NTNU – Trondheim
Norwegian University of
Science and Technology

Using Evolutionary Multiobjective Optimization Algorithms to Evolve Lacing Patterns for Bicycle Wheels

Mats Krüger Svensson

Master of Science in Computer Science
Submission date: June 2015
Supervisor: Keith Downing, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

This thesis investigates the use of evolutionary algorithms (EAs) to evolve and optimize lacing patterns of spokes for a bicycle wheel. There are multiple objectives and tradeoffs to be considered when evaluating a lacing pattern, for instance, strength versus balance. To handle this, an evolutionary multiobjective optimization (EMO) method has been used.

Various EMO algorithms and approaches are tested. Among these, the new NSGA-III algorithm is used. Different representations of the lacing patterns for the wheels are compared, and also two different representations of the problem for the EMO.

A novel wheel simulator has been made to test the lacing patterns. As the number of needed wheel simulations take a considerable amount of time, the work is distributed among multiple computers.

The EMO is successful in the search for good lacing patterns. One of the most common patterns, the 3x, is found, along with other comparable patterns with different tradeoffs. The results show no improvement in using the new NSGA-III algorithm compared to the older NSGA-II. The representation with a bias for uniformly laced spokes let the EA evolve better wheels than other representations, showing that carefully choosing the representation for what is being optimized is important. The results also show how important it is to represent a problem using multiple objectives when possible, as the wheels evolved using a higher number of objectives were found to be best.

Sammendrag

This is a Norwegian translation of the abstract.

Denne avhandlingen undersøker bruken av evolusjonære algoritmer (EAer) for å optimalisere eike-mønstre for sykkelhjul. Det er flere *mål* og kompromiss som må tas i betraktning når et eike-mønster skal evalueres, for eksempel styrke i forhold til balanse. For å takle dette har en evolusjonær multiobjektiv optimeringsmetode (EMO) blitt brukt.

Ulike EMO algoritmer og tilnærminger til problemet har blitt testet. Blant disse har den nye NSGA-III algoritmen blitt brukt. Forskjellige representasjoner av eike-mønstre er sammenlignet samt to forskjellige representasjoner av problemet.

For å teste eike-mønstre har det blitt laget en sykkelhjulsimulator. Ettersom antallet simuleringer som trengs tar lang tid, har arbeidet blitt fordelt mellom flere datamaskiner.

EMOen lykkes i søket etter gode eike-mønstre. Et av de mest brukte mønstrene i dag, et 3x mønster, ble funnet sammen med andre sammenlignbare hjul med andre avveininger. Resultatene viser ingen forbedring ved å bruke den nye NSGA-III algoritmen i forhold til den eldre NSGA-II. En representasjon med en preferanse for eike-mønstre som er jevnt fordelt gir bedre hjul enn andre representasjoner, og viser hvor viktig det er å velge riktig representasjon for et problem. Resultatene viser også hvor viktig det er å bruke flere mål som skal optimaliseres om mulig, da hjulene funnet med et høyere antall mål var best.

Preface

This report is the master's thesis for the author. Written in the spring of 2015, it concludes his study in Computer Science at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The supervisor for the project has been Keith L. Downing.

Personal Motivation

I was first introduced to evolutionary algorithms by taking a class named Subsymbolic AI Methods by Keith Downing, in which we used these algorithms to solve various problems. Some of the problems were hard to solve using a standard EA, as there were multiple tradeoffs to be considered. I figured many real-life problems would be similarly hard, and became interested in knowing how this could be solved better.

As for the application of the EA, it was important to me that what I was optimizing was something *tangible*. Something with a close connection to the real world. I have been interested in bicycles for a long time, I even built one from the ground up not long ago. I am very happy about being able to combine two of my passions, bicycling and programming, for this project.

> *"The correct number of bikes to own is $n + 1$, where n is the number of bikes currently owned."* –The Rules

Acknowledgments

I would like to thank Keith, my supervisor, for his valuable feedback the last couple of months, and for letting me do this project which has been a challenging and fun tour of learning.

I would also like to thank the hardware department at IDI, giving me access to an abundance of computers to run experiments on. A big thank you to both the other students at the office and my girlfriend for listening to me ramble about bicycle wheels for months. Lastly, I would like to thank the contributors to the various open source software and libraries that have been used throughout the project.

Mats Krüger Svensson
Trondheim, June, 2015

Contents

1 Introduction **1**
1.1 Background and Motivation . . . 1
1.2 Research Questions . . . 2
1.3 Research Method . . . 3
1.4 Structure of Report . . . 3

2 Background **5**
2.1 Optimization and Evolutionary Algorithms . . . 5
2.1.1 Optimization Problems and Local Search . . . 5
2.1.2 Introduction to Evolutionary Algorithms . . . 7
2.2 Other Types and Variations . . . 10
2.2.1 Real Values . . . 10
2.2.2 Permutations . . . 12
2.3 Evolutionary Multiobjective Optimization . . . 14
2.3.1 Multiobjective Problems . . . 14
2.3.2 Non-Domination and Pareto-Optimality . . . 15
2.3.3 Commonly Used Algorithms . . . 17
2.3.4 Comparing Evolutionary Multiobjective Optimization Algorithms . . . 20
2.4 Domain: Bicycle Wheel Physics . . . 24
2.4.1 The Bicycle Wheel . . . 24
2.4.2 The Physics . . . 25
2.4.3 Truing the Wheel . . . 27
2.5 Literature Review . . . 28
2.5.1 Related Work . . . 28
2.5.2 Structured Literature Review . . . 33

3 Methodology and Implementation **35**
3.1 The Wheel Simulator . . . 35

3.1.1 Approach . . . 35
3.1.2 Implementation . . . 37
3.1.3 Result . . . 39
3.2 Evolutionary Algorithm . . . 43
3.2.1 Approach . . . 43
3.2.2 Implementation . . . 43
3.2.3 Parallelization . . . 44
3.3 Representations . . . 46
3.3.1 Wheel Tests . . . 46
3.3.2 Problem Representation . . . 48
3.3.3 Wheel Representation . . . 50
3.4 Overview of the system . . . 51
3.5 Experimental Plan . . . 52
3.5.1 Parameters . . . 52
3.5.2 Experiments to Run . . . 54

