >	$1.20 \cdot 10^{-34}$ >	$1.18 \cdot 10^{-34}$ >

In Table 3.17, we find that ma_8_1swapU performs clearly better than the plain EA and the hill climber on all four instances, which is confirmed by a statistical test in Table 3.18. Except on swv15, it also always has the best mean and median result quality. The differences to eac_4_5%_nswap are very small and therefore not interesting – except on swv15, where the Memetic Algorithm clearly loses. This must be the result of the very low number of individuals that can be refined on swv15 using the local search within the three minute budget.

From Figure 3.32, we can see that the ma_8_1swapU behaves almost identical to hc2r_1swapU during the first approximately ten seconds of the runs. This must be the time when the first $\mu + \lambda$ individuals undergo the local search. From then on, the algorithm makes better progress than hc2r_1swapU. It seems that our binary sequence operator can combine different good traits of candidate solutions after all! The fact that the ma_8_1swapU can improve beyond the hill climber means that sequence is able to combine two local optima to a new point in the search space, which then can be refined by local search to another local optimum.

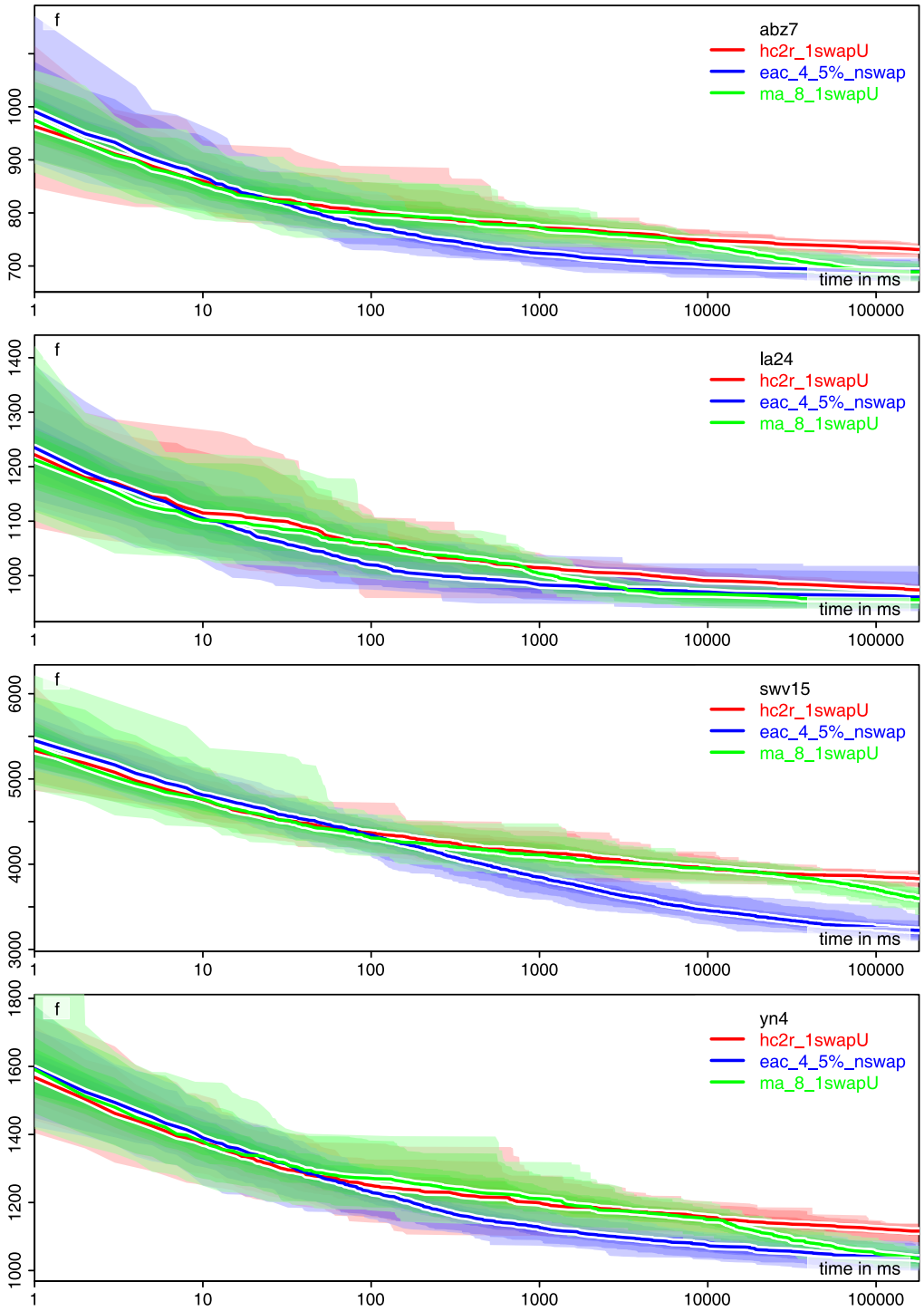

Figure 3.32: The median of the progress of the `ma_8_1swapU`, `eac_4_5%_nswap`, and `hc2r_1swapU` algorithms over time, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis). The color of the areas is more intense if more runs fall in a given area.

3.7.5 Summary

With Memetic Algorithms, we learned about a family of hybrid optimization methods that combine local and global search. Of course, here we just scratched the surface of this concept.

For example, we again only considered one simple implementation of this idea. Instead of `hc2r`-style local search, we could as well have used our stochastic hill climber or even Simulated Annealing for a fixed number of iterations as refinement procedure. Indeed, instead of doing a full local search until reaching a local optimum, we could also limit it to only a fixed number S of steps (and indeed the Java implementation Listing 3.27 has this feature).

Our MA did neither outperformed the EA with clearing nor Simulated Annealing.
Regarding the former, we could also apply clearing in the MA. I actually tried that, but it did not really lead to a big improvement in my preliminary experiments, so I did not discuss it here. The reason is most likely that the MA performs too few generations for it to really kick in.

The computational budget of only three minutes may have prevented us from finding better results. If we had a larger budget, we could have used a larger population. In a different application scenario, the comparison of an MA with the EA with clearing and SA algorithms might have turned out more favorable. On the plus side, the MA did perform significantly better than a pure EA with recombination and than the neighborhood-enumerating hill climber the two algorithms it is composed of!

3.8 Estimation of Distribution Algorithms

Estimation of Distribution Algorithms (EDAs) [131,137,150,152] follow a completely different paradigm than the methods that we have discussed up to here. So far, we have directly focused on the points x inside the search space $\mathbb{X}$. We tried to somehow "navigate" from one or multiple such points to better points, by investigating their neighbors via the unary operator or by trying to good find locations "between" two points via the binary operator. EDAs instead look at the bigger picture and try to discover and exploit the structure of the space $\mathbb{X}$ itself. The ask the questions "Can we somehow learn which areas in $\mathbb{X}$ are promising? Can we learn how good solutions look like?"

3.8.1 The Algorithm

How do good solutions look like? It is unlikely that good (or even optimal) solutions are uniformly distributed over the search space. If the optimization problem that we are trying to solve is reasonable, then it should have some structure and there will be areas in the search space $\mathbb{X}$ that are more likely to contain good solutions than others. *Distribution* here is already an important keyword: There should be some kind of (non-uniform) probability distribution of good solutions over the search space. If we

can somehow learn this distribution, we obtain a *model* of how good solutions look like. Then we could *sample* this distribution/model. Sampling a distribution just means to create random points according to it: If we sample λ points from a distribution M, then more points will be created in areas where M has a high probability density and fewer points from places where it has a low probability density.

A basic EDA works roughly as follows:

1. Set the best-so-far objective value z_b to $+\infty$ and the best-so-far candidate solution y_b to NULL.
2. Initialize the model M_0, e.g., to approximate a uniform distribution.
3. Create $\lambda > 2$ random points x in the search space $\mathbb{X}$, either by sampling the model or by using a nullary search operator.
4. Repeat until termination criterion is met (where i be the iteration index):
   a. Map the λ points to candidate solutions $y \in \mathbb{Y}$ using the representation mapping $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ and evaluate their objective value $z = f(y)$.
   b. y' be the best of the λ new candidate solutions and z' be its objective value. If $z' < z_b$, then store y' in y_b and z' in z_b.
   c. Select the μ best points from the set of λ points (with $1 < \mu < \lambda$).
   d. Use the set P_μ of these μ points to build the new model M_i. This step can also be implemented as model update U and make use of the information in the old model M_{i-1} as well as the problem instance $\mathcal{I}$, i.e., $M_i = U(P_\mu, M_{i-1}, \mathcal{I})$.
   e. Sample λ points from M_i.
5. Return the candidate solution y_b and its objective value z_b to the user.

This structure looks different from what we had before, but we can recognize some familiar components. The algorithm starts by creating a set of λ random points in the search space. We can use the nullary search operator for this. So this step is very similar to what EAs do (see Section 3.4). From these λ points, the set P_μ of the μ best ones are selected – again, very similar to EAs. However, EDAs do *not* apply unary or binary search operations to these points to obtain an offspring generation. Instead, they condense the information from the selected points into a model M, which usually is a stochastic distribution. This can be done by estimating the parameters of a stochastic distribution from all points in P_μ. We then can create λ new points by sampling them from this parameterized distribution M.

3.8.1.1 An Example

Maybe it is easier to understand how this algorithm works if we look at the simple example illustrated in Figure 3.33. Imagine you are supposed to find the minimum of an objective function f defined over a two-dimensional sub-space space of the real numbers, i.e., $\mathbb{Y} \subset \mathbb{R}^2$. For the sake of simplicity, let's assume the search and solution space are the same ($\mathbb{X} = \mathbb{Y}$) and its two decision variables be $x1$

and $x2$. The objective function is illustrated in the top-left corner of Figure 3.33. It has a nice basin with good solutions, but also is a bit rugged.

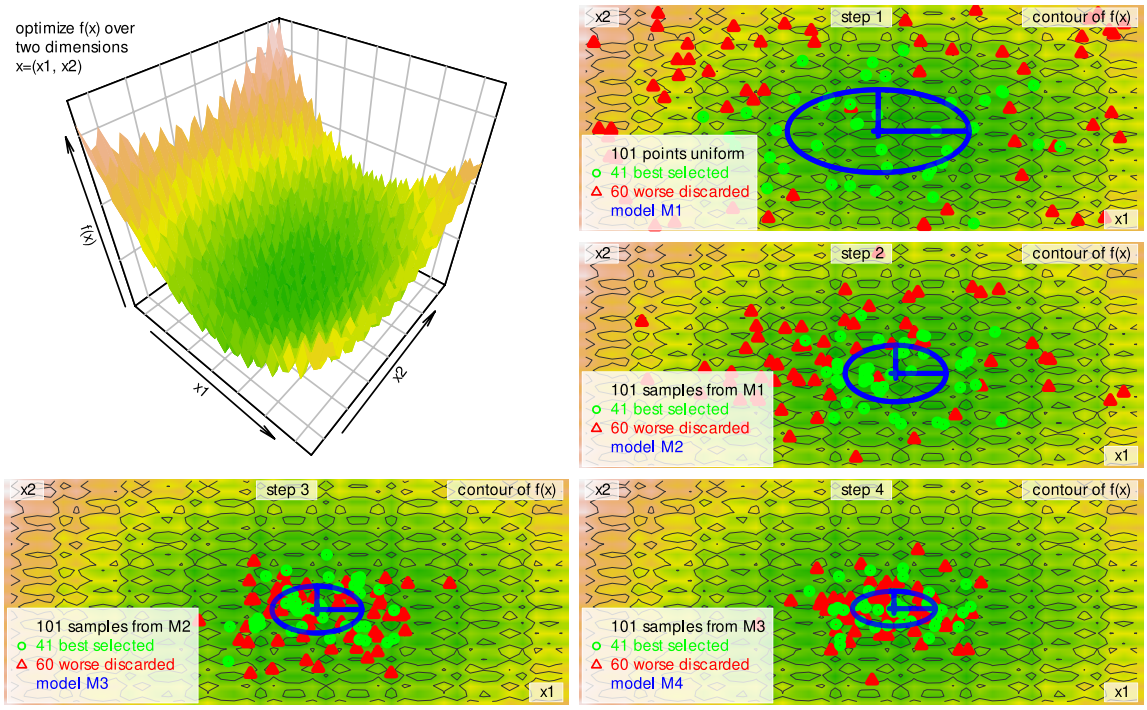

Figure 3.33: An example of how we could apply an EDA to an objective function f defined over a two-dimensional search/solution space. In each iteration $i > 1$, 101 points are sampled and the best 41 are used to derive a model M, which is defined as a normal distribution parameterized by the arithmetic means and standard devisions of the selected points along the two axes.

At the beginning of our EDA, we just generate $\lambda = 101$ points uniformly distributed in the search space (*step 3* of our algorithm). This is illustrated in the top-right sub-figure of Figure 3.33, where we only show two dimensions but put the contour of the objective function in the background. From the λ points, we keep the $\mu = 41$ ones with the best objective values (colored green) and discarded the other $101 - 41 = 60$ points (colored red). We can now compute the mean and standard deviation of the μ selected points along each axis, i.e., get four parameters. In the figures, these models are illustrated in blue. In their center, i.e., at the two mean coordinates, you can find a little blue cross. From the blue cross, one line along each axis, with the length of the corresponding standard deviation, forming the two axes of an ellipse. We can see that the ellipse is located nicely in the center of the area, where we indeed can find solutions which tend to have smaller (better) objective values.

But what can we do with this model? We can use it to generate λ new points. Since we are in the continuous space, a normal distribution lends itself for this purpose. A one-dimensional normal

distribution has the feature that it gives high probability to values close to its mean. The probability of encountering points farther away from the mean decreases quickly. If we sample a normal distribution with a given mean and standard deviation, then most points will located within one standard deviation distance from the mean. Since we have two means and two standard deviations, we could just use one independent normal distribution for each coordinate, i.e., each of the two decision variables.

In the second sub-figure from the top on the right hand side of Figure 3.33, we did exactly that. As can be seen, the points are located more closely to the center of the figure. We again select the best $\mu = 41$ points of the $\lambda = 101$ samples obtained this way and build new model based on them. The standard deviations of the model are smaller now. We repeat this process for two more steps and tend to get points more closely to the optimum of f and models which better represent this characteristic.

Of course, this was just an example. We could have chosen different probability distributions as model instead of the normal distribution. We could have updated the model in a different way, e.g., combine the previous model with the new parameters. Also we treated the two dimensions of our search space as independent, which may not be the case in many scenarios. And of course, for each search space, we may need to use a completely different type of model.

3.8.2 The Implementation

What we first need in order to implement EDAs is an interface to represent the new structural component: a model.

Listing 3.28 An excerpt of the interface of models that can be used in Estimation of Distribution Algorithms. (src)

```
1 public interface IModel<X> extends INullarySearchOperator<X> {
2   void initialize();
3   void update(Iterable<Record<X>> selected);
4   void apply(X dest, Random random);
5   default int minimumSamplesNeededForUpdate() {
6     return 1;
7   }
8
9 }
```

Listing 3.28 gives an idea how a very general interface for the required new functionality could look like. The search space $\mathbb{X}$ is represented by the generic parameter X. In our previous example, it could be equivalent the `double[2]`. The model used in our example would internally store four `double` values, namely the means and standard deviations along both dimensions.