4 Results and Discussion **57**
4.1 Solving the Problem . . . 58
4.1.1 The Problem Domain . . . 58
4.1.2 The Evolved Wheels . . . 60
4.2 Comparing NSGA-II to NSGA-III . . . 63
4.2.1 Results . . . 63
4.2.2 Discussion . . . 63
4.3 Comparing the Wheel Representations . . . 65
4.3.1 Results . . . 65
4.3.2 Discussion . . . 67
4.4 Comparing the Number of Objectives . . . 68
4.4.1 Results . . . 68
4.4.2 Discussion . . . 68

5 Conclusions **71**
5.1 Evaluation . . . 71
5.1.1 Research Questions . . . 71
5.1.2 Summary . . . 72
5.2 Contributions to the field . . . 73
5.3 Future Work . . . 74
5.3.1 Problem Domain . . . 74
5.3.2 EA . . . 74

Bibliography **77**

A Source Code **81**

A.1 Getting It 81
A.2 Running It 81
A.3 Overview 82

B Results **83**
B.1 Viewing Results Yourself 83
B.2 Results From Runs 83

C Additional Wheels **89**
C.1 Testing a wheel 89

List of Figures

2.1 Solution landscapes 6
2.2 The basic evolutionary loop 8
2.3 Crossover and mutation 9
2.4 Basic mutation distribution 11
2.5 Polynomial mutation distribution 12
2.6 Mutation on a permutation 13
2.7 Partially mapped crossover 13
2.8 Example of a multiobjective problem 15
2.9 Non-domination 16
2.10 Diversity of multiobjective algorithm 17
2.11 Workings of NSGA-II 18
2.12 Reference points for NSGA-III 19
2.13 Hypervolume and spacing indicators 21
2.14 Spacing indicator 22
2.15 A bicycle wheel and close up of a spoke 24
2.16 Common lacing patterns 25
2.17 Physics behind the spokes 26
2.18 Wheel truing 27
2.19 EA with island topology 29

3.1 Inner workings of physics engine 38
3.2 Forces from below in the simulator 40
3.3 Torque in the simulator 40
3.4 Simulated wheel from the side 41
3.5 Rim not bendable 42
3.6 Axes of the wheel 46
3.7 Wheel representations 50
3.8 Overview of the system 53

4.1 Solutions in objective space 59
4.2 Some evolved wheels 61
4.3 NSGA-II vs. NSGA-III plot 64
4.4 Permutation vs. Free representation plot 66
4.5 OBJ4 vs. OBJ2 plot 69

C.1 Additional wheels 93

Chapter 1

Introduction

This chapter gives an introduction to the project by discussing its motivation and goals.

1.1 Background and Motivation

In this project, *evolutionary algorithms* will be used to evolve lacing patterns for a bicycle wheel. The wheels should be optimized in several criteria: strength, maximal smoothness of the ride, minimum stress on the spokes and other relevant goals. The wheels will be simulated to measure how good the various lacing patterns are.

Evolutionary algorithms are used for optimization problems. Inspired from biology, they evolve a set of individuals (solutions) in parallel as a population. The individuals are mutated and combined in order to create a new generation of, hopefully, better individuals.

Since the very beginning of EAs, they have been used to optimize real-life mechanical problems. The first decades, the focus was mainly on maximizing or minimizing *one* problem objective. Most real-life problems do, however, consist of several objectives with tradeoffs. For instance, there is often a relation between the weight and strength of an object. Only optimizing for strength could give objects that are too heavy, the optimal solution would probably be something in-between.

For this, Goldberg did in 1989 propose how to extend evolutionary algorithms to take into account multiple objectives [1]. The first successful evolutionary multiobjective algorithms came a few years later, one of them being NSGA [2]. The second generation, consisting of algorithms like NSGA-II [3] and SPEA2 [4], have had great success for many years. These and other algorithms have been used to optimize flywheels [5], bicycle frames [6], several problems relating to the aircraft industry [7, 8, 9] and many other problems from the real world. These and other applications will be discussed more in section 2.5.

A common factor among many of these applications is that they all mostly try to optimize two or three objectives. Handling problems with four or more objectives, called many-objective problems, is difficult. Recently there have been a lot of research on extending the algorithms to handle more objectives, and one of the newest algorithms is NSGA-III [10].

NSGA-III had its first real-life application presented at GECCO14 [11], so it has not been tested much yet. Therefore, testing this new algorithm and others on a problem with multiple objectives is interesting for the field of evolutionary multiobjective optimization.

1.2 Research Questions

The initial problem description from my advisor was:

> *Evolutionary algorithms (EAs), such as genetic algorithms and genetic programs can be applied to a wide array of search problems. Find a challenging problem in any domain in which you have some experience (or a deep interest) and use an EA to solve it.*

The selected problem is *lacing patterns for a bicycle wheel*, in which the goal is to find good lacing patterns. This is a complex problem and needs a computationally heavy simulation to evaluate each pattern. There are also tradeoffs concerning strength, balance, etc., so it will need to be solved using a multiobjective optimization algorithm.

The research questions will thus be:

- Can an evolutionary multiobjective optimization algorithm be used to optimize lacing patterns of a bicycle wheel?
- How does the performance of the new NSGA-III EMO algorithm compare to the older NSGA-II algorithm?
- How do the representations of the problem affect the outcome?

1.3 Research Method

The main research method to answer these questions has been to perform experiments by implementing a system. However, before conducting these experiments, thorough background reading has been done. Chapter 2 contains much of what was learned in this phase.

To run the experiments, a novel bicycle wheel simulator has been implemented. Together with an extension of the MOEA Framework [12] this system achieves the goals outlined in the previous section. Details of the implementation and experimental setup are found in chapter 3.

1.4 Structure of Report

The report is structured in this manner:

- **Chapter 1** introduces the problem, research questions and motivation for this work.
- **Chapter 2** discusses the needed background theory in the involved fields. Evolutionary algorithms for multiobjective problems are described, and we take a look at how they are being used in related work.
- **Chapter 3** goes into detail of how the problem is formulated and solved, and the test setup.
- **Chapter 4** shows the results and discusses them.
- **Chapter 5** sums up what have been achieved.