We can update the model by passing μ samples from the search space `X` to the `update` method in form of the `Record<X>` records we already used in our implementation of EAs. The source for these samples can be any `Java` collection (all of which implement `Iterable`). In our example in the previous section, the `update` method could iterate over the `double[2]` values provided to it and compute, for both of their dimensions, the means and standard deviations.

Then, we can call `apply` λ times to sample the model and generate a new points in the search space.[3] This could be implemented for our example by generating the two normally distributed random numbers, one for each dimension, based on the means and standard deviations stored in the model.

It can be seen that this interface is rather general. We make very few assumptions about the nature of the model and how the update and sampling process will work. In order to cover models that are continuously updated and might need a certain (potentially changing) minimum number of unique samples for an update, we add two more methods: Before beginning an optimization run, the EDA should call the method `initialize` of the model. The model should then in an unbiased, initial state. Before updating the model, a call to `minimumSamplesNeededForUpdate` returns the minimum number of samples required for a meaningful update. If the number of selected individuals falls below this threshold, the algorithm could terminate or restart.

A basic EDA can now be implemented as shown in Listing 3.29. In this implementation excerpt, we have omitted the checks to `minimumSamplesNeededForUpdate` some calls to the termination criterion as well as the initialization of variables, to provide more concise and readable code.

[3] We call this method `apply` and it has the same structure as the `apply` method of the `INullarySearchOperator` interface which it overrides. This trick will come in handy later on, as it allows us to also use biased models as nullary search operator. For now, please ignore it.

Listing 3.29 An excerpt of the implementation of the Estimation of Distribution Algorithm. (src)

```
public class EDA<X, Y> extends Metaheuristic0<X, Y> {
  public void solve(IBlackBoxProcess<X, Y> process) {
// local variable initialization omitted for brevity
      M.initialize(); // initialize to uniform distribution

// first generation: fill population with random solutions
      for (int i = P.length; (--i) >= 0;) {
        X x = searchSpace.create();
        this.nullary.apply(x, random);
        P[i] = new Record<>(x, process.evaluate(x));
      }

      for (;;) { // each iteration: update model, sample model
        Arrays.sort(P, Record.BY_QUALITY);
// update model with mu<lambda best solutions
        M.update(IModel.use(P, 0, this.mu));

// sample new population
        for (Record<X> dest : P) {
          M.apply(dest.x, random); // create new solution
          dest.quality = process.evaluate(dest.x);
          if (process.shouldTerminate()) { // we return
            return; // best solution is stored in process
          }
        } // the end of the solution generation
      } // the end of the main loop
  }
}
```

3.8.3 Ingredient: A Stochastic Model for the JSSP Search Space

We now want to apply the EDA to our Job Shop Scheduling Problem. Our search space represents solutions for the JSSP as a permutation of a multi-set, where each of the n jobs occurs exactly m times, once for each machine. Unfortunately, this representation does not lend itself for stochastic modeling at all – we need a probability distribution over such permutations. It should be said that there exist clever solutions [35] for this problem, but for our introductory book, they may be too complicated.

3.8.3.1 A First and Naïve Idea

We will follow a *very* naïve approach to apply the EDA idea to the JSSP. The points in the search space are permutations with repetitions, integer vectors of length $n * m$. At each index, there could be any of the n jobs. We want to build a model by using μ such vectors. For each index $k \in 0 \ldots (n * m - 1)$,

we could simply store how often each of the jobs i occurred there in the μ permutations in $M_{k,i}$. This means our model M consists of $n*m$ vectors, each holding n numbers ranging from 0 to μ. A 0 at $M_{k,i}$ means that job i never occurred at index k in any of the μ selected points, whereas a value of μ would mean that all solutions had job i at index k.

When we sample a new point x from this model, we would process all the indices $k \in 0\ldots(n*m-1)$. The probability of putting a job $i \in 0\ldots(n-1)$ at index k into x should be roughly proportional to $M_{k,i}$. In other words, if a job i occurs often at index k in the μ selected solutions, which we used to build the model M, then it should also often occur there in λ new points we sample from M. Of course, we will need to adher to the constraint that no job can occur more then m times. While this indeed a naïve method with several shortcomings (which we will discuss later), it should work "in principle".

3.8.3.2 An Example of the Naïve Idea

We illustrate the idea of this model update and sampling process by using our demo instance from Section 2.2.2.3 in Figures 3.34 and 3.35.

The demo instance has $n = 4$ jobs that are processed on $m = 5$ machines. The points in the search space thus have $m*n = 20$ decision variables. We can build the model by considering each of the 20 decision variables separately. Their indices k range from 0 to 19.

Assume that $\mu = 10$ such points have been selected. In the upper part of Figure 3.34, we illustrate these ten points, marking the occurrences of job 0 red, of job 1 blue, of ob 2 green, and leaving those of job 3 black for clarity. The model derived from the selected points is illustrated in the middle part. It has one row for each job and one column for each decision variable index. When looking at the first decision variable (index 0), we find that job 0 occurred twice in the selected points, job 1 seven times, job 3 once, and job 2 never. This is shown in the first column of the model. The second column of the model stands for the jobs seen at index 1. Here, job 0 was never encountered, job 1 and job 2 four times, and job 3 twice. These values are obtained by simply counting how often a given job ID appears at the same index in the $\mu = 10$ selected solutions. The model can be built iteratively in about $\mathcal{O}(\mu*m*n)$ steps.

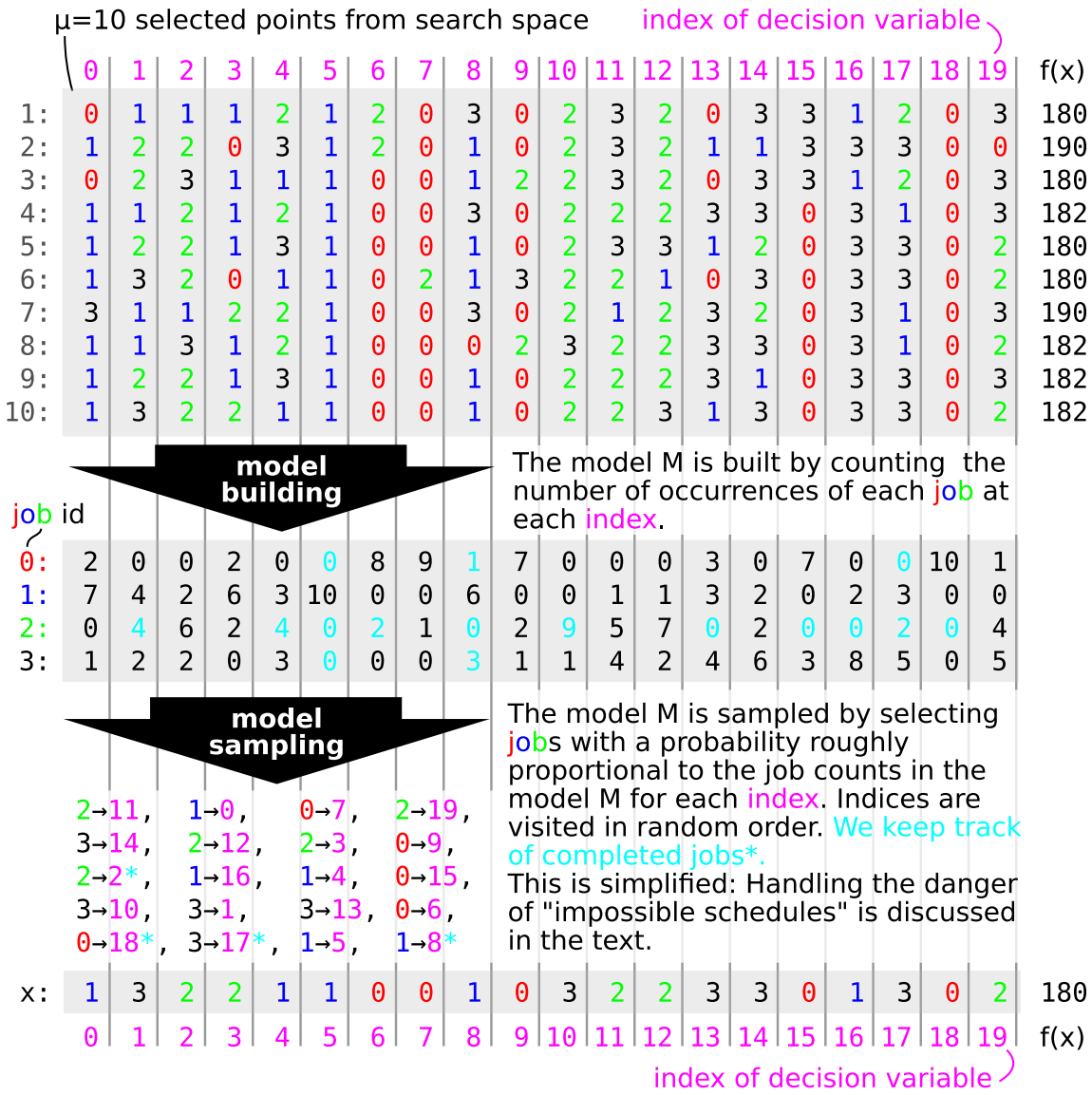

Figure 3.34: An example of how the model update and sampling in our naïve EDA could look like on the demo instance from Section 2.2.2.3; we set $\mu = 10$ and 1 new point x is sampled. See also Figure 3.35.

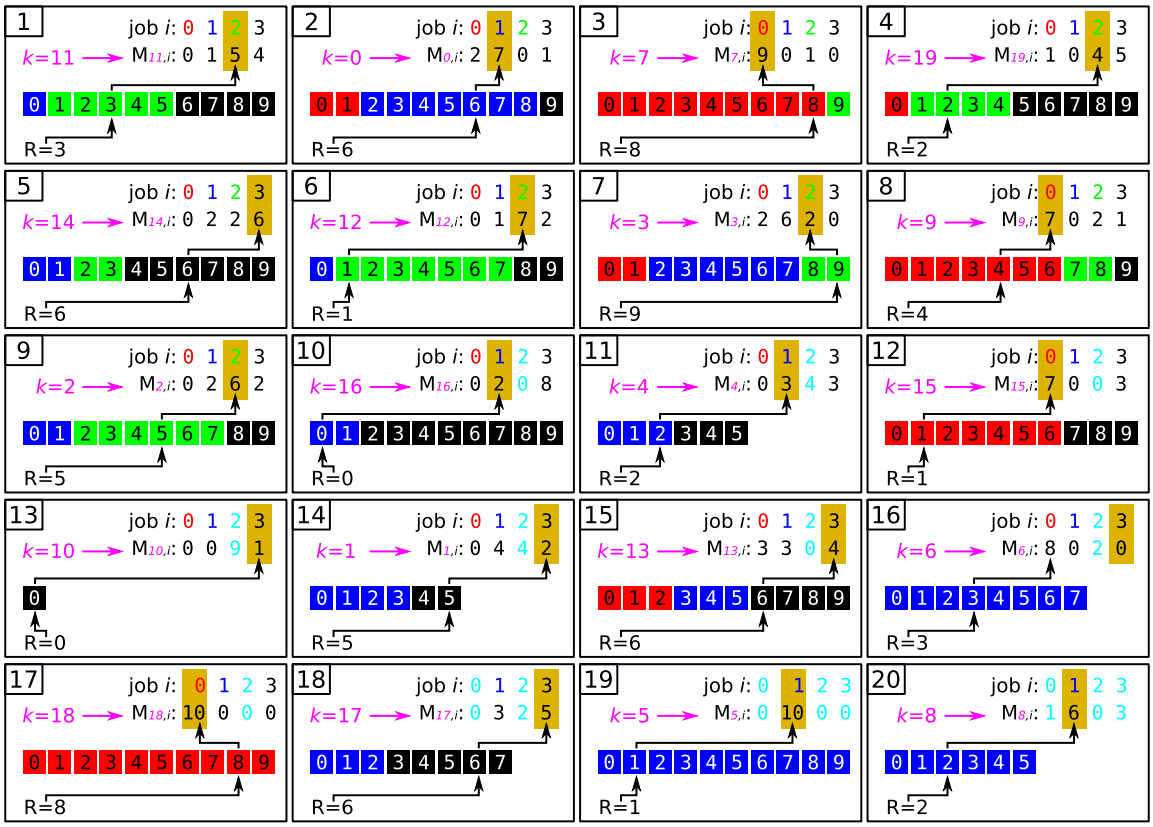

Figure 3.35: A clearer illustration of the example for sampling the model in our naïve EDA one time given in Figure 3.34.

An example for sampling one new point in the search space from this model is given in the lower part of Figure 3.34 and illustrated in complete detail in Figure 3.35. From the model which holds the frequencies of each job for each index k, we now want to sample the points of length $m * n$. In other words, based on the model, we need to decide which job to put at each index. The more often a job occurred at a given index in the μ selected points, the more likely we should place it there as well. Naturally, we could iterate over all indices from $k = 0$ to $k = (n * m - 1)$ from beginning to end. However, to increase the randomness, we process the indices in a random order.

Initially, the new point is empty. In each step of the sampling process, one index k is chosen uniformly at random from the not-yet-processed ones. In our example, we first randomly picked $k = 11$. In the model M, we have $M_{11,0} = 0$, because job 0 never occurred at index 11 in any of the $\mu = 10$ solutions used for building the model. Because job 1 was found at $k = 11$ exactly once, $M_{11}1 = 1$. Since job 2 was found at $k = 11$ 5 times, $M_{11}2 = 5$. And $M_{11}3 = 4$ because job 3 was found four times at index 11 in the selected individuals. We find that $(\sum_{i=0}^{3} M_{11}i) = 10 = \mu$. We want that the chance to sample job 0 at the here should be 0%, the chance to choose job 1 should be 10%, the chance to pick job 2

should be 50%, and, finally, job 3 should be picked with 40% probability. This can be done by drawing a random number R uniformly distributed in $0 \ldots 9$. If it happens to be 0, we pick job 1, if it is in $1 \ldots 5$, we pick job 2, and if it happens to be in $6 \ldots 9$, we pick job 3. In our example, R happened to be 3, so we set $x_{11} = 2$.

This can be implemented by writing a cumulative sum of the n frequencies into an array Q of length n. We would get $Q = (0, 1, 6, 10)$. After drawing R, we can apply binary search to find the index of the smallest number $r \in Q$ which is larger than R. Here, since $R = 3$, $r = 6$ and its zero-based index is 2, i.e., we chose job 2.

In the next step, we randomly choose k from $0 \ldots 19 \setminus 11$. We happen to obtain $k = 0$. We find $M_{0,0} = 2$, $M_{0,1} = 7$, $M_{0,2} = 0$, and $M_{0,3} = 1$. We again draw a random number R from $0 \ldots 9$, which this time happens to be 6. A value $R \in \{0, 1\}$ would have led to choosing job 0, but $R \in 2 \ldots 8$ leads us to pick job 1 for index $k = 1$ and we set $x_0 = 1$.

The process is repeated an in step 3, we randomly choose k from $0 \ldots 19 \setminus \{11, 0\}$. We happen to pick $k = 7$. Only two different jobs were found in the selected individuals, namely nine times job 0 and once job 2. We draw a random number R from $0 \ldots 9$ again and obtained $R = 6$, which leads us to set $x_7 = 0$.

We repeat this again and again. At step 9, we this way chose job 2 for the fifth time. Since there are $m = 5$ machines, this means that job 2, placed at indices 11, 2, 12, 3, and now 2, is completed and cannot be chosen anymore – regardless what the model suggests. For instance, in step 11 (see Figure 3.35), we chose $k = 4$. Job 2 did occur four times at index 4 in the selected individuals! But since we already assigned it five times, it cannot be chosen here anymore. Thus, we cannot use the probabilities 30%, 40%, and 30% for jobs 1, 2, and 3 as suggested by the model. We have to deduce the already completed job and raise the probabilities of jobs 1 and 3 to 50% each accordingly. We can still pick the job exactly as before, but instead using a random number R from $0 \ldots 9$, we use one from $0 \ldots 5$ (because $3 + 3 = 6$). As it turned out randomly to be $R = 2$, it falls in the interval $0 \ldots 2$ where we would choose job 1. Hence, we set $x_4 = 1$.

We continue this process and complete assigning job 0 in step 17, after which only either job 1 or job 3 remain as choices. Job 3 is assigned for the fifth time in step 18, after which only job 1 remains. After twenty steps, the sampling is complete. The resulting point $x \in \mathbb{X}$ is shown at the bottom of Figure 3.34.

Of course, this was just one concrete example. Every time we sample a new point x in the search space $\mathbb{X}$ using our model M, the indices k and numbers R would be drawn randomly and probably would be very different. But each time, we could hope to obtain results at least somewhat similar but yet slightly different from the μ points that we have selected.

The complexity of the model sampling is $\mathcal{O}(n * m * (n + \ln n))$: For each of the $n * m*$ indices k into the new point x, we need to add up the n frequencies stored in the model and then draw the random

number R (which can be done in $\mathcal{O}(1)$) and find the corresponding job index via binary search (which takes $\mathcal{O}(\ln n)$).

3.8.3.3 Shortcomings of the Naïve Idea

At first glance, it looks as if our approach might be a viable method to build and sample a model for our JSSP scenario. But we are unlucky: There is are two major shortcomings.

First, we might get into a situation where we have to pick a job which should have probability 0 in our model, because all other jobs have already been assigned! Let's take a look at index $k = 5$ in Figure 3.34. Here, $M_{5,1} = 10$, whereas the measured frequency of all other jobs is zero. In other words, our model prescribes that job 1 must be placed at index 5 into the new point x, regardless of whatever other choice we made during the sampling process. However, since we proceed randomly, it is entirely possible, however, that index $k = 5$ is drawn later during the sampling process and job 1 has already been assigned five times.

This situation can always occur if one of the values in M for a given index k is 0. We may always end up in a situation where we cannot finish sampling the new point because of it.

Getting a 0 at an index k for a job i in the model M also would mean that we *never* place job i again at this index in any future iterations of our algorithm. The EDA concept prescribes and alternation between building the model from the μ selected solutions, then sampling a new set of λ solutions and choosing the $\mu < \lambda$ best of them, and then building the model again from these. If job i is not placed at index k during the sampling process (for whatever reason), then it cannot have a non-zero probability in the next model. Thus, it would again not be placed at index k – and this option would have disappeared forever in the optimization process.

We can combat this problem by changing our model building process a bit. When counting, the frequencies of the jobs for a given index k in the μ selected points, we do not start at 0 but at 1. This way, each job will always have non-zero probability. Of course, for a small value of μ, this would skew our distributions quite a bit towards randomness. The solution is to not add 1 for each encounter of a job, but a very large number, say 1'000'000.

At index $k = 5$, we would then have $M_{5,1} = 10 * 1'000'000$ and $M_{5,i} = 1$ for $i \in \{0, 2, 3\}$. In other words, all jobs have non-zero probability, but if we can place job 1 at index $k = 5$, then we will most likely do so.

The model sampling does not need to be changed in any way: For each index k, we first sum up all the n numbers in the model and obtain a number Z. For index $k = 5$, we would obtain $Z = 1'000'003$. Then we sample a random number R uniformly distributed from $0 \ldots Z - 1$. At index 5, if $R = 0$, we would pick job 0. If $R \in 1 \ldots 1'000'000$, we would pick job 1. If $R = 1'000'001$, we pick job 2 and if

$R = 1'000'002$, we pick job 3. The binary search does not take any longer if the numbers we search are larger, so the problem is solved without causing any harm.

The situation remains that we cannot perfectly faithfully model and sample the selected points. When approaching the end of the sampling of one new point, we are always likely to deviate from the observed job probabilities. By randomizing the order in which we visit the indices k, which we already do, we try to at least distribute this "unfaithfulness" evenly over the whole length of the solutions in average.

The second big problem with this naïve idea is that it does not distinguish between, e.g., the first and the second occurrence of a job in the solutions we use to update the model from. For instance, in Figure 3.34, we have two solutions that contain the first occurrence of job 0 at index 0 (namely solutions 1 and 3) and two solutions (2 and 6) that contain it at index 3. There is no solution that contains the second occurrence of job 0 at index 3. Yet, if we sample the model, we might well choose index 0 for the first and index 3 for the second occurrence of job 0. In other words, even if we could faithfully sample our model, we might still arrive at solutions that are completely different from those that we used to build it. Darn.

The reader will understand that this chapter is already somewhat complex and we will have to leave it at this naïve approach. As stated before, better models and methods exists, e.g., in [35]. The focus of the book, however, is to learn about different algorithms by attacking a problem with them in a more or less ad-hoc way, i.e., by doing what seems to be a reasonable first approach. The idea proposed here, to me, seems to be something like that.

3.8.3.4 Implementation of the Naïve Idea

We can now implement our model and we do this in the class `JSSPUMDAModel`. We call it UMDA model because it the probability of choosing one value for a given decision variable assigned by the model only depends on the values of only that variable in the solutions used for building the model – and the first EDA doing something like that was the Univariate Marginal Distribution Algorithm (UMDA) [151,152]. The model can be stored in a two-dimensional array of type `long[n*m][n]`. Here, we discuss the code for model building and model sampling in Listings 3.30 and 3.31, respectively.

We realize the model building by implementing the routine `IModel.update` (see Listing 3.28) in Listing 3.30. We first initialize the complete model matrix M to all 1 values. We then process each selected point in the search space from beginning to end. If a job is encountered at an index k, we add a value `this.m_base` to the corresponding cell of the matrix. While allowing `this.base` to be set as a configuration parameter, we use `Integer.MAX_VALUE` by default. This means that jobs not encountered at a certain index k in the selected individuals will only be placed there during the sampling process if all other jobs have already been scheduled m times.

Listing 3.30 The building process of our naïve model for the JSSP in EDAs. (src)

```
1   public void update(Iterable<Record<int[]>> selected) {
2     int l = this.mModel.length; // == m*n
3
4 // Make sure that all values are >= 1
5     for (long[] a : this.mModel) {
6       Arrays.fill(a, 1L);
7     }
8
9 // For each encountered job, add the large value this.base
10    for (Record<int[]> ind : selected) { // selected
11      int[] sel = ind.x;
12      for (int k = l; (--k) >= 0;) { // valid indices
13        this.mModel[k][sel[k]] += this.base;
14      }
15    }
16  }
```

The routine `IModel.apply` is implemented in Listing 3.31. It starts by picking the full set of jobs and permitting m occurrences for each. It then shuffles the array of indices. Processing this array from front to end then means picking all values for k in a random order. For each index k, it fills an array N with the cumulative sum of the (modified) encounter frequencies of the jobs that are not finished. The random number R is then drawn and its location in N is determined. From this, we know the selected job. The job is placed into the new solution and the number of remaining times it can be placed is reduced. If the number reaches 0, the job is removed from the set of selectable jobs. This is repeated until the destination point is completed. This implements the process discussed in the previous section.

Listing 3.31 The sampling process of our naïve model for the JSSP in EDAs. (src)

```
  public void apply(int[] dest,
      Random random) {
    int[] perm = this.mPerm; // all indices
// each job occurs m times
    int[] jobRemainingTimes = this.mJobRemainingTimes;
    Arrays.fill(jobRemainingTimes, this.mMachines);
// the jobs we can choose from:
    int[] jobChooseFrom = this.mJobChoseFrom;
    long[] prob = this.mProb; // used for cumulative sum
    long[][] model = this.mModel; // the model
// we can choose from n jobs
    int jobChooseLength = jobChooseFrom.length; // = n

// permute the indexes for which we pick jobs
    RandomUtils.shuffle(random, perm, 0, perm.length);

// iterate over the indices into the array (in random order)
    for (int k : perm) {
      long N = 0L;

// build the cumulative frequency vector, N be the overall sum
      for (int j = 0; j < jobChooseLength; ++j) {
        N += model[k][jobChooseFrom[j]];
        prob[j] = N;
      }

// pick index with probability proportional to cumulative sum via
// modified binary search.
      int select = JSSPUMDAModel.find(
          RandomUtils.uniformFrom0ToNminus1(random, N), prob,
          jobChooseLength);

      int job = jobChooseFrom[select]; // get selected job
      dest[k] = job; // store job in result
      if ((--jobRemainingTimes[job]) == 0) { // job completed?
        jobChooseFrom[select] = jobChooseFrom[--jobChooseLength];
        jobChooseFrom[jobChooseLength] = job;
      }
    } // end iteration over array indices
  }
```

3.8.4 The Right Setup

We again do a similar setup experiment as we did for the EA in Section 3.4.1.2 to configure our EDA. The setup umda_3_32768, which, in each iteration, samples $\lambda = 32768$ new solutions and uses the $\mu = 3$ best of them to update the model, seems to yield the best performance in average. We also try apply clearing in the objective space discussed in Section 3.4.5 in the EDA. Clearing removes candidate solutions with duplicate makespan value before the model update. This is done by simply applying the routine specified in Listing 3.17 before the selection step into our algorithm. The implementation of the `EDAWithClearing` thus is almost the same as the basic EDA implementation in Listing 3.29 and both are given in the online repository. After another setup experiment, we identify the best setup of our UMDA-EDA with clearing for the JSSP. In each iteration, it samples $\lambda = 64$ new candidate solutions and keeps the $\mu = 2$ unique best of them. We will call this setup umdac_2+64.

3.8.5 Results on the JSSP

In Table 3.19, we compare the performance on the JSSP of both EDA variants to the best stochastic hill climber with restarts, namely hcr_65536_nswap. We can find that the UMDA without clearing is generally worse than the hill climber, while the umdac_2+64 with clearing can perform somewhat better on abz7 and yn4. On swv15, both algorithms perform particularly badly. The performance of our adaptation of the EDA concept towards the JSSP is not very satisfying.

Table 3.19: The results of the EDAs umda_3_32768 (without clearing) and umdac_2+64 (with clearing) in comparison to hcr_65536_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_65536_nswap	712	731	732	**6**	96s	21’189’358
		umda_3_32768	713	742	742	14	**93s**	**1’901’332**
		umdac_2+64	**701**	**727**	**726**	15	141s	2’860’265
la24	935	hcr_65536_nswap	**942**	**973**	**974**	**8**	71s	31’466’420
		umda_3_32768	982	1020	1018	20	**15s**	**659’470**
		umdac_2+64	956	996	991	23	70s	3’021’933
swv15	2885	hcr_65536_nswap	**3740**	**3818**	**3826**	35	**89s**	10’783’296

$\mathcal{I}$	$\mathrm{lb}f$	setup	best	mean	med	sd	med(t)	med(FEs)
		umda_3_32768	4117	4306	4318	62	156s	1'481'110
		umdac_2+64	4480	4606	4609	**35**	94s	**859'250**
yn4	929	hcr_65536_nswap	1068	1109	1110	**12**	**78s**	18'756'636
		umda_3_32768	1071	1148	1151	37	158s	2'488'518
		umdac_2+64	**1044**	**1096**	**1092**	26	152s	**2'143'989**

In Table 3.19 we make an interesting observation: It seems that the EDAs have a much lower median number of FEs *med(FEs)* until discovering their end result compared to the hill climber, while the time *med(t)* they need for these FEs does not tend to be lower at all. The time that our EDA needs to create and evaluate one candidate solution, to perform one objective function evaluation (one FE), is higher compared to the hill climber. For instance the 1'901'332 FEs within 93 seconds of umda_3_32768 on abz7 equal roughly 20'450 FEs/s, whereas the hill climber hcr_65536_nswap can generate 21'189'358 candidate solutions within 96 seconds, i.e., achieves 220'700 FEs/s on the same problem, which is roughly ten times as much. On swv15, the hill climber converges to its final result within 89 seconds, during which it performs 10'783'296 FEs, i.e., 12'1000 FEs/s. Here, umdac_2+64 is 13 times slower and performs 859'250 FEs in 94 seconds, i.e., 9140 FEs/s.

Let us therefore compare two perspectives on the progress that EDAs make with what our EAs are doing. In Figure 3.36, we plot the progress over (log-scaled) time in milliseconds as we did before. This perspective fits to our goal, to obtain good solutions for the JSSP within three minutes. The umda_32768 behaves somewhat similar to the ea_32768_nswap, which also creates $\lambda = 32768$ new solutions in each generations, but is generally slower and finishes at worse results. The gap between umdac_2+64 and eac_4+5%_nswap, which both apply clearing, is much wider.

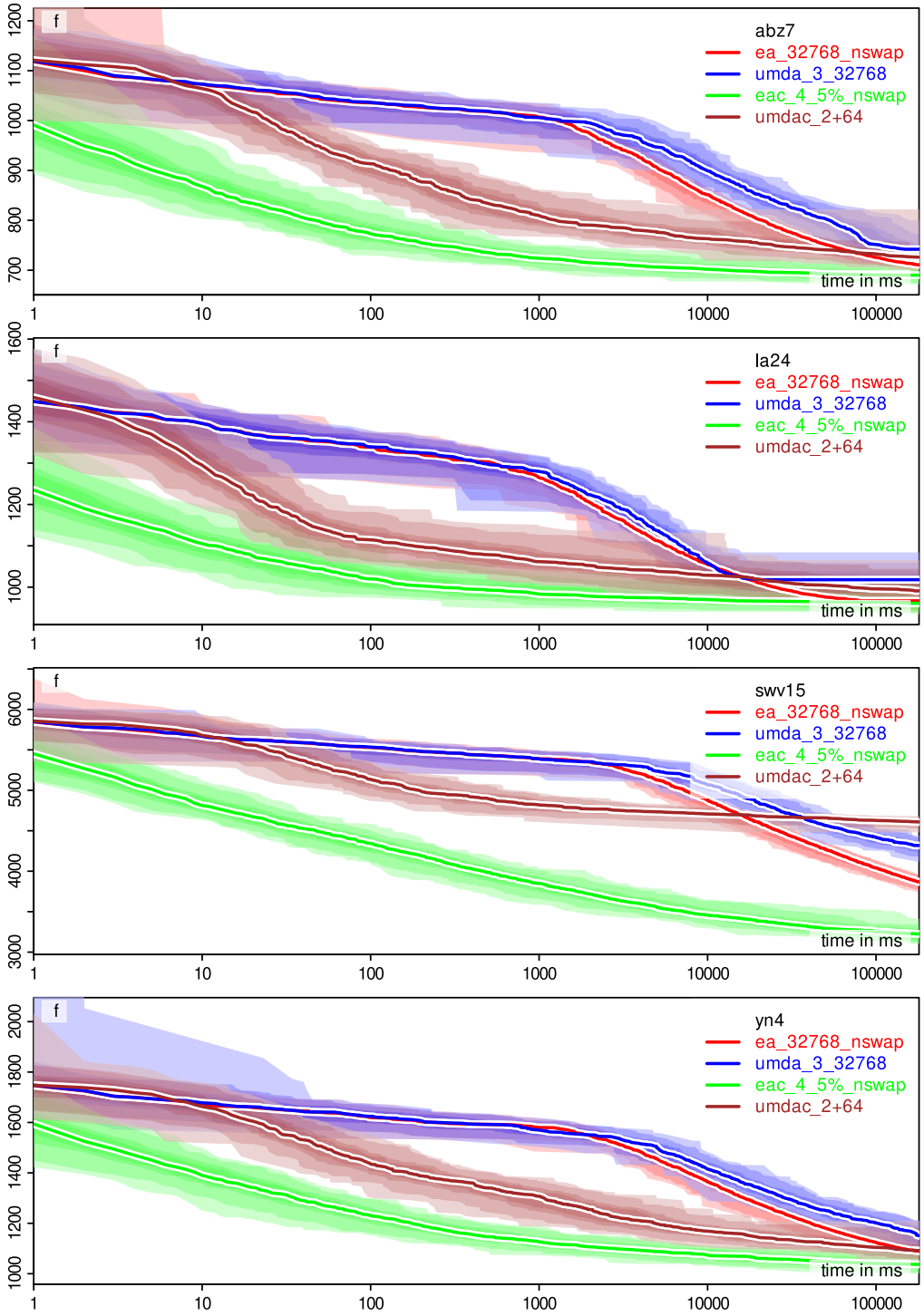

Figure 3.36: The median of the progress of the ea_32768_nswap, eac_4+5%_nswap, umda_32768, and umdac_2+64 algorithms over **time**, i.e., the current best solution found by each of the 101 runs at each point of time (over a logarithmically scaled time axis). The color of the areas is more intense if more runs fall in a given area.