Chapter 2

Background

In this chapter, the needed theoretical background is provided. Optimization problems and standard evolutionary algorithms are discussed before a deeper look is taken on solving multiobjective optimization problems with evolutionary algorithms. An introduction to the problem domain, bicycle wheels, will be given. At the end of this chapter, related applications of evolutionary algorithms are discussed.

2.1 Optimization and Evolutionary Algorithms

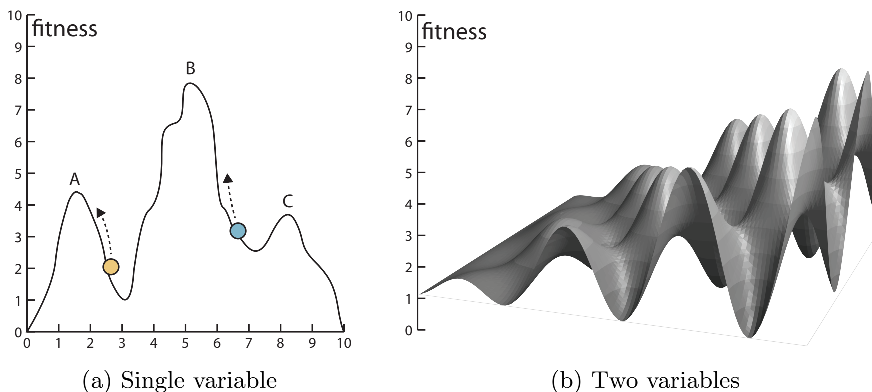
This section gives an introduction to optimization problems in general and how they are solved. The basic workings of an evolutionary algorithm are explained in detail so that a later section can focus only on the multiobjective part.

2.1.1 Optimization Problems and Local Search

Given a set of variables, the goal of an optimization problem is to assign some values to these variables in order to minimize or maximize some *objective function*. A set of values for the variables is a complete solution, so optimizing such a problem can be done with a *local search* in the solution space. A local search works by modifying a solution in several ways to create *neighboring solutions*, rather than building a solution by exploring paths from some initial state [13].

An example of such a mentioned neighbor generation is a problem with two variables, (x, y), and an example solution (0.3, 0.7). The local search procedure can create new solutions by copying the solution and adjusting one of the values a bit. In this example, it ends up with four new solutions: (0.2, 0.7), (0.4, 0.7), (0.3, 0.6) and (0.3, 0.8). These solutions are then run through the objective function to get a value, and then the local search procedure can choose between them based on how good they are. A problem with two variables is shown in figure 2.1b.

There exist several local search procedures, or algorithms, the most basic one being Hill Climbing. Hill climbing is a greedy algorithm that always chooses the best neighbor solution. Figure 2.1a shows an optimization problem where the goal is to maximize an objective function that only takes one variable. It is easy to see that if a solution has the value 2.5 (the orange circle), a greedy search will only bring it up to point A, a local optimum. The global optimum is however at point B, so the greedy approach fails to find the optimal solution. If it started at the blue circle, it would have made it.



(a) Single variable

(b) Two variables

Figure 2.1: Solution landscapes for optimization problems with one (a) and two (b) variables.

To overcome the shortcomings of a greedy approach, most local search algorithms incorporate some randomness and do not always choose the best solution. One such approach is Simulated Annealing [13]. Here, a temperature variable that steadily decreases decides the chance of not choosing the current best solution. In the beginning, when the temperature is high, it chooses almost randomly from the

neighboring solutions. Later in the search, it becomes greedier. If, in figure 2.1a, the current solution was in point C, selecting a non-optimal neighbor solution a few times could move the current solution to the blue circle. From there, it has the possibility to reach the global optimum. While simulated annealing can overcome local optimums, it is still not guaranteed to find the global optimum.

These two algorithms, and most other local search algorithms, are general and contain no problem specific information in their search procedure. All that information is inside the objective function that evaluates the solutions and in a neighbor generating function. This makes it easy to use local search algorithms for new problems. All that has to be done is to write these two functions for the new problem, the cores of the algorithms stay the same without needing any changes. The drawback with this is that the search cannot really exploit known information in the problem domain.

2.1.2 Introduction to Evolutionary Algorithms

An evolutionary algorithm is a more advanced local search algorithm, inspired from evolution in biology. Compared to the already mentioned local search algorithms, evolutionary algorithms work on a *set of solutions* in parallel. In EA terms, a solution is an individual, and the set of individuals is the population. Using many solutions at the same time has the advantage of exploring more areas of the solution space. This parallelism is different from running multiple instances of simulated annealing, as in this case the solutions can "communicate" during the run.

There are three key elements from evolution that should be present in an evolutionary algorithm: selection, variation and heritability. Selection is that there must be something that makes one solution more favorable than another solution. This selection should create pressure on the individuals so only the fittest survive and mate, and thus the population becomes fitter with time. Variation means that some new traits in the population should randomly appear, in order to evolve the population and make it better. Heritability means that children should inherit many traits from their parents. If a good parent survives and gets to mate, it would all be a waste if the children were nothing like the parent. There should also be a certain degree of diversity among the individuals in the population, in order to not too quickly converge to a local optima.

Figure 2.2 shows the basic evolutionary loop. It all starts with an initial population of solutions (individuals). Often they are randomly created, but known information about the solution space can be used to create better fit individuals from the start.

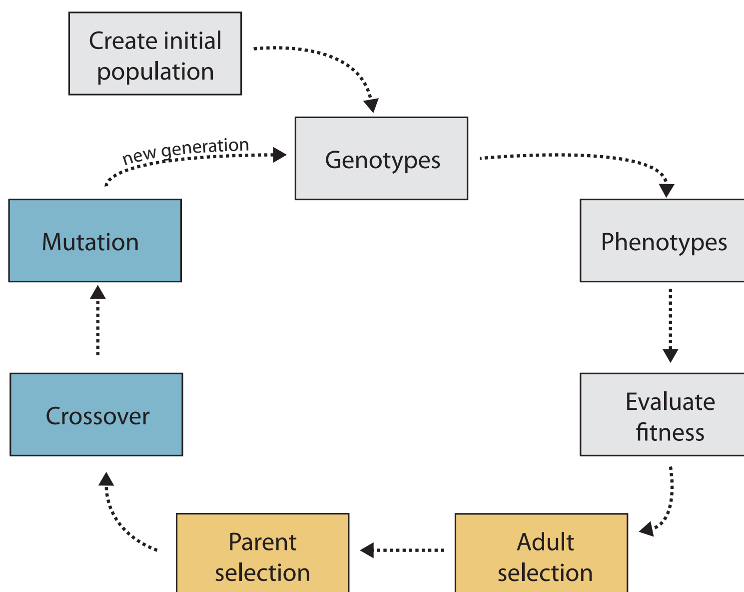


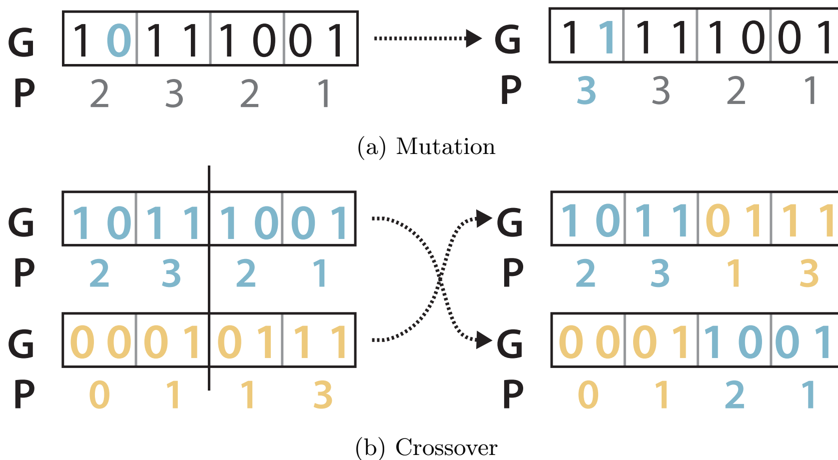
Figure 2.2: The life of an individual in a basic evolutionary algorithm.

Genotype is the *DNA* of an individual. It is often represented as a list of bits, but other genotypes are possible and will be discussed later. **Phenotype** is the *solution* an individual represents. A step is needed to convert the genotype to a phenotype. Figure 2.3 shows various genotypes as bitstrings and their corresponding phenotype. In this case, two and two bits are considered an integer, so the genotype is a list of integers. A phenotype can also be a list of real values or any other thing that is a solution for some problem. It is important that small changes to an individual's DNA do not create vastly different solutions, as this removes the locality of the search. The representations for a genotype and phenotype can often be reused between problems with only small adjustments. For instance, for a problem where the solution is a list of integers, the only change needed is how many bits are used per integer and the length of the bitstring.

Fitness function gives each individual a score based on the phenotype. As with the local search algorithms discussed earlier, an EA is also very general. It is in the fitness function that problem-specific adjustments need to be made. It is important that the fitness function can give partial credit to solutions that are not optimal but still good, so that they have a higher chance of survival compared to not as good solutions.

Adult selection decides which individuals should be allowed to *"grow up"* and later have the chance to become parents. Often, this step is skipped, and a *full generation selection* is used, in which all individuals from the earlier generation are removed and only those from the current generation gets to live. Another possibility is *generational mixing selection*, in which the individuals from the current generation has to compete with individuals from previous generations. A third option is *overproduction selection*, in which each generation produces more offspring than it allows to grow up.

Parent selection decides which individuals get to pass their genes to the next generation. As discussed earlier in this section, a greedy search may quickly converge to a local optima and get stuck, and simulated annealing avoids this by using randomness. An EA does something similar: the chance of an individual to be selected as a parent and go into the mating pool is higher when its fitness is high, but it is still possible to get there without being the best solution. This maintains diversity between the solutions. A common way to do this is *fitness proportionate selection*, where the chance of being selected is proportionate to the individual's fitness in relation to the total fitness of all individuals in the population. Another often-used selection is *Tournament selection*, in which k random individuals are selected, and with probability p the best of those is selected or with chance 1-p one is selected at random.



(a) Mutation

(b) Crossover

Figure 2.3: Crossover and mutation of genotypes and what happens to their phenotypes. (a) shows a mutation where a bit is flipped, (b) shows a crossover where two parents are split and combined to create two offsprings.

Crossover is a way to combine parents' genotype to make a new individual. Heritability is important, so the crossover operator should try to combine the parents in a way that keeps most of their good properties intact. How this is done depends on the representation of the genotype. For bitstrings, the standard way to do a crossover is to randomly select a crossover point, and then swap what is after that point between the two parents. Figure 2.3b shows a crossover where the crossover point is in the middle of the bitstring. This creates two new children genotypes. It is also possible to swap multiple segments.

Mutation is what introduces variance to the population by modifying a solution just a little. When the genotype is a bitstring, this is most often done by flipping some bits. Normally, each bit is flipped with a probability of 1/n, where n is the number of bits. This probability ensures that on average one bit is flipped, but that there are chances for bigger changes. It is also possible to set a probability for *an individual* to be mutated, and when selected for mutation a single, random bit in its genotype is flipped. Figure 2.3a shows a bit being flipped and the resulting phenotype.

In addition to these, a common strategy is to use **eltitism**. Elitism is that some of the best individuals from a generation are copied directly to the next generation. This makes it so that the best fitness of the population is never decreasing.

2.2 Other Types and Variations

Above, a bitstring genotype that represents a list of integers as phenotype was explained. This representation is easy to explain when introducing evolutionary algorithms and works fairly well on some problems. However, for this project, some more advanced representations with different mutation and crossover operators are needed.

2.2.1 Real Values

In many problems, the phenotype should be a list of real values with some upper and lower bound. This can be achieved using the bitstring genotype, by converting the integer value to a new value within this range. If we have 7 bits, where 127 would be the max value, a value can be turned into a real value in the correct range using this formula: $bound_{lower} + \frac{value}{127} \cdot (bound_{upper} - bound_{lower})$

However, mutating the underlying bitstring is problematic. To show the problem, a test has been run where the 7-bit long bitstring [1000000], representing the value

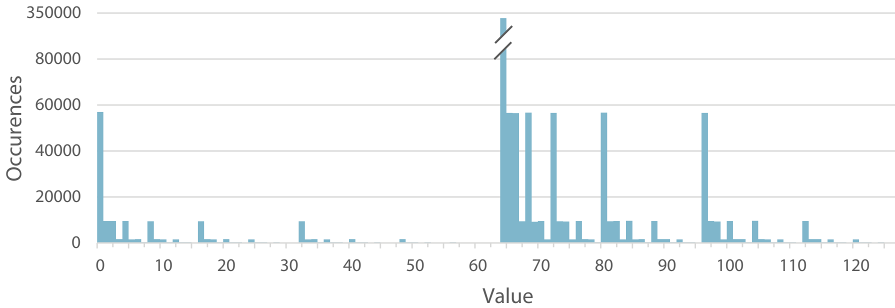


Figure 2.4: How many times each value appeared when mutating the value 64 one million times. Note that the graph is cut off, the value 64 appeared almost 350 000 times.