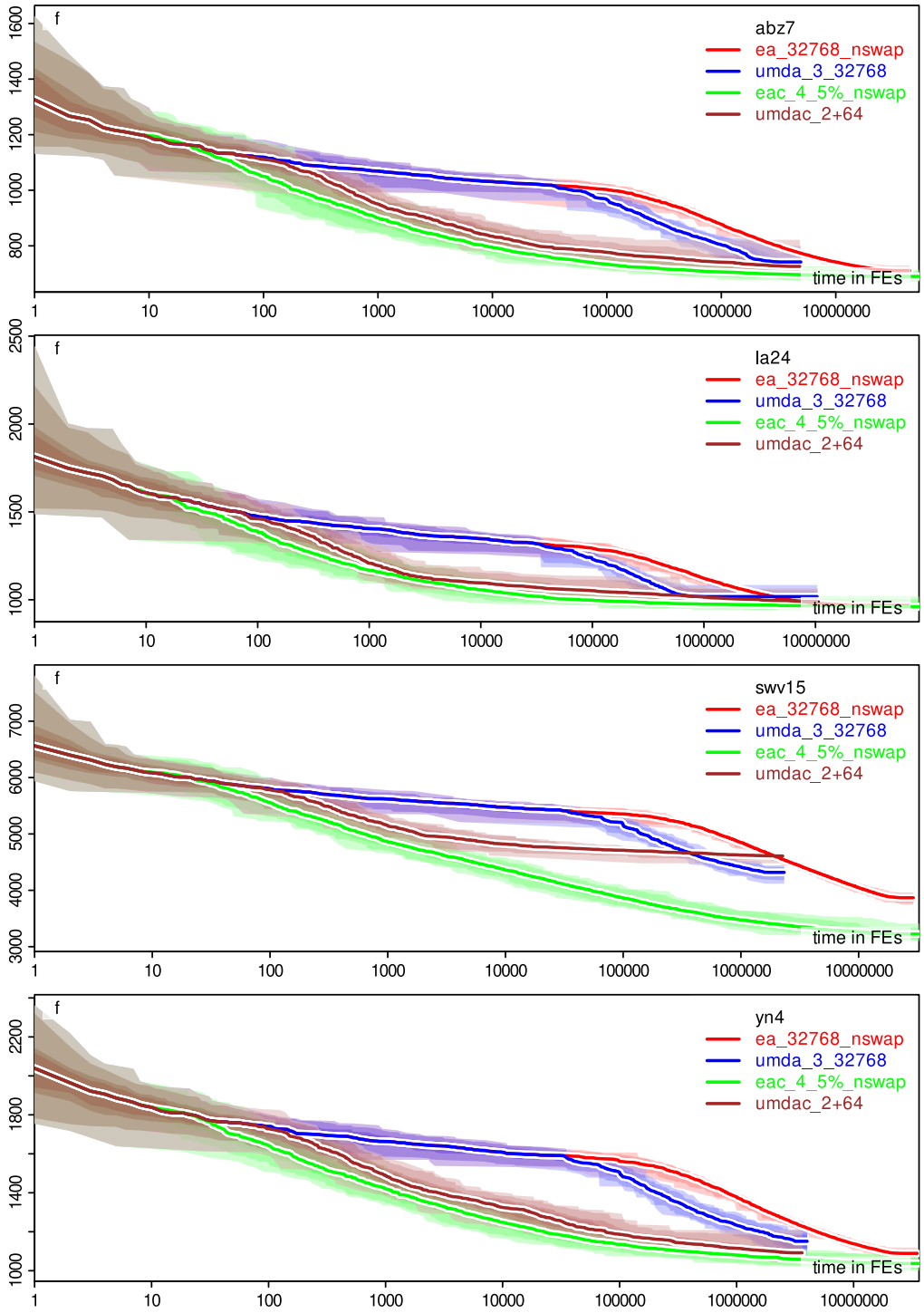

Figure 3.37: The median of the progress of the `ea_32768_nswap`, `eac_4+5%_nswap`, `umda_32768`, and `umdac_2+64` algorithms over **the consumed FEs**, i.e., the current best solution found by each of the 101 runs at each point of FE (over a logarithmically scaled time FE). The color of the areas is more intense if more runs fall in a given area.

Now we take a look at the same diagrams, but instead of the actual consumed clock time, we use the number of generated solutions as horizontal time axis. This basically shows us the progress that the algorithms make per call to the objective function, i.e., per FE. This perspective is very useful in many theoretical scenarios and also justified in scenarios where the FEs take a very long time (see later in Section 4.2.2).

Figure 3.37 shows a very different picture. `umda_32768` actually has a better median solution quality than `ea_32768_nswap`, but it stops much earlier and then is overtaken. Only on `la24`, the plain EA overtakes the plain EDA before the EDA stops. With the exception of instance `swv15`, the gap between `umdac_2+64` and `eac_4+5%_nswap` also becomes smaller.

Of course we cannot really know whether the EDA would eventually reach better solutions than the EA if it perform the same number of FEs. We could find this out with more experiments, maybe with runtime limits of 30 minutes instead of three. We will not do this here, as our scenario has a hard time limit.

What we found out is still interesting: Even the trivial, naïve model for the JSSP seems to "work," despite its shortcomings. The biggest problem here seems to be the algorithmic time complexity of the model sampling process. The `1swap` or `nswap` operators in the EA copy the original point in the search space $\mathbb{X}$ and then swap one or multiple jobs. To generate a new point in $\mathbb{X}$, they thus require a number of algorithm steps roughly proportional to $n * m$. As discussed at the end of Section 3.8.3.2, our UMDA model needs $\mathcal{O}(n * m * (n + \ln n))$ steps for this. The higher complexity of sampling the search space here clearly shows.

3.8.6 Summary

In this chapter, we have discussed the concept of Estimation of Distribution Algorithms (EDAs): the idea of learning statistical models of good solutions. Models can be probability distributions, which assign higher probability densities to areas where good previously observed solutions were located. In each "generation", we can sample λ points in the search space using this model/distribution and then use the best μ of these samples to update the model. What we hopefully gain are two things: better solutions, but also a better model, i.e., an abstract representation of what good solutions look like in general.

This concept lends itself to many domains. Already in high school we learned probability distributions over real numbers. It is very straightforward to adapt the idea of EDAs to subsets of the n-dimensional Euclidean space, which we did as introductory example in Figure 3.33. The concept also sends itself if the our search space are vectors of bits, which is the domain where the original Univariate Marginal Distribution Algorithm (UMDA) [151,152] was applied.

Unfortunately for the author of the book, probability distributions over permutations are a much harder nut to crack. As mentioned before, we tried a very naïve approach to this with several flaws – but we got it to work. Some of the flaws of our approach could also be fixed: The fact that the model does not distinguish between the first and second occurrence of a job ID can be fixed by using unique operation IDs instead of job IDs, i.e., using $m * n$ unique values for each location in the model instead of only n ones. I tried this out, but it does not lead to tangibly better results within our three minute budget. Most likely because it makes the sampling of new solutions even slower and thus decreases the total number of search steps we can do in total even more…

And this brings us two unexpected lessons to learn from this section: First, algorithmic time complexity does matter. Well, every computer scientist knows this. Professors do not bother undergraduate students with this topic for fun. But here we are reminded that efficiently implementing algorithms is important, also in the field of randomized metaheuristics. If we were to invent a very strong optimization approach but would not be able to implement it with low complexity, then it could be infeasible in practice.

The second lesson is that comparing algorithm performance using FEs could yield different results from comparing them based on clock time. Both have different advantages which we discuss in detail in Section 4.2.2, but we need to always be clear which one is more important in our scenario. In our scenario here, clock time is more important.

So was this all what is to say about EDAs? No. The EDAs we discussed here are very simple. There are at least two aspects that we did not cover here:

One aspect that we did not discuss here is that our model M is actually not *updated* but *overwritten* in each generation. Instead, we could also combine the new information M_{new} with the current model M_{old} the model M in. The UMDA implementation in [152] and the equivalent Population-based Incremental Learning Algorithm (PBIL) [13,14], for instance, do this by simply setting $M = \delta M_{new} + (1-\delta)M_{old}$, where $\delta \in (0, 1]$ is a learning rate. Another interesting approach to such iterative learning is the Compact Genetic Algorithm (cGA) [102].

The second aspect is that we treated all decision variables as if they were independent in our model. This is where the *univariate* in UMDA comes from. However, variables are hardly independent in practical problems (see also Section 5.5). For instance, the decision whether I should *next* put job 1 on machine 2 or job 3 in the JSSP will likely depend on which job I assigned to which machine *just now*. Such inter-dependencies between decision variables can be represented by *multivariate* distributions. Examples for algorithms trying to construct such models are the Bayesian Optimization Algorithm (BOA) [159,160] for bit strings. The Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [11,99] learns the relationship between pairs of variables for continuous problems in the Euclidean space. It is basically the state-of-the-art for complicated numerical optimization problems.

3.9 Ant Colony Optimization

Ant Colony Optimization (ACO) [62–65] is a family of metaheuristics, which can be considered EDAs with two special characteristics:

1. They are designed for problems whose solutions can be expressed as paths x in graphs G.
2. During the model sampling (i.e., the creation of a path x), they make use of additional heuristic information obtained from the problem instance.

You can imagine a graph as a mathematical abstraction that can represent a street map, a computer network, a power network, a network of water pipes, and such and such. More formally, graph $G = (V, E)$ is a structure described by a set V of v vertices, which we will here identify by integers IDs, i.e., $V = 0 \dots v-1\}$. $E \subseteq V \times V$ is the set of edges connecting the vertices. A path x through the graph G is a sequence $(x_0, x_1, \dots, x_l)$ such that $x_i \in V$ for $\forall i \in 0 \dots (l-1)$. The vertexes at index i and $i+1$ in x for $i \in 0 \dots l-2$ form the edges of the path, i.e., (x_i, x_{i+1}) are elements of E. The search space $\mathbb{X}$ of ACO corresponds to the set of all paths $x \in \mathbb{X}$ admissible by the problem instance. If we represent a street map as a graph G, then a path x through G corresponds to a route on the map from a starting point to an end point.

3.9.1 The Algorithm

If we recall the structure of EDAs from Section 3.8.1, then ACO adds

1. a special model structure,
2. a special sampling method, and
3. a special model update method.

We do not require a new overall algorithm structure or new implementation – Listing 3.29 will do just fine – all we need is to understand these new concepts.

3.9.1.1 Model and Model Sampling

The model sampling in ACO works by iteratively building a path through a graph. In each step, a vertex is added to the path until no nodes can be added anymore and the path x is completed. The choice which vertex to add is probabilistic. It depends on two factors: the model M and the heuristic information H.

Definition 30. The model M used in ACO is a two-dimensional table called *pheromone matrix* and the values stored in it are called *pheromones*.

Each row $i \in 0 \dots (v-1)$ stands for one of the vertices and holds a value $M_{i,j}$ for each other vertex $j \in 0 \dots (v-1)$ with $j \neq i$. The higher the value $M_{i,j}$, the more likely should it be that the edge (i,j) is added to the path, i.e., that vertex j follows after vertex i.

Besides the model M, ACO also uses one heuristic value $H_{i,j}$ for each such edge. This value reflects a "gain" that can be obtained by adding the edge. As in the case of $M_{i,j}$, larger values of $H_{i,j}$ make adding the edge (i,j) to the path more likely.

The first vertex at which the path x is started at iteration index $i = 0$ is either chosen completely randomly, fixed, or based on some policy. In each following step $i > 0$ of the model sampling, the algorithm first determines the set N of vertices that can be added. Often, these are all the vertices not yet present in x, i.e., $V \setminus x$. In this case, we can initially set $N = V \setminus x_0$ and iteratively remove every vertex added to x from N. But there may also be other scenarios, for instance, if we navigate through a graph where not all vertexes are directly connected.
Either way, once N has been determined, the choice about which edge to add to x in iteration $i > 0$ is made probabilistically. Assume that N contains v' vertices, then the probability $P(\text{add } N_j)$ to add vertex N_j to x is:

$$P(\text{add } N_j) = \frac{\left(M_{x_{i-1},N_j}\right)^{\alpha} * \left(H_{x_{i-1},N_j}\right)^{\beta}}{\sum_{k=0}^{v'-1} \left(M_{x_{i-1},N_k}\right)^{\alpha} * \left(H_{x_{i-1},N_k}\right)^{\beta}} \tag{3.5}$$

Here, $\alpha > 0$ and $\beta > 0$ are two configuration parameters which weight the impact of the model M and the heuristic information H, respectively. The higher the model and heuristic values for the edge from the last-added vertex x_i to a potential next vertex N_j, the more likely it is selected. The model sampling then works as follows:

1. The point to sample x will be an array of vertexes which represents the path. It is initially empty.
2. Allocate an array p of real numbers of length $v-1$.
3. Set an index variable $i = 0$.
4. Choose a first vertex and store it in x_i.
5. Repeat:
    a. Set $i = i + 1$.
    b. Obtain $N \subset V$ as the set of vertices that can potentially be added to x in this round. (N is usually equivalent to V minus all vertices already added.)
    c. The number of vertices in N be v'.
    d. If $v' = 0$, exit the loop, because the path construction is complete.
    e. If $v' = 1$, set $k = 0$, because there is only one vertex to choose from.
    f. else, i.e., if $v' > 1$, do:

i. Set a real variable ps to 0.
ii. For j from 0 to $v'-1$ do: A. Set $ps = ps + \left(M_{x_{i-1},N_j}\right)^{\alpha} * \left(H_{x_{i-1},N_j}\right)^{\beta}$. B. Set $p_j = ps$.
iii. Draw a random number r uniformly distributed in $[0, ps)$.
iv. Determine k to be the index of the smallest value in p which is greater than r. It can be found via binary search (we may need to check for smaller values left-wards if model or heuristic values can be 0).

g. Set $x_i = N_k$, i.e., append the vertex to x.