64, has been mutated a million times. Each bit was flipped with probability 1/7. The distribution of the new values can be seen in figure 2.4. As the figure shows, certain values appear more often than others with a very bad distribution. It is easy to see why, for instance, the value 0 appears many times since [0000000] is only a single flip from the original bitstring. The same goes for 65 with the bitstring [1000001], it appears over 50 000 times. However, the value 63 only appears 1 time, as [0111111] would require *all* bits to be flipped. This is called a Hamming cliff, when there are large distances between adjacent integers. These issues destroy the *locality* of the evolutionary algorithm, as it is hard to adjust the values just a little.

Instead, a good way to work with mutation and real values is to use *polynomial mutation* (PM) [14]. Instead of having the genotype be a bitstring representing real values, the genotype would here be a list of reals (of course, these real values are still bits deeper down). For the polynomial mutation operator, one specifies a distribution index. The operator then works by changing the input value to a new value with a polynomial distribution around the original value, with the set distribution index determining the variance.

The results of doing this one million times on the value 64 can be seen in figure 2.5 for various values of the distribution index. This mutation gives values that are better distributed and also has the added benefit that one can control the spread of the new values. A high distribution index gives a higher probability for the new value being closer to the original value, while a low index gives a more uniform distribution.

There is also a crossover operator exhibiting some of the same qualities as polynomial mutation, named *simulated binary crossover* [15], often shortened to SBX.

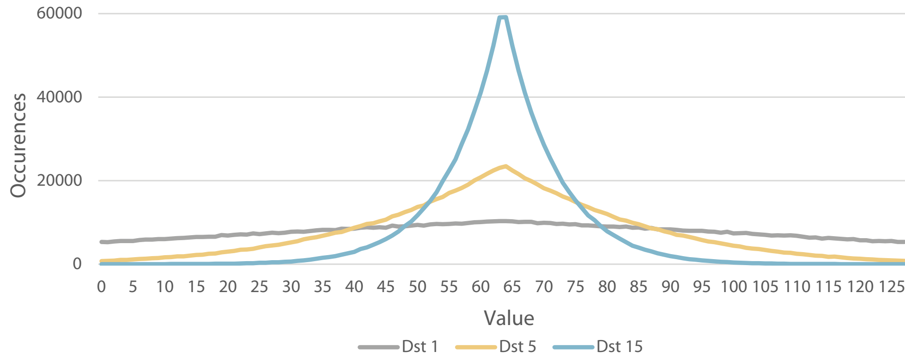
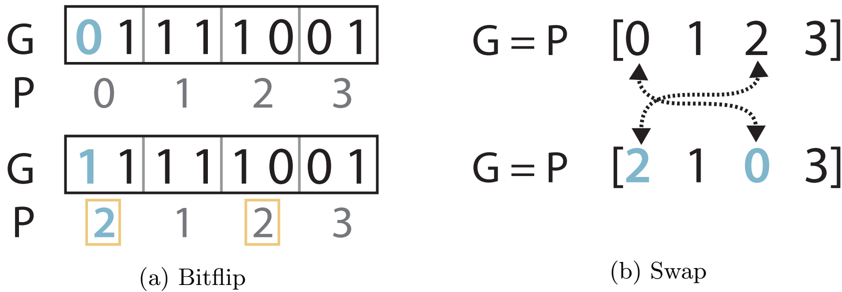


Figure 2.5: Using polynomial mutation on the value 64 for various distribution indexes, recording how many times each value appeared during a million mutations. The distribution is much better than for bit flipping.

2.2.2 Permutations

A way to handle permutations is also needed. Figure 2.6a shows an example with four numbers, 0 to 3, represented by a bitstring genotype. A mutation that randomly flips some bits will give many invalid solutions, phenotypes that do not represent a permutation of the numbers 0 to 3. This can trivially be fixed by modifying the mutation operator to instead swap places of bit segments. However, not using a bitstring at all is even easier, so when working with permutations having the genotype be a list of numbers is the preferred solution. Then all that is needed is to swap the place of two numbers in the list, as shown in figure 2.6b. In addition to swap, *insertion* is often also used. Insertion takes a random number and inserts it somewhere else in the permutation, shifting the other values to the side.

Handling crossover with permutations is not as straightforward. One way to do it is to do a normal crossover, and then later try to fix it by removing duplicate elements and filling in the missing ones. This can, however, destroy the locality of the inheritance, as the changes will be very drastic and the result will not



(a) Bitflip (b) Swap

Figure 2.6: Using mutation on a bitstring that represents an ordering can give invalid results, as shown in (a) that ends up with two 2's and no 0. A better solution is to swap two elements, as in (b).

have the same properties as the parents. A better way, but with the same idea, is called Partially Mapped Crossover (PMX), and was made for arranging the order of cities in a traveling salesman problem[16]. The idea is to pick two segments of equal size from both parents, and create a mapping between them and apply that mapping to the elements. Figure 2.7 shows an advanced example, where there even after the first mapping remain duplicates. So then the mapping has to be run again.

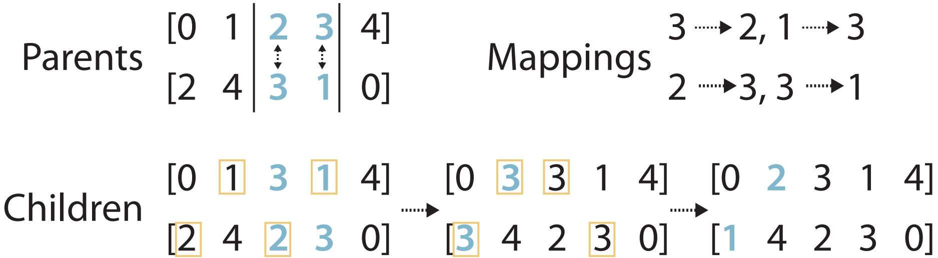


Figure 2.7: Using PMX to crossover a permutation. First swap two substrings, then use the mapping until there are no more duplicates (marked with orange boxes).