6. Return the completed path x.

Definition 31. A completed path x sampled from the model in ACO is called an *ant*.

We implement Equation (3.5) in *lines 5.e* and *5.f*. Obviously, if there is only $v' = 1$ node that could be added, then it will have probability 1 and we do not need to actually compute the equation. If $v' > 1$, then we need to compute the product P_j of the model value $M^{\alpha}_{x_{i-1},N_j}$ and heuristic value $H^{\beta}_{x_{i-1},N_j}$ for each vertex. The probability for each vertex to be chosen would then be their corresponding result divided by the overall sum of all of these values. *Lines 5.f.i* to *5.f.iv* show how this can be done efficiently: Instead of assigning the results directly to the vertices, we use a running sum ps instead. Thus, the value p_0 is P_0, $p_1 = P_0 + P_1$, $p_2 = P_0 + P_1 + P_2$, and so on. Finally, we just need to draw a random number r from $[0, ps)$. If it is less than $p_0 = P_0$, then we choose vertex N_0. Otherwise, if it is less than $p_1 = P_0 + P_1$, we pick N_1. Otherwise, if it is less than $p_2 = P_0 + P_1 + P_2$, we pick N_2, and so on. We can speed up finding the right node by doing a binary search. (In the case that model or heuristic values can be zero, we need to be careful because we then could have some $p_\kappa = p_{\kappa+1}$ and thus would need to check that we really have the lowest index k$ for which $p_k > r$.)

If we need to add all vertexes in V, then it is relatively easy to see that this model sampling routine has quadratic complexity: For each current vertex we need to look at all other (not-yet-chosen) vertices due to *line 5.f.ii*.

3.9.1.2 Model Update

In the traditional ACO approach, the model will be sampled $\lambda > 0$ times in each iteration and $0 < \mu \leq \lambda$ of the solutions are used in the model update step. Here, we discuss the model update procedure for the first ACO algorithm, the *Ant System* (AS) [62,64] and its improved version, the *Max-Min Ant System* (MMAS) [192].

Let X be the list of the μ selected paths. Then, the model values in the matrix M are updated as follows:

$$M_{i,j} = \min\{U, \max\{L, (1-\rho)M_{i,j} + \sum_{\forall x \in X} \tau^x_{i,j}\}\} \tag{3.6}$$

All model values are limited to the real interval $[L, U]$ and L and U are its upper and lower bound, respectively. If the pheromone $M_{i,j}$ on an edge (i, j) would become 0, then its probability to be added will also be zero and j will never be added again to any x directly after i. Since we want to preserve a certain minimum probability, setting a lower limit $L > 0$ makes sense. If $M_{i,j}$ gets too large, this can have the opposite effect and then j will always be chosen directly after i. Thus, the upper limit $U > L$ is introduced.

The current model value is reduced by multiplying it with $(1 - \rho)$, where $\rho \in [0, 1]$ is called the "evaporation rate." This concept has already been mentioned in our summary on EDAs (Section 3.8.6) and is used in the PBIL [13,14] algorithm.

The amount $\tau_{i,j}^x$ added to the model values for each $x \in X$ is:

$$\tau_{i,j}^x = \begin{cases} Q/f(\gamma(x)) & \text{if edge } (i,j) \text{ appears in } x \\ 0 & \text{otherwise} \end{cases}$$

where Q is a constant. In other words, the pheromone $M_{i,j}$ of an edge (i, j) increases more if it appears in selected solutions with small objective values.

In the AS, $\mu = \lambda$ and now bounds for the pheromones are given, i.e., $L = 0$ and $U = +\infty$. In the MMAS in [192], the matrix M is initialized with the value U, $\alpha = 1$, $\beta = 2$, $\lambda = v$ (i.e., the number of vertices), $\mu = 1$, $Q = 1$. There, values of $\rho \in [0.7, 0.99]$ are investigated and the smaller values lead to slower convergence and more exploration whereas the high ρ values increase the search speed but also the chance of premature convergence.

Either way, from Equation (3.6), we know that the model update will need at least a quadratic number of algorithm steps and the model itself is also requires quadratic amount of memory, i.e., both are at least in $\mathcal{O}(()v^2)$. Both of these can be problematic for large numbers v of vertices or short time budgets.

The actual "size" of the model M and the heuristic information H depends on whether the edges in the graph are directed or not: Normally, the size is $v(v-1)$. If going from i to j always has exactly the same cost and is equivalent to going from j to i, then it is sufficient to maintain one single pheromone for both edges, i.e., $v(v-1)/2$ in total. A quadratic data structure size begins to become problematic for $v \geq 10'000$ on today's machines. In our JSSP scenario, we are far from that, but have already learned in Section 3.8.5 that the quadratic runtime of the model sampling is indeed a bottleneck.

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 [206,209]:

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 [206,209]. 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 32. 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.

There exist ideas to *mitigate* the drawback that clock times are hard to compare [120,209]. For a specific optimization problem, one can clearly specify a simple standardized algorithm $\mathbf{B}$, which always terminates in a relatively short time, say a simple heuristic. Before applying the algorithm $\mathbf{A}$

that we actually want to investigate to an instance $\mathcal{I}$ of the problem, we first apply $\mathbf{B}$ to $\mathcal{I}$ and measure the time $T[\mathbf{B}|\mathcal{I}]$ it takes on our machine. We then can divide the runtime $T[\mathbf{A}|\mathcal{I}]$ needed by $\mathbf{A}|\mathcal{I}$ by $T[\mathbf{B}|\mathcal{I}]$. We can then hope that the resulting, normalized runtime is somewhat comparable across machines. Of course, this is a problem-specific approach, it does not solve the other problems with measuring runtime directly, and it likely will still not generalize over different computer architectures or programming languages.

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 33. 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 Mathematics 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 (TSP) [8,94] 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 $\mathcal{O}(n^2)$ [209]. We observed this problem in the experiment with our EDA implementation for the JSSP in Section 3.8.5. FEs are fair time measures only if the algorithms that we compare have roughly the same time complexity per FE.
- Time measured in FEs is harder to comprehend in the context of parallelization and distribution of algorithms.

There exists an idea to *mitigate* the problem with the different per-FE complexities: counting algorithm steps in a problem-specific method with a higher resolution. In [209], for example, it was proposed to count the number of distance evaluations on the TSP and in [97], bit flips are counted on the MAX-SAT problem.

4.2.3 Do not count generations!

As discussed in Definition 24 in Section 3.4.1, a generation is one iteration of a population-based optimization method. At first glance, generations seem to be a machine-independent time measure much like FEs. However, measuring runtime in "generations" is a very bad thing to do.

In one such generation, multiple candidate solutions are generated and evaluated. How many? This depends on the population size settings, e.g., μ and λ. So generations are not comparable across different population sizes.

Even more: if we use algorithm enhancements like clearing (see Section 3.4.5), then the number of new points sampled from the search space may be different in each generation. In other words, the number

of consumed generations does not necessarily have a relationship to FEs (and neither to the actual consumed runtime). Therefore, counting FEs should always be preferred over counting generations.

4.2.4 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 $\mathbf{A}$ can solve a problem instance in ten million years while algorithm $\mathbf{B}$ only needs one million. In a practical scenario, usually neither is useful nor acceptable and the fact that $\mathbf{B}$ is better than $\mathbf{A}$ would not matter.[1]

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

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 [73,98], 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.

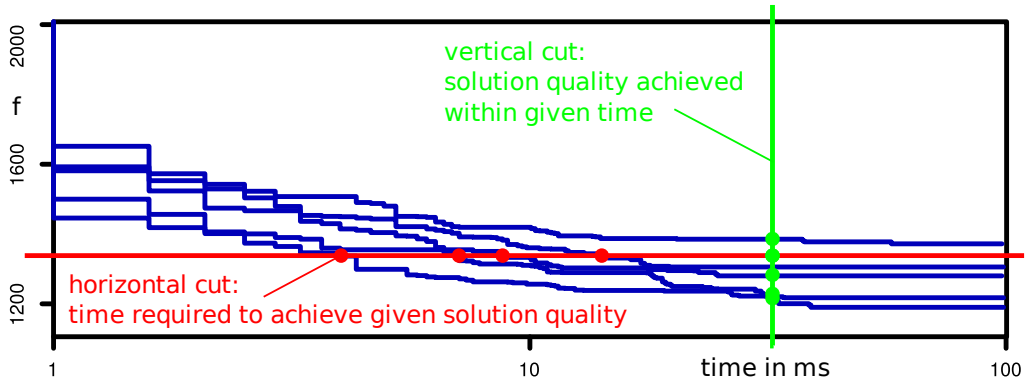

Figure 4.1: Illustration of the two basic forms to measure performance from raw data, based on a fraction of the actual experimental results illustrated in Figure 3.7 and inspired by [73,98].

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. Of course, budgets based consumed clock time are hard to compare or reproduce [119].

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, [195].

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 [73,98] 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 $\mathbf{B}$ is ten times faster than algorithm $\mathbf{A}$ [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 [73,98].

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 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:

[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).

- 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 [209,210,213], 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 34. 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 [127,172,186,194,196].

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

Definition 36. A *random sample* of length $k \geq 1$ is a set of k independent observations of an experiment following a random distribution F.

Definition 37. 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
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 38. 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.

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

Definition 39. The median $\mathrm{med}(A)$ 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.

We may now ask: Why are we considering two measures of the average, the arithmetic mean and the median? Which one is better? Which one should I take?

This question is actually hard to answer. It very much depends on your application. In particular, it depends very much on the question of whether or not your data might contain outliers and - very important – what could cause these outliers?

4.4.2.2 Outliers and Skewed Distributions

Let us consider two example data sets A and B, both with $n_A = n_B = 19$ values, only differing in their largest observation:

- $A = (1, 3, 4, 4, 4, 5, 6, 6, 6, 6, 7, 7, 9, 9, 9, 10, 11, 12, 14)$
- $B = (1, 3, 4, 4, 4, 5, 6, 6, 6, 6, 7, 7, 9, 9, 9, 10, 11, 12, 10'008)$

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 $b_{18} = 10'008$ 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.

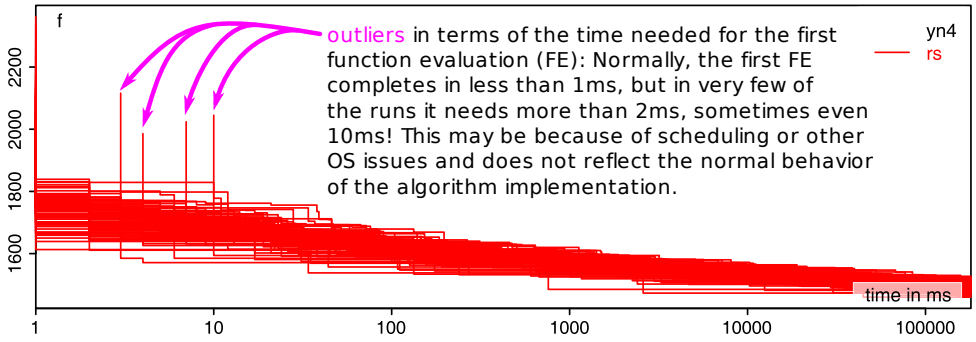

Figure 4.2: Illustrative example for outliers in our JSSP experiment: sometimes the first function evaluation takes unusually long, although this did not have an impact on the end result (clipping from Figure 3.4).

We often call such odd values outliers [93,139]. They sometimes may represent measurement errors or observations which have been been disturbed by unusual effects. In our experiments on the JSSP, for instance, some runs may perform unexpectedly few function evaluations, maybe due to scheduling issues. In Figure 4.2, this becomes visible in some cases where the first FE was delayed for some reason – while it would not be visible if somewhere during the run an unusual delay would occur. As a result, some runs might perform worse, because they receive fewer FEs.

So outliers have a big impact on the arithmetic mean and no influence on the median. What does that mean? Do we want our measure of average to be influenced by outliers or not?

This very much depends on the source of outliers. Let's say you are a biologist and want to count ants per square meters of a lawn. There can be lots of totally uncontrollable factors that influence the outcome. Maybe an ant eater has passed through and ate all the ants of one the lawn cells and then left. Such factors are probably not what you are after. You would probably want a representative value and do not care about outliers very much. Then, you prefer statistical measures, which do not suffer too much from anomalies in the data. You would prefer the median.

For example, in [170] 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. The median would show us that, for normal people, the income did not really grow significantly.

But there are also scenarios where outliers contain important information, e.g., represent some unusual side-effect in a clinical trial of a new medicine. **Actually, in our domain – optimization – outliers will very often give us very important information**! Think about it: When we measure the performance of an algorithm implementation, there are few possible sources of "measurement errors" apart from

unusual scheduling delays and even these cannot occur if we measure runtime in FEs. If there are unusually behaving runs, then the most likely source is a bug in the algorithm implementation! If the cause is not a bug, then the second most likely source is that our algorithm has a bad worst case behavior. We do want to know this. Thus, we must check the arithmetic mean.

Let us take the MAX-SAT problem, an $\mathcal{NP}$-hard problem. If we apply a local search algorithm to a set of different MAX-SAT instances, it may well be that the algorithm requires exponential runtime on 25% of them while solving the others in polynomial time [111]! This would mean that if we consider only the median runtime, it would appear that we could solve an $\mathcal{NP}$-hard problem in polynomial time, as the median is not influenced by the worst 25% of the runs… In other words, our conclusion would be quite spectacular, but also quite wrong. The arithmetic mean is much more likely to be influenced by the long runs. From it, we could see that our algorithm, in average, still needs exponential time.

In optimization, the quality of good results is limited by the quality of the global optimum. Most reasonable algorithms will give us solutions not too far from it (but obviously never anything better). In such a case, the objective function appears almost "unbounded" towards worse solutions. The real upper bound of the objective function, i.e., the worst possible objective value, will normally be very far away from what the algorithm tends to deliver. 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 distribution of the final result quality might be skewed, too. Thinking that we will most often get results similar to the arithmetic mean might then be wrong.

4.4.2.3 Summary

Both the arithmetic mean and median carry useful information. The median tells us about values we are likely to encounter if we perform an experiment once and it is robust against outliers and unusual behaviors. The mean tells us about the average performance if we perform the experiment many times. If we try to solve 1000 problem instances, the overall time we will need will probably be similar to 1000 times the average time we observed in previous experiments. It also incorporates information about odd, rarely occurring situations while the median may "ignore" phenomena even if they occur in one third of the samples. If the outcome in such situations is bad, then it is good to have this information.

Today, the median is often preferred over the mean because it is a robust statistic. Actually, I myself often preferred it in the past. The fact that skewed distributions and outliers have little impact on it makes it very attractive to report average result qualities. There is no guarantee whatsoever that a solution of mean quality exists in an experiment.

However, the weakness of the arithmetic mean, i.e., the fact that every single measured value does have an impact on it, can also be its strength: If we have a bug in our algorithm implementation that

only very rarely has an impact on the algorithm behavior and that only in these very few cases leads unexpectedly bad results, this will show up in the mean but not in the median. If our algorithm on a few problem instances needs particularly long to converge, we will see it in the mean but not in the median. *For this reason, I now find the mean to be the more important metric.*

But we do not need to decide which is better. I think there is no reason for us to limit ourselves to only one measure of the average. I suggest to report both, the median and the mean, to be on the safe side – as we did in our JSSP experiments. Indeed, the maybe best idea would be to consider both the mean and median value and then take the worst of the two. This should always provide a conservative and robust outlook on algorithm performance.

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 40. 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 41. 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 42. 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.

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 $P\left[z < \text{quantile}_q^k\right]$ 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:

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

Quantiles are a generalization of the concept of the median, in that $\text{quantile}_2^1 = \text{med} = \text{quantile}_{2i}^i \forall i > 0$. There are actually several approaches to estimate quantiles from data. The R programming language widely used in statistics applies Equation (4.2) as default [19,115]. 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 $\lfloor h \rfloor + 1$.

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

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:

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

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 $\mathbf{A}$ and $\mathbf{B}$ 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 $\mathbf{A}$ is really better than $\mathbf{B}$ or – as mentioned in Section 3.3.5.2 – by pure coincidence, as artifact from the randomness of our algorithms.

If we say "$\mathbf{A}$ is better than $\mathbf{B}$" because this is what we saw in our experiments, we have a certain probability p to be wrong. Strictly speaking, the statement "$\mathbf{A}$ is better than $\mathbf{B}$" 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 $\mathbf{A}$ from which the data sample A stems tends to produce bigger results than $\mathbf{B}$ 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.

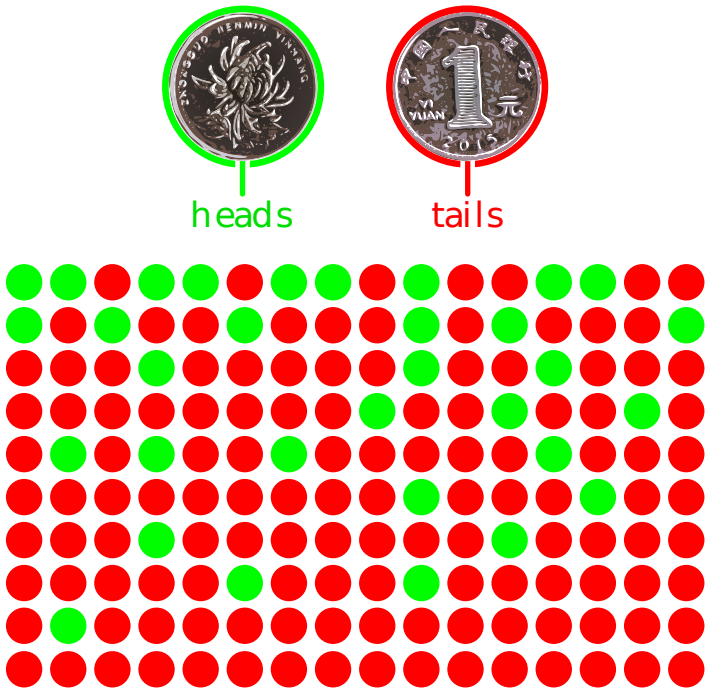

Figure 4.3: The results of our coin tossing game, where I win 128 times (red) and you only 32 times (green).

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

$$\begin{aligned} P[k \geq 128|160] &= \frac{1}{2^{160}} \sum_{k=128}^{160} \binom{n}{k} \\ &= \frac{1'538'590'628'148'134'280'316'221'828'039'113}{365'375'409'332'725'729'550'921'208'179'070'754'913'983'135'744} \\ &\approx \frac{1.539*10^{33}}{3.654*10^{47}} \\ &\approx 0.000000000000000421098571 \\ &\approx 4.211 * 10^{-15} \end{aligned}$$

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 $\mathbf{A}$ and $\mathbf{B}$ 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 $\mathbf{A}$ is better than algorithm $\mathbf{B}$." 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., $\mathbf{A} \equiv \mathbf{B}$. 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 $\mathbf{A} \equiv \mathbf{B}$. 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 $\mathbf{A}$ and $\mathbf{B}$ 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 $\mathbf{B}$ 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 $\mathbf{B}$ is truly better than $\mathbf{A}$, 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, $\mathbf{A} \equiv \mathbf{B}$. Then, this would mean that the data samples A and B stem from the *same* algorithm (as $\mathbf{A} \equiv \mathbf{B}$). 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 $\mathbf{B}$ is better than $\mathbf{A}$, 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 $\mathbf{B}$ 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 [28,69]. 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 [32] 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 [16,110,140,186],
- Fisher's Exact Test [75],
- the Sign Test [95,186],
- the Randomization Test [28,69], and
- Wilcoxon's Signed Rank Test [221].

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 $\mathbf{A}$ and $\mathbf{B}$ 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 [67]. 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.

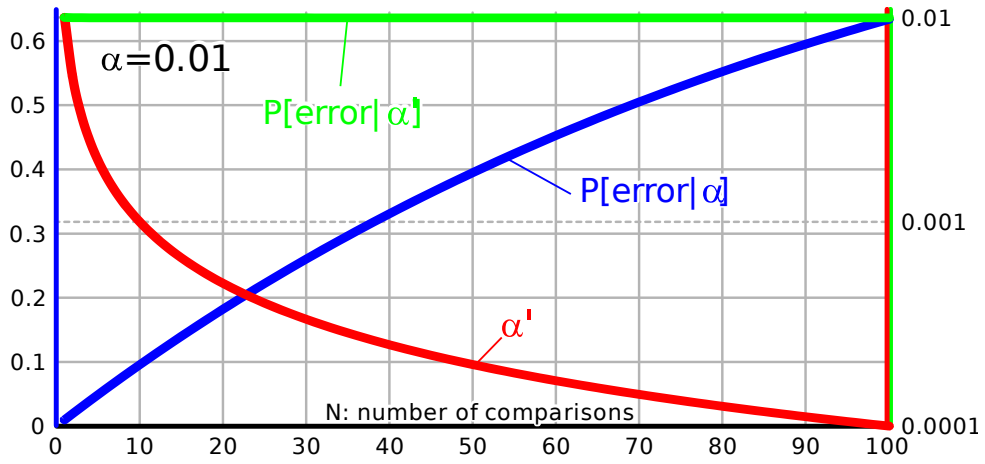

Figure 4.4: The probability $P[\text{error}|\alpha]$ of accepting at least one wrong hypothesis when applying an unchanged significance level α in N tests (left axis) versus similar – and almost constant – $P[\text{error}|\alpha']$ when using corrected value $\alpha' = \alpha/N$ instead (both right axis),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

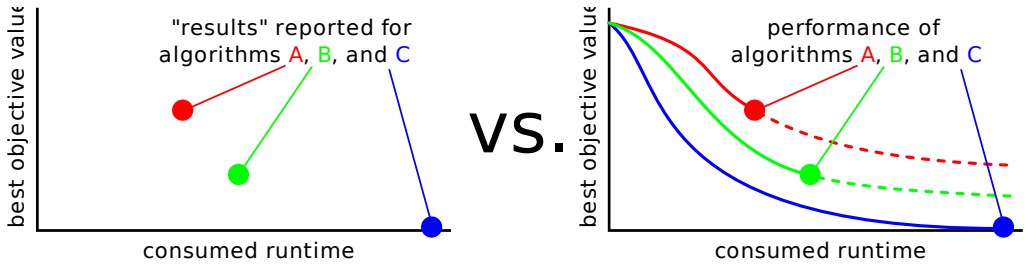

Figure 4.5: "End results" experiments with algorithms versus how the algorithms could actually have performed.

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 $\mathbf{C}$ needs a long time, but provides the best median result quality. Algorithm $\mathbf{B}$ is faster, but we pay for it by getting worse results. Finally, algorithm $\mathbf{A}$ is the fastest, but has the worst median result quality. We could conclude that, if we would have much time, we would choose algorithm $\mathbf{C}$ while for small computational budgets, algorithm $\mathbf{A}$ 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 $\mathbf{C}$ is always better than algorithm $\mathbf{B}$, which, in turn, is always better than algorithm $\mathbf{A}$. 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 [209,213,214].

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.

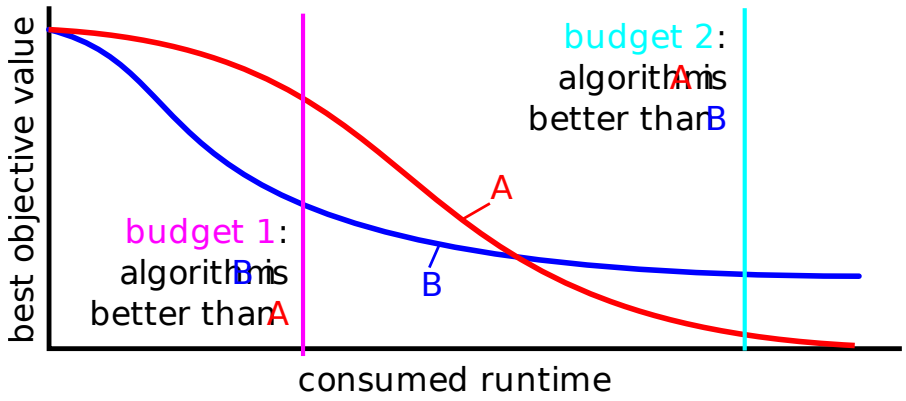

Figure 4.6: Different algorithms may perform best at different points in time.

Figure 4.6, for instance, illustrates a scenario where the best algorithm to choose depends on the available computational budget. Initially, an algorithm $\mathbf{B}$ produces the better median solution quality. Eventually, it is overtaken by another algorithm $\mathbf{A}$, 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?* [208,216]

5.1 Premature Convergence

Definition 43. 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 44. An optimization problem is multi-modal if it has more than one local optimum [53,113,167,181].

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 45. Convergence to a local optimum is called *premature convergence* ([208,216], see also Definition 23).

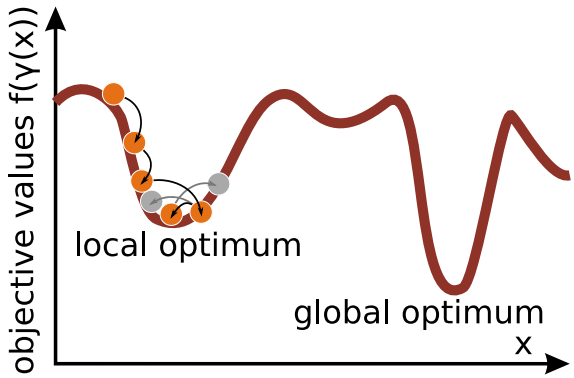

Figure 5.1: An example for how a hill climber from Section 3.3 could get trapped in a local optimum when minimizing over a one-dimensional, real-valued search space.

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 46. 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 3.4.1.4.

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 [72], 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 3.3.4.2 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 [48,184,189], including:

1. Sharing and Niching [50,109,179] 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 [161,162] takes this idea one step further and only allows the best solution within a certain radius survive. We plugged a variant of this idea with radius 0 into an Evolutionary Algorithm in Section 3.4.4 and found that it indeed improves the performance.

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 [26,85] such as Tournament selection [26,29], Ranking Selection [12,29], or the (discouraged! [26,55,218]) fitness-proportionate selection [53,85,109] 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

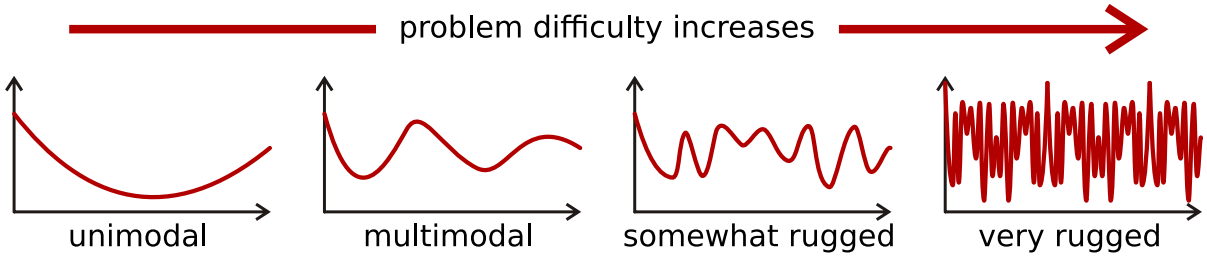

Figure 5.2: An illustration of problems exhibiting increasing ruggedness (from left to right).

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 44) 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

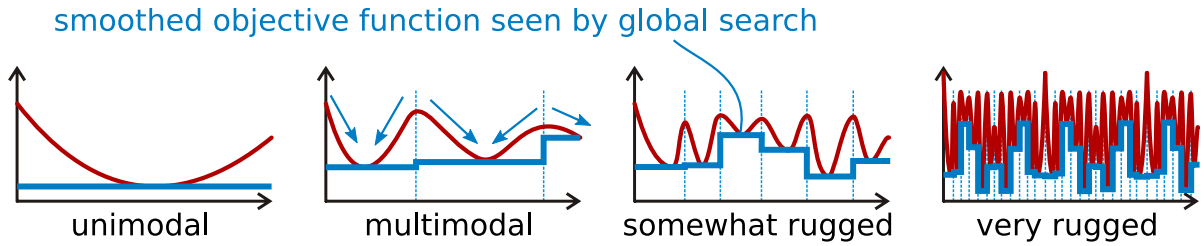

Figure 5.3: An illustration of how the objective functions from Figure 5.2 would look like from the perspective of a Memetic Algorithm: The local search traces down into local optima and the MA hence only "sees" the objective values of optima [220].

It has been suggested that combining global and local search can mitigate the effects of ruggedness to some degree [220]. 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 [92,108,220]. 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 [220].

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

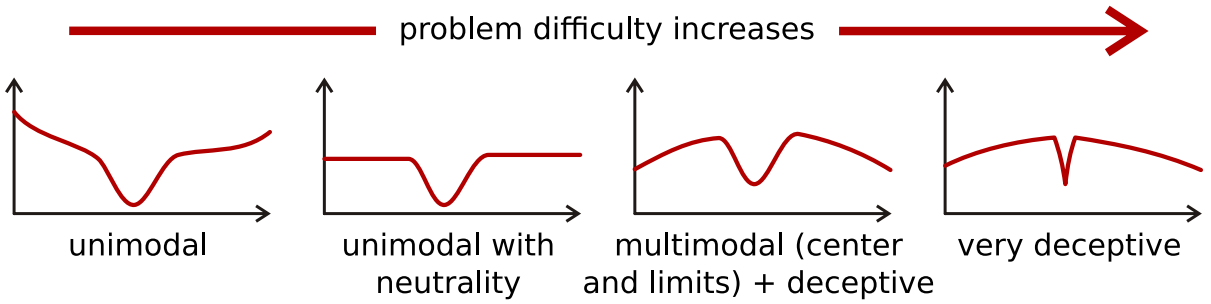

Figure 5.4: An illustration of problems exhibiting increasing deceptiveness (from left to right).

A problem is deceptive if following such a trail of improving solutions leads us away from the actual optimum [208,216]. Figure 5.4 illustrates different problems with increasing deceptiveness of the objective function.