2.3 Evolutionary Multiobjective Optimization

While the local search procedures introduced earlier work great for optimizing problems with a single objective, many real-life problems have multiple objectives and tradeoffs that need to be considered. This section gives an introduction on how to tackle such problems, and how evolutionary algorithms can be extended to work with these strategies.

2.3.1 Multiobjective Problems

Multiobjective problems are problems where more than one function should be minimized or maximized. An often-used example [2] for such a problem is this:

$$\begin{aligned} Minimize \quad f_1 &= x^2 \\ Minimize \quad f_2 &= (x-2)^2 \end{aligned}$$

A graph of these two functions is shown in figure 2.8. A solution in this case is a single variable, (x), and the fitness of the solution is a tuple $(f_1(x), f_2(x))$. For the solution $x_1 = 0$, the fitness would be $(0, 4)$, and for the solution $x_2 = 2$ the fitness would be $(4, 0)$. It is impossible to define one of these solutions as better than the other without additional information. How can then an evolutionary algorithm choose between a set of individuals with a multi-dimensional fitness?

One naive way to do this is to convert the multiple fitness values into a single scalar. This conversion is often done by summing the values and weighting them by preference.

$$newfitness = \sum w_i \cdot f_i(x)$$

Where w_i is the importance of objective i. Using the example from before, having equal weights for both objectives, e.g. $w_1 = w_2 = 0.5$, would result in $x = 1$ as the found solution. The solutions $x_1 = 0$ and $x_2 = 2$ from before would be the results of having one of the weights equal to 0 and the other equal to 1.

While this works for simple problems, it will in general not give good results. The only advantages of this technique are that it can be used to make existing local search algorithms handle multiple objectives, and that the user of it can to some extent specify the relative importance of each objective. However, setting the proper weights is a hard problem in itself, and can often lead to bad solutions.

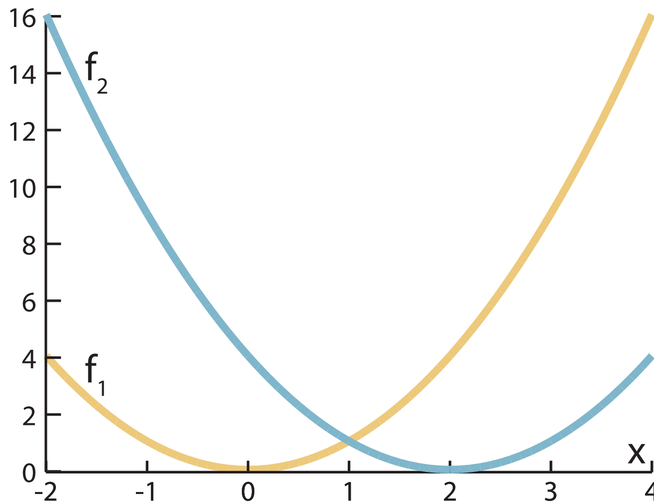


Figure 2.8: The two fitness functions for a fictive optimization problem.

Besides, this method will converge towards a single point in the solution space, while for real-life problems one would often be interested in knowing about multiple solutions and their tradeoffs. This convergence can also make the algorithm quickly become stuck in initially promising solutions.

2.3.2 Non-Domination and Pareto-Optimality

In 1989, Goldberg [1] introduced an idea of how a proper multiobjective algorithm could work for an evolutionary algorithm: It should sort the individuals on non-domination and have a niching operator to maintain diversity.

The two solutions from before, $x_1 = 0$ and $x_2 = 2$ with fitness $(0, 4)$ and $(4, 0)$ are considered equally good and cannot be discerned between. However, if one considers a third solution, $x_3 = -2$ with fitness $(4, 16)$, and compare it to x_2 one can say that one of them is better than the other. They have the same value for f_1, but for f_2 the value for x_2 is much lower. As this is a minimization problem, it is obvious that x_2 is better than x_3 since one gets a lower value for f_2 without getting a worse value for f_1. So we say that x_2 *dominates* x_3.

The formal definition for domination is: For two solutions x_1 and x_2, x_1 dominates x_2 if x_1 is no worse than x_2 in any objective, and x_1 is strictly better than x_2 in at least one objective.

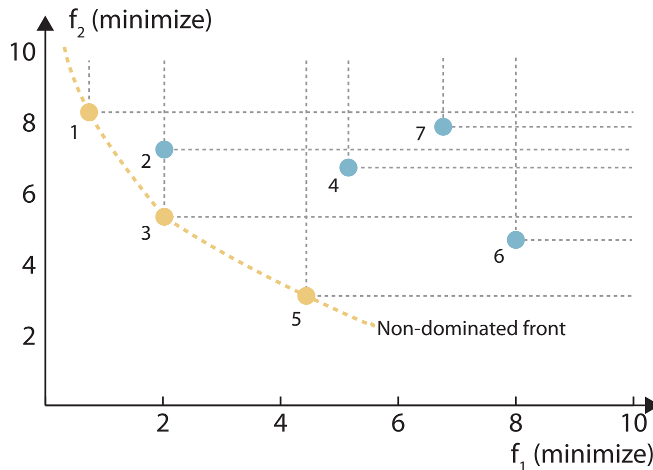


Figure 2.9: Showing a set of solutions in the objective space, where each axis is one of the objectives. Each solution dominates those above and to the right of it (the gray dotted lines).

Solutions that are not dominated by any other solutions are said to be *non-dominated*. Figure 2.9 shows a minimization problem with two objectives and a set of solutions in the objective space. By *objective space*, one means a space where each dimension is one of the objectives in the problem. There are several non-dominated points that are "equally good", and they together make what is called the non-dominated front. If we take the set of *all* possible solutions, those that are then non-dominated are said to be *pareto-optimal*. The goal of the evolutionary search should be to find solutions as close to the pareto-optimal front as possible while being spread out and cover most of it.