Definition 47. 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 47 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 47 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

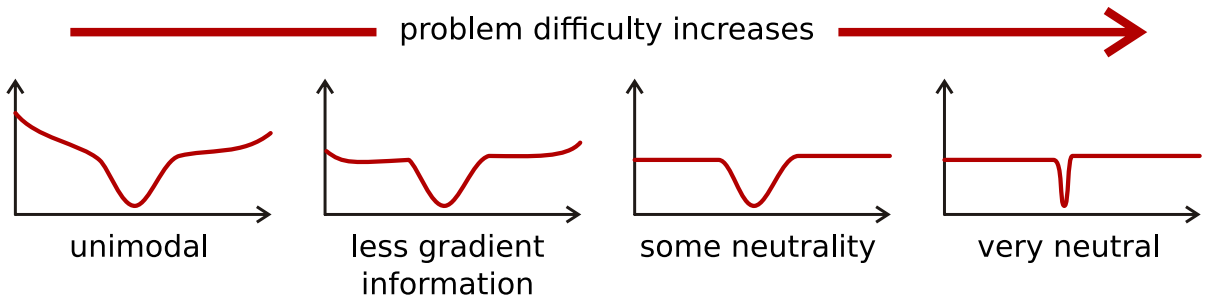

Figure 5.5: An illustration of problems exhibiting increasing neutrality (from left to right).

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 48. 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 [114]), 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 [15,183,198]. A common idea on how neutrality could be beneficial was the that *neutral networks* would form connections in the search space [15,183].

Definition 49. *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 [183].

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. It seems that the performance of our $(1+1)$ EA in Section 3.4.6.2 was much better than that of our hill climber in Section 3.3.2.2 because our search space $\mathbb{X}$ is much larger than the solution space $\mathbb{Y}$, i.e., $|\mathbb{X}| \gg |\mathbb{Y}$ and such networks seem to "naturally" exist in this representation.

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 [128]. 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 3.3.4.2), even if we could create them [128]. Random, uniform, or non-uniform redundancy in the representation are not helpful for optimization [128,171] and should be avoided.

Another idea [198] 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 50. A function of n variables is *separable* if it can be rewritten as a sum of n functions of just one variable [96,100].

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 operation 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*.

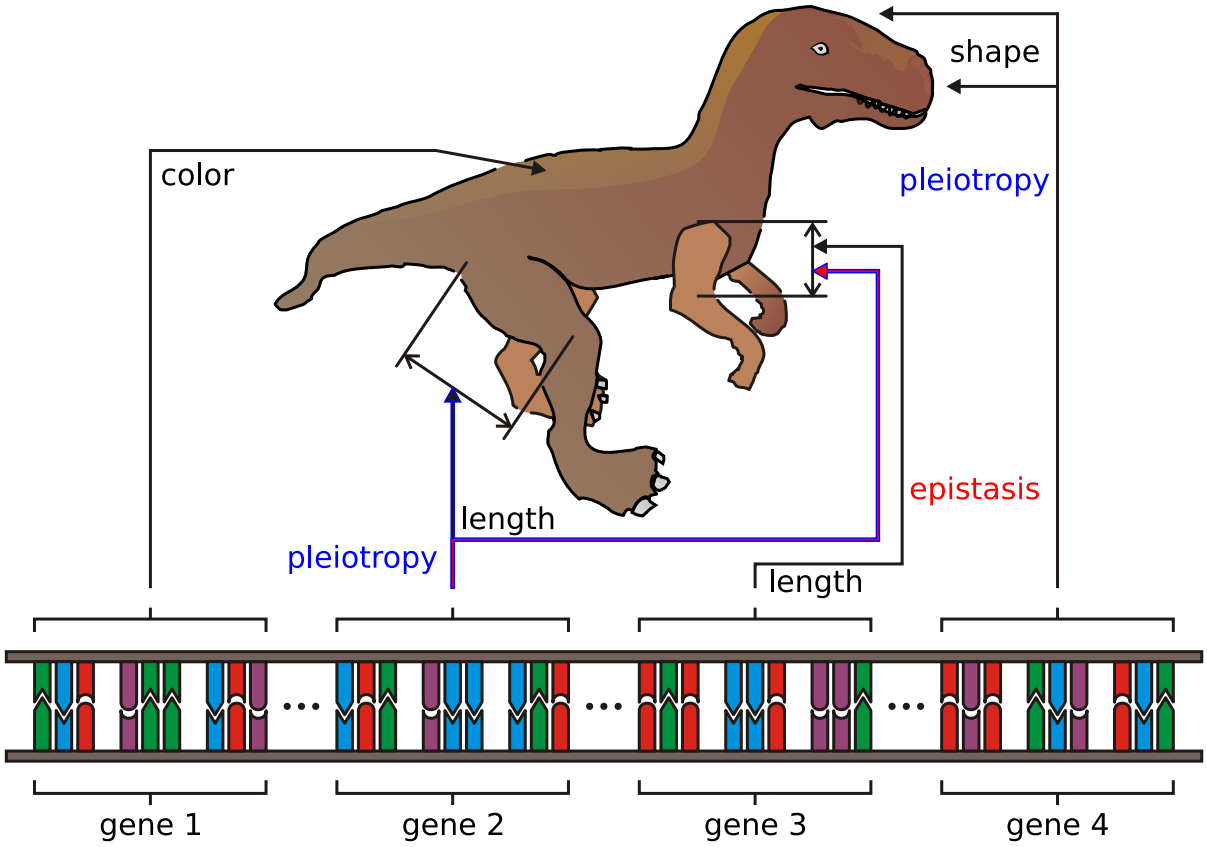

Figure 5.6: An illustration of how genes in biology could exhibit epistatic and pleiotropic interactions in an (entirely fictional) dinosaur.

In biology, epistasis is defined as a form of interaction between different genes [163]. 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 [138].

Definition 51. 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 [6,51,154,208,216].

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 [100]) when no proper subset of decision variables is independent of any other decision variable [154].

Another related biological phenomenon is *pleiotropy*, which means that a single gene is responsible for multiple phenotypical traits [114]. Like epistasis, pleiotropy can sometimes lead to unexpected improvements but often is harmful. Both effects are sketched in Figure 5.6.

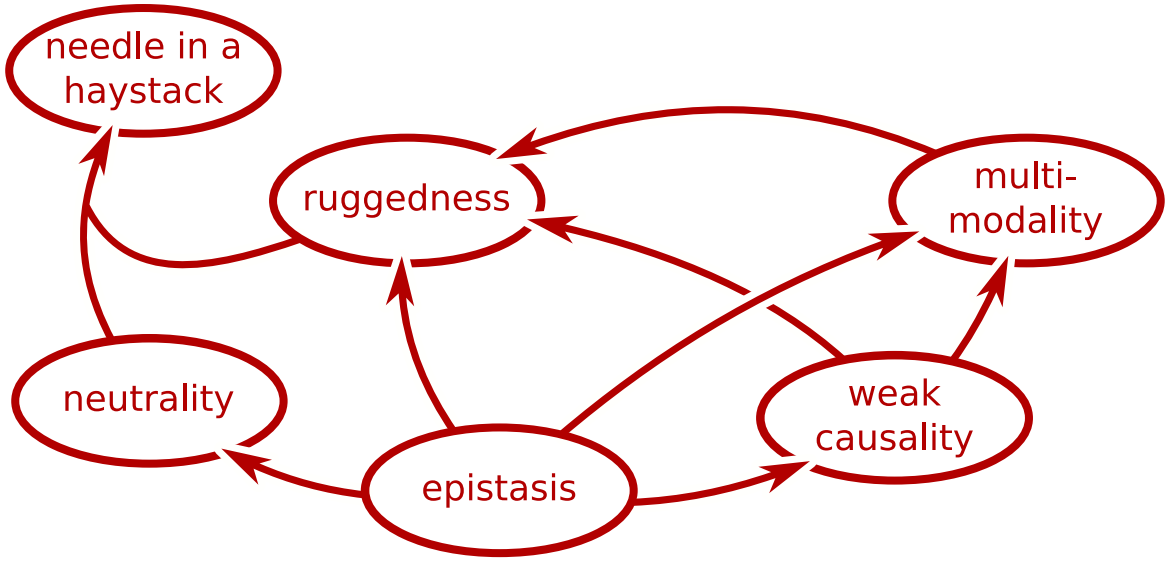

Figure 5.7: How epistasis creates and influences the problematic problem features discussed in the previous sections.

As Figure 5.7 illustrates, epistasis causes or contributes to the problematic traits we have discussed before [208,216]. 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 operations 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 [39,86,101,149]
- modeling of variable dependency via statistical models [34,160]
- variable interaction learning [38]

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"* [21,22] will also strike metaheuristics.

5.6.1 The Problem: Lack of Scalability

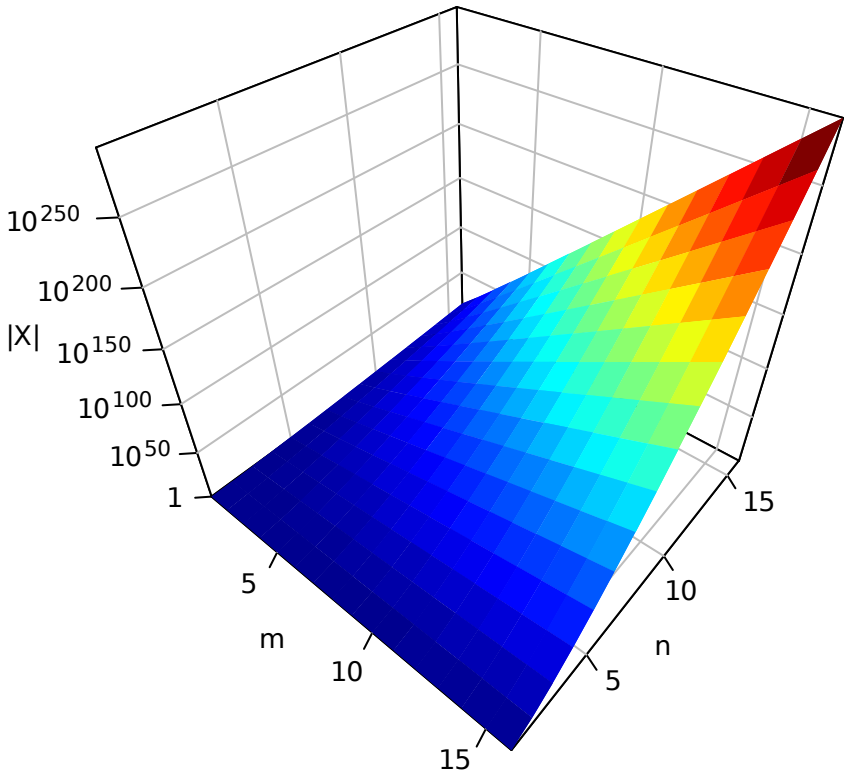

Figure 5.8: The growth of the size of the search space for our representation for the Job Shop Scheduling Problem; compare with Table 2.3.

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 3.5.5) 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 [90,200,202]. 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 [123] 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., $|\mathbb{X}| \ll |\mathbb{Y}|$ [23,58].

The first group of indirect representations uses so-called *generative mappings* assume some underlying structure, usually forms of symmetry, in $\mathbb{Y}$ [49,174]. 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* [58,59,71]. 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 [66]. 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 operations 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 operations j' cannot start right away on the machine, namely if they are not the first operation of their job. The minimum idle time of such a sub job is then the sum of the runtimes of the operations 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 operations.

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

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 [9,66,143,199,203,204].

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] 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.

[9] 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

[10] 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

[11] Anne Auger and Nikolaus Hansen. 2005. A restart CMA evolution strategy with increasing population size. In *Proceedings of the IEEE congress on evolutionary computation (CEC'05), September 2-4, 2005, Edinburgh, UK*, 1769–1776. DOI:https://doi.org/10.1109/CEC.2005.1554902

[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] Shumeet Baluja. 1994. *Population-based incremental learning – a method for integrating genetic search based function optimization and competitive learning*. Carnegy Mellon University (CMU), School of Computer Science, Computer Science Department, Pittsburgh, PA, USA. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.8554&rep=rep1&type=pdf

[14] Shumeet Baluja and Rich Caruana. 1995. Removing the genetics from the standard genetic algorithm. In *Proceedings of the twelfth international conference on machine learning, July 9–12, 1995, Tahoe City, CA, USA*, Morgan Kaufmann, 38–46. DOI:https://doi.org/10.1016/b978-1-55860-377-6.50014-1

[15] 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

[16] 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

[17] Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz (Eds.). 1997. *Handbook of evolutionary computation*. Oxford University Press, Inc., New York, NY, USA.

[18] 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

[19] 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.

[20] Martin Beckmann and Tjalling Charles Koopmans. 1957. Assignment problems and the location of economic activities. *Econometrica* 25, 1 (1957), 53–76.

[21] Richard Ernest Bellman. 1957. *Dynamic programming*. Princeton University Press, Princeton, NJ, USA.

[22] Richard Ernest Bellman. 1961. *Adaptive control processes: A guided tour*. Princeton University Press, Princeton, NJ, USA.

[23] 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.

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] Anne F. Brindle. 1980. Genetic algorithms for function optimization. PhD thesis. University of Alberta, Edmonton, Alberta, Canada.

[30] 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

[31] Rainer E. Burkard, Eranda Çela, Panos M. Pardalos, and Leonidas S. Pitsoulis. 1998. The quadratic assignment problem. In *Handbook of combinatorial optimization* (1st ed.), Panos M. Pardalos, Ding-Zhu Du and Ronald Lewis Graham (eds.). Springer, Boston, MA, USA, 1713–1809. DOI:https://doi.org/10.1007/978-1-4613-0303-9_27

[32] Shaun Burke. 2001. Missing values, outliers, robust statistics & non-parametric methods. *LC.GC Europe Online Supplement* 1, 2 (2001), 19–24.

[33] 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

[34] 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

[35] Josu Ceberio Uribe, Alexander Mendiburu, and José Antonio Lozano. 2015. Kernels of mallows models for solving permutation-based problems. In *Proceedings of the genetic and evolutionary computation conference (GECCO'15), July 11-15, 2015, Madrid, Spain*, ACM, 505–512. DOI:https://doi.org/10.1145/2739480.2754741

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

[43] Philippe Chrétienne, Edward G. Coffman, Jan Karel Lenstra, and Zhen Liu (Eds.). 1995. *Scheduling theory and its applications*. John Wiley; Sons Ltd., Chichester, NY, USA.

[44] 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

[45] William John Cook. 2003. Results of concorde for tsplib benchmark. Retrieved from http://www.tsp.gatech.edu/concorde/benchmarks/bench99.html

[46] 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

[47] 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

[48] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. 2013. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys* 45, 3 (2013), 35:1–35:33. DOI:https://doi.org/10.1145/2480741.2480752

[49] 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

[50] 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

[51] 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.

[52] 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

[53] 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

[54] Kenneth Alan De Jong. 2006. *Evolutionary computation: A unified approach*. MIT Press, Cambridge, MA, USA.

[55] 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.

[56] 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

[57] 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

[58] 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

[59] 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

[60] Edsger Wybe Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1, (1959), 269–271. DOI:https://doi.org/10.1007/BF01386390

[61] Benjamin Doerr, Edda Happ, and Christian Klein. 2008. Crossover can provably be useful in evolutionary computation. In *Proceedings of the genetic and evolutionary computation conference (GECCO'08), July 12-16, 2008, Atlanta, GA, USA*, ACM, 539–546. DOI:https://doi.org/10.1145/1389095.1389202

[62] Marco Dorigo. 1992. Optimization, learning and natural algorithms. PhD thesis. Electronico, Politecnico di Milano, Milano, Italy.