To end up with a lot of solutions spread out, diversity must be maintained throughout the search. For this, Goldberg suggested a niching strategy that takes into account a solution's proximity to other solutions in addition to its non-domination when selecting parents. Figure 2.10 shows a search with four solutions that are non-dominated. If there were to be an equal chance to choose between each of them to become parents, by time they would all converge to a single point close to the pareto-optimal front. In this case, the niching operator should increase the chances of selecting the solution far away from the others, as this will maintain the diversity and hopefully give solutions that cover more of the pareto-optimal front.

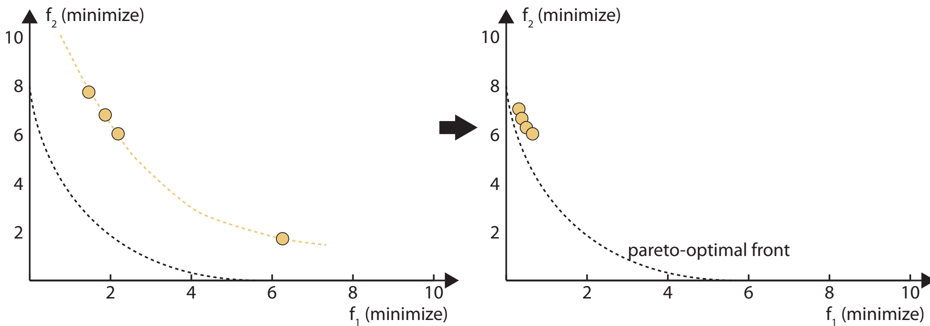


Figure 2.10: The algorithm should ensure high diversity in the population. Otherwise it will converge to a single point and not cover the whole pareto-optimal front.

2.3.3 Commonly Used Algorithms

There exist many algorithms to deal with multiobjective problems, and here some of the most common ones will be discussed.

NSGA

Nondominated Sorting Genetic Algorithm [2] is the first of a line of algorithms by Deb et al. While it today is obsolete, as many newer and better algorithms have been found since 1994, it is interesting because it was the first algorithm to successfully incorporate the ideas put forward by Goldberg.

NSGA starts by identifying the individuals in the non-dominated front and assign a large dummy fitness value to them. Then those individuals that have similar *phenotypes* gets a penalty that decreases this dummy value, in order to promote diversity. After this, the non-dominated individuals are removed from the population, and the fitness procedure is run again on the remaining population. The new non-dominated front will this time get a lower dummy fitness value. This continues until the entire population has gotten a dummy fitness value. The individuals are then selected based on this new fitness value.

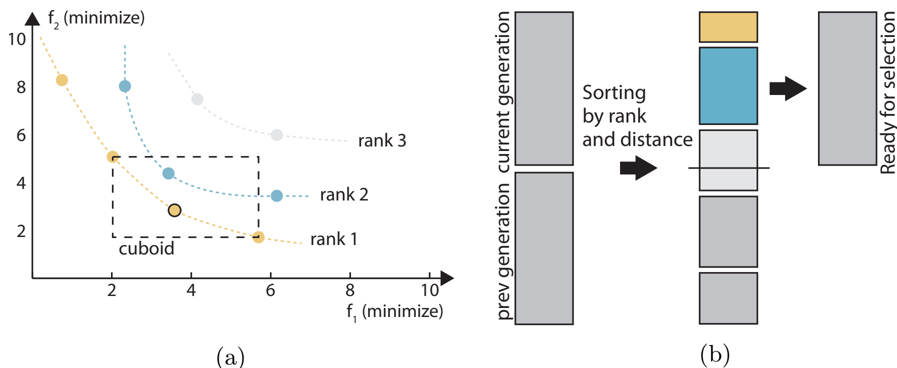
The fact that the individuals end up with a single fitness value means that the NSGA is just a step between the fitness function and the selection operators. No other modifications to the standard evolutionary algorithm are needed. One big drawback is that it is hard to find a good value for how much solutions should

be penalized for being similar. It is also a drawback that it considers similarity on the phenotype, not in how good the solutions are in the various objectives.

NSGA-II

The second version of NSGA, the NSGA-II [3], is a highly successful algorithm that has been used in many applications. NSGA-II starts similarly as its predecessor by finding the non-dominated individuals. The non-dominated individuals are given *rank 1*, and those only dominated by individuals in rank 1 are then assigned rank 2. This goes on until each individual has a rank. Several ranks are displayed in figure 2.11a. To maintain diversity, a cuboid between an individual's two closest neighbors in the same rank is found, also shown in figure 2.11a. This distance is found in the *objective space*, compared to NSGA that used similarity between solutions.

The individuals are then sorted by rank and distance such that lower ranks come first, but within the same rank those with a higher distance from other individuals come first. Then the first N individuals are selected for the normal selection, crossover and mutation steps. To utilize elitism, NSGA-II adds all individuals from the previous generation to the population before the non-dominated sorting and distance calculations are carried out. Figure 2.11b shows the process.



(a) (b)

Figure 2.11: NSGA-II (a) shows individuals being given a rank and a distance value, (b) shows how this is used to select which individuals can later be selected by the evolutionary algorithm.

SPEA

Strength Pareto Evolutionary Algorithm, SPEA2 [4], is another often used way to tackle multiobjective problems. It is similar to NSGA-II in idea but somewhat different in execution. As NSGA-II, SPEA2 also holds an archive of the best solutions from the previous steps, and the non-domination and distances are calculated including that archive in addition to the current population. SPEA2 does not divide the individuals into different ranks, however. Instead, it calculates per individual which individuals that dominate it, and which individuals it dominates. The number of individuals dominated becomes that individual's *strength* number. Then each individual is assigned a *raw fitness*, which is the sum of all strengths of individuals dominating it. Compared to NSGA-II, this means that individuals in the same rank here will get different fitness values, which makes it possible to select better between them.