[63] Marco Dorigo, Mauro Birattari, and Thomas Stützle. 2006. Ant colony optimization. *IEEE Computational Intelligence Magazine (CIM)* 1, 4 (2006), 28–39. DOI:https://doi.org/10.1109/MCI.2006.329691

[64] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. 1996. Ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics* 26, 1 (1996), 29–41. DOI:https://doi.org/10.1109/3477.484436

[65] Marco Dorigo and Thomas Stützle. 2004. *Ant colony optimization*. MIT Press, Cambridge, MA, USA.

[66] 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

[67] 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

[68] 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

[69] Eugene S. Edgington. 1995. *Randomization tests* (3rd ed.). CRC Press, Inc., Boca Raton, FL, USA.

[70] Ágoston Endre 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

[71] 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

[72] 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

[73] Steffen Finck, Nikolaus Hansen, Raymond Ros, and Anne Auger. 2015. COCO documentation, release 15.03. Retrieved from http://coco.lri.fr/COCOdoc/COCO.pdf

[74] 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.

[75] 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

[76] Sir Ronald Aylmer Fisher and Frank Yates. 1948. *Statistical tables for biological, agricultural and medical research* (3rd ed.). Oliver & Boyd, London, UK.

[77] Edward W. Forgy. 1965. Cluster analysis of multivariate data: Efficiency vs interpretability of classifications. *Biometrics* 21, 3 (1965), 768–780.

[78] Tobias Friedrich, Nils Hebbinghaus, and Frank Neumann. 2007. Rigorous analyses of simple diversity mechanisms. In *Proceedings of the genetic and evolutionary computation conference (GECCO'07), July 7-11, 2007, London, England*, ACM, 1219–1225. DOI:https://doi.org/10.1145/1276958.1277194

[79] Michael R. Garey and David Stifler Johnson. 1979. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman; Company, New York, NY, USA.

[80] Michael R. Garey, David Stifler 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

[81] 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

[82] 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

[83] Fred Glover and Gary A. Kochenberger (Eds.). 2003. *Handbook of metaheuristics*. Springer Netherlands, Dordrecht, Netherlands. DOI:https://doi.org/10.1007/b101874

[84] David Edward Goldberg. 1989. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[85] 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

[86] 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

[87] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT Press, Cambridge, MA, USA. Retrieved from http://www.deeplearningbook.org

[88] 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

[89] 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

[90] 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

[91] 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

[92] 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

[93] 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

[94] 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

[95] 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

[96] George Hadley. 1964. *Nonlinear and dynamics programming*. Addison-Wesley Professional, Reading, MA, USA.

[97] Doug Hains, L. Darrell Whitley, Adele E. Howe, and Wenxiang Chen. 2013. Hyperplane initialized local search for MAXSAT. In *Proceedings of the genetic and evolutionary computation conference (GECCO'13) July 6–10, 2013, Amsterdam, The Netherlands*, ACM, 805–812. DOI:https://doi.org/10.1145/2463372.2463468

[98] 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 http://hal.inria.fr/inria-00462481

[99] Nikolaus Hansen, Sibylle D. Müller, and Petros Koumoutsakos. 2003. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). *Evolutionary Computation* 11, 1 (2003), 1–18. DOI:https://doi.org/10.1162/106365603321828970

[100] 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

[101] 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

[102] Georges Raif Harik, Fernando G. Lobo, and David Edward Goldberg. 1999. The compact genetic algorithm. *IEEE Transactions on Evolutionary Computation (TEVC)* 3, 4 (1999), 287–297. DOI:https://doi.org/10.1109/4235.797971

[103] 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

[104] John A. Hartigan and Manchek A. Wong. 1979. Algorithm AS 136: A K-Means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979), 100–108. DOI:https://doi.org/10.2307/2346830

[105] 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

[106] 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

[107] 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

[108] Geoffrey Everest Hinton and Steven J. Nowlan. 1987. How learning can guide evolution. *Complex Systems* 1, 3 (1987). Retrieved from http://www.complex-systems.com/abstracts/v01_i03_a06/

[109] 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.

[110] Myles Hollander and Douglas Alan Wolfe. 1973. *Nonparametric statistical methods*. John Wiley; Sons Ltd., New York, USA.

[111] Holger H. Hoos and Thomas Stützle. 2000. Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning* 24, 4 (2000), 421–481. DOI:https://doi.org/10.1023/A:1006350622830

[112] Holger H. Hoos and Thomas Stützle. 2005. *Stochastic local search: Foundations and applications*. Elsevier.

[113] 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.

[114] 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

[115] 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

[116] Dean Jacobs, Jan Prins, Peter Siegel, and Kenneth Wilson. 1982. Monte carlo techniques in code optimization. *ACM SIGMICRO Newsletter* 13, 4 (1982), 143–148.

[117] 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

[118] 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

[119] David Stifler Johnson. 2002. A theoretician's guide to the experimental analysis of algorithms. In *Data structures, near neighbor searches, and methodology: Fifth and sixth dimacs implementation challenges, proceedings of a dimacs workshop* (DIMACS – series in discrete mathematics and theoretical computer science), 215–250. Retrieved from https://web.cs.dal.ca/~eem/gradResources/A-theoreticians-guide-to-experimental-analysis-of-algorithms-2001.pdf

[120] David Stifler Johnson and Lyle A. McGeoch. 2002. Experimental analysis of heuristics for the stsp. In *The traveling salesman problem and its variations*, Gregory Z. Gutin and Abraham P. Punnen (eds.). Kluwer Academic Publishers, 369–443. DOI:https://doi.org/10.1007/0-306-48213-4_9

[121] 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

[122] 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

[123] Winfried Kalfa. 1988. *Betriebssysteme*. Akademie-Verlag, Berlin, Germany.

[124] 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

[125] 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

[126] 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

[127] 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

[128] 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

[129] Donald Ervin Knuth. 1969. *Seminumerical algorithms*. Addison–Wesley, Reading, MA, USA.

[130] Mohamed Kurdi. 2015. A new hybrid island model genetic algorithm for job shop scheduling problem. *Computers & Industrial Engineering* 88, (2015), 273–283. DOI:https://doi.org/10.1016/j.cie.2015.07.015

[131] Pedro Larrañaga and José A. Lozano (Eds.). 2002. *Estimation of distribution algorithms: A new tool for evolutionary computation*. Springer US. DOI:https://doi.org/10.1007/978-1-4615-1539-5

[132] 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

[133] 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.

[134] 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

[135] 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.

[136] 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

[137] José A. Lozano, Pedro Larrañaga, Iñaki Inza, and Endika Bengoetxea (Eds.). 2006. *Towards a new evolutionary computation: Advances on estimation of distribution algorithms*. Springer-Verlag, Berlin/Heidelberg, Germany. DOI:https://doi.org/10.1007/3-540-32494-1

[138] 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

[139] Gangadharrao Soundalyarao Maddala. 1992. *Introduction to econometrics* (Second ed.). MacMillan, New York, NY, USA.

[140] 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

[141] Silvano Martello and Paolo Toth. 1990. *Knapsack problems: Algorithms and computer implementations*. John Wiley; Sons Ltd., Chichester, NY, USA. Retrieved from http://www.or.deis.unibo.it/knapsack.html

[142] 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

[143] 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

[144] 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

[145] Zbigniew Michalewicz. 1996. *Genetic algorithms + data structures = evolution programs*. Springer-Verlag GmbH, Berlin, Germany.

[146] Melanie Mitchell. 1998. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, USA.

[147] 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

[148] 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

[149] 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

[150] Heinz Mühlenbein. 1997. The equation for response to selection and its use for prediction. *Evolutionary Computation* 5, 3 (1997), 303–346. DOI:https://doi.org/10.1162/evco.1997.5.3.303

[151] Heinz Mühlenbein and Thilo Mahnig. 2002. Mathematical analysis of evolutionary algorithms for optimization. In *Essays and surveys in metaheuristics*, Celso C. Ribeiro and Pierre Hansen (eds.). Kluwer Academic Publisher, Norwell, MA, USA, 525–556. DOI:https://doi.org/10.1007/978-1-4615-1507-4_24

[152] Heinz Mühlenbein and Gerhard Paass. 1996. From recombination of genes to the estimation of distributions i. Binary parameters. In *Proceedings of the 4th international conference on parallel problem solving from nature (PPSN IV), an international conference on evolutionary computation, September 22-26, 1996, Berlin, Germany* (Lecture Notes in Computer Science (LNCS)), Springer, 178–187. DOI:https://doi.org/10.1007/3-540-61723-X/_982

[153] 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

[154] 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

[155] 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

[156] 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

[157] José António Oliveira, Luís Dias, and Guilherme Pereira. 2010. Solving the job shop problem with a random keys genetic algorithm with instance parameters. In *Proceedings of the 2nd international conference on engineering optimization (EngOpt2010), September 6–9, 2010, Lisbon, Portugal*, Associação

Portuguesa de Mecânica Teórica, Aplicada e Computacional (APMTAC), Lisbon, Portugal. Retrieved from http://www1.dem.ist.utl.pt/engopt2010/Book_and_CD/Papers_CD_Final_Version/pdf/08/01512-01.pdf

[158] Beatrice M. Ombuki and Mario Ventresca. 2004. Local search genetic algorithms for the job shop scheduling problem. *Applied Intelligence – The International Journal of Research on Intelligent Systems for Real Life Complex Problems* 21, 1 (2004), 99–109. DOI:https://doi.org/10.1023/B:APIN.0000027769.48098.91

[159] Martin Pelikan, David E. Goldberg, and Erick Cantú-Paz. 2000. Linkage problem, distribution estimation, and bayesian networks. *Evolutionary Computation* 8, 3 (2000), 311–340. DOI:https://doi.org/10.1162/106365600750078808

[160] 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. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.8131&rep=rep1&type=pdf

[161] 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

[162] Alan Pétrowski. 1997. *An efficient hierarchical clustering technique for speciation*. Institut National des Télécommunications, Evry Cedex, France.

[163] 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

[164] 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

[165] 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

[166] Chris N. Potts and Vitaly A. Strusevich. 2009. Fifty years of scheduling: A survey of milestones. *The Journal of the Operational Research Society (JORS)* 60, sup1: Special Issue: Milestones in OR (2009), S41–S68. DOI:https://doi.org/10.1057/jors.2009.2

[167] 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.

[168] Ingo Rechenberg. 1994. *Evolutionsstrategie '94*. Frommann-Holzboog Verlag, Bad Cannstadt, Stuttgart, Baden-Württemberg, Germany.

[169] Ingo Rechenberg. Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution. PhD thesis. Technische Universität Berlin; Friedrick Frommann Verlag, Stuttgart, Germany, Berlin, Germany.

[170] Uwe E. Reinhardt. 2011. What does 'economic growth' mean for americans? *The New York Times, Economix, Today's Economist* (2011). Retrieved from http://economix.blogs.nytimes.com/2011/09/02/what-does-economic-growth-mean-for-americans

[171] 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

[172] Y. A. Rozanov. 1977. *Probability theory: A concise course* (new ed.). Dover Publications, Mineola, NY, USA.

[173] Stuart Jonathan Russell and Peter Norvig. 2002. *Artificial intelligence: A modern approach (AIMA)* (2nd ed.). Prentice Hall International Inc., Upper Saddle River, NJ, USA.

[174] 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

[175] Sudip Kumar Sahana, Indrajit Mukherjee, and Prabhat Kumar Mahanti. 2018. Parallel artificial bee colony (PABC) for job shop scheduling problems. *Advances in Information Sciences and Service Sciences (AISS)* 10, 3 (2018), 1–11. Retrieved from http://www.globalcis.org/aiss/ppl/AISS3877PPL.pdf

[176] Sartaj Sahni and Teofilo Francisco Gonzalez Arce. 1976. P-complete approximation problems. *Journal of the ACM (JACM)* 23, 3 (1976), 555–565. DOI:https://doi.org/10.1145/321958.321975

[177] Paul Anthony Samuelson and William Dawbney Nordhaus. 2001. *Microeconomics* (17th ed.). McGraw-Hill Education (ISE Editions), Boston, MA, USA.

[178] Danilo Sipoli Sanches, L. Darrell Whitley, and Renato Tinós. 2017. Building a better heuristic for the traveling salesman problem: Combining edge assembly crossover and partition crossover. In *Proceedings of the genetic and evolutionary computation conference (GECCO'17), July 15-19, 2017, Berlin, Germany*, ACM, 329–336. DOI:https://doi.org/10.1145/3071178.3071305

[179] 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

[180] 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

[181] 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.

[182] 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

[183] 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

[184] Ofer M. Shir. 2012. Niching in evolutionary algorithms. In *Handbook of natural computing*, Grzegorz Rozenberg, Thomas Bäck and Joost N. Kok (eds.). Springer, Berlin/Heidelberg, Germany, 1035–1069. DOI:https://doi.org/10.1007/978-3-540-92910-9_32

[185] Oleg V. Shylo. 2019. Job shop scheduling (personal homepage). Retrieved from http://optimizizer.com/jobshop.php

[186] Sidney Siegel and N. John Castellan Jr. 1988. *Nonparametric statistics for the behavioral sciences*. McGraw-Hill, New York, NY, USA.

[187] 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

[188] James C. Spall. 2003. *Introduction to stochastic search and optimization*. Wiley Interscience, Chichester, West Sussex, UK. Retrieved from http://www.jhuapl.edu/ISSO/

[189] Giovanni Squillero and Alberto Paolo Tonda. 2016. Divergence of character and premature convergence: A survey of methodologies for promoting diversity in evolutionary optimization. *Information Sciences* 329, (2016), 782–799. DOI:https://doi.org/10.1016/j.ins.2015.09.056

[190] Leon Steinberg. 1961. The backboard wiring problem: A placement algorithm. *SIAM Review* 3, 1 (1961), 37–50. DOI:https://doi.org/10.1137/1003003

[191] 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

[192] Thomas Stützle and Holger H. Hoos. 2000. MAX-MIN ant system. *Future Generation Computer Systems* 16, 8 (2000), 889–914. DOI:https://doi.org/10.1016/S0167-739X(00)00043-1

[193] Dirk Sudholt. 2020. The benefits of population diversity in evolutionary algorithms: A survey of rigorous runtime analyses. In *Theory of evolutionary computation: Recent developments in discrete optimization*, Benjamin Doerr and Frank Neumann (eds.). Springer, Cham, Switzerland, 359–404. DOI:https://doi.org/10.1007/978-3-030-29414-4_8

[194] 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/

[195] 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.

[196] Oliver Theobald. 2018. *Statistics for absolute beginners* (Paperback ed.). Independently published.

[197] Renato Tinós, L. Darrell Whitley, and Gabriela Ochoa. 2014. Generalized asymmetric partition crossover (GAPX) for the asymmetric TSP. In *Proceedings of the genetic and evolutionary computation conference (GECCO'14), July 12-16, 2014, Vancouver, BC, Canada*, ACM, 501–508. DOI:https://doi.org/10.1145/2576768.2598245

[198] 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

[199] Jelke Jeroen van Hoorn. 2015. Job shop instances and solutions. Retrieved from http://jobshop.jjvh.nl

[200] 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

[201] 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

[202] 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

[203] 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

[204] 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

[205] 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

[206] 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

[207] Thomas Weise. 2019. JsspInstancesAndResults: Results, data, and instances of the job shop scheduling problem. Retrieved from http://github.com/thomasWeise/jsspInstancesAndResults

[208] 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

[209] 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

[210] 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

[211] 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

[212] 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

[213] 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

[214] 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

[215] 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

[216] 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

[217] 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

[218] 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

[219] 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

[220] 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

[221] 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

[222] 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

[223] 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

[224] 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.

[225] 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.