NSGA-III

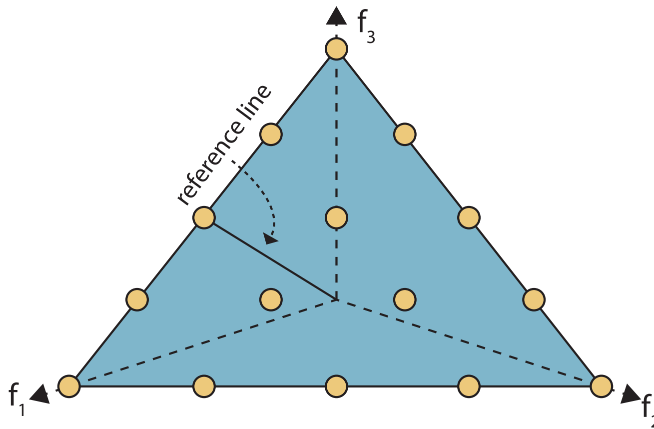


Figure 2.12: Reference points and a reference line for a 3-objective problem with p=4 for use in the NSGA-III algorithm.

While NSGA-II and SPEA have been used with great success for numerous multiobjective problems, they and many other non-domination based searches suffer when the number of objectives becomes too high. Some of the problems [10, 11]

are: as the objective space gets more dimensions, a larger and larger fraction of the population will be non-dominated. That finding the nearest individuals to compute distance becomes more computationally demanding and that this distance become somewhat meaningless. Lastly, recombining parents will often lead to bad individuals, as the parents are far away from each other.

NSGA-III [10, 17] is an upgrade of NSGA-II to alleviate some of these problems. The core of NSGA-III is very similar to NSGA-II. The ranking of individuals and preservation of a group of the best individuals from previous generations work just as before. However, where NSGA-II sorted the ranks internally based on individuals' distance to other individuals, NSGA-III uses reference points.

These reference points can be added manually, to fine-tune which area the search should concentrate on, or be automatically calculated to lay uniformly on a plane in the objective space with p divisions. Figure 2.12 shows such calculated points for a 3-objective minimization problem where the number of divisions is 4. An *ideal point* lies in origo, and for each reference point a line from the ideal point to the reference point is made. Then for each individual, the orthogonal distance to each reference line is calculated, and the individual is associated with the reference point belonging to the closest reference line. The selection operator will then pick individuals such that they cover most of the reference points, and thus are thoroughly spread out in the objective space.

NSGA-III is a state of the art algorithm that has been shown to handle 15-objective test-problems. It has, however, not been tested much in real-life applications yet.

2.3.4 Comparing Evolutionary Multiobjective Optimization Algorithms

As discussed, EMO algorithms should try to find solutions that approach the pareto-optimal front while still having a high diversity. The algorithms should preferably also be consistent and fast. To compare algorithms, various metrics (or *indicators*) have been devised that can quantify how well an algorithm achieves these goals [18]. These metrics often measure the quality of the solutions found by an algorithm. Since algorithms can excel in different metrics, deciding which algorithm is best ironically becomes a multiobjective problem.

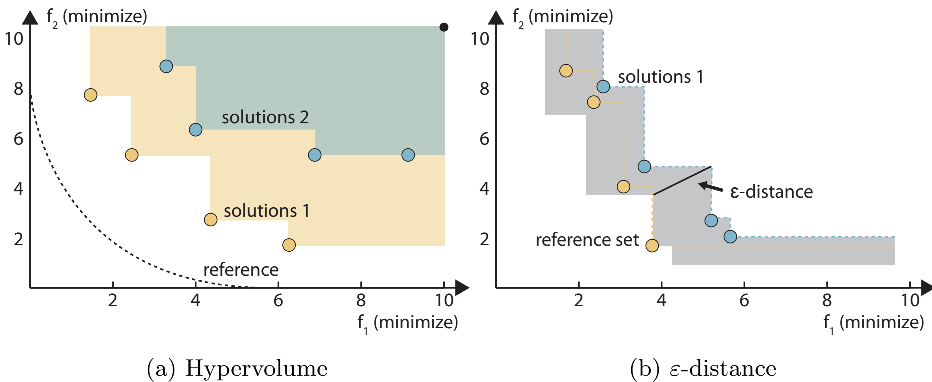
Some of the following indicators need a *reference set* to calculate their values. The reference set is a set of non-dominated solutions, preferably the true pareto-optimal front. For problems where this front is not known, which is true for

most real-life problems, one can instead use the set of best-found solutions from multiple runs.

Hypervolume Indicator

The hypervolume indicator tries to measure the area (or volume, etc. in higher dimensions) covered by the solutions in the objective space. This is done by calculating the union of the area dominated by each solution, from a reference point. The hypervolume of a solution set indicates how close the set is to the reference set, and also somewhat how well the solutions are spread out.

The reference point is set so as to be dominated by all solutions in the reference set. In figure 2.13a it can be seen in the upper, right corner. The hypervolumes of the various solution sets then grow from this point towards the reference set. Before calculating the hypervolume value for a solution set, the objective values should be normalized with respect to the reference set as to not weigh any objective more than the others.



(a) Hypervolume

(b) ε-distance

Figure 2.13: Two indicators for comparing multiobjective algorithms. Hypervolume (a) measures the area covered, ε-distance (b) the distance one set has to be moved to completely dominate some reference set.

As shown in figure 2.13a, this can give a good quantification on how good a solution set is. The orange set, solutions 1, is much closer to the reference solutions, and ends up with a bigger area (volume) than the blue solution set. Some

drawbacks with the hypervolume indicator are that it is computationally heavy to calculate for higher dimensions with big solution sets, and that it can give a false indication of which algorithm is best when the pareto-optimal front has weird shapes.

Additive Epsilon Indicator

The additive epsilon indicator measures the distance one solution set has to be moved in the objective space to completely dominate the reference set. This is shown in figure 2.13b, where the blue solution set is moved to cover the reference set. The sets are with this indicator also normalized before calculation.

The value of the epsilon indicator indicates how close the solutions in the set are to the reference set, so the goal is to get as small a value as possible. The indicator says nothing about the distribution of the solutions, so one should not make any conclusions based on this indicator alone.

Spacing Indicator

The spacing indicator measures how well spread the solutions in a set are. For each solution, the distance to the closest neighbor solution is calculated. The spacing value is then calculated as the standard deviance of the distances. So the goal should be to get the spacing value low, as this indicates that the solutions are uniformly distributed in the objective space. To calculate this, no reference set is needed.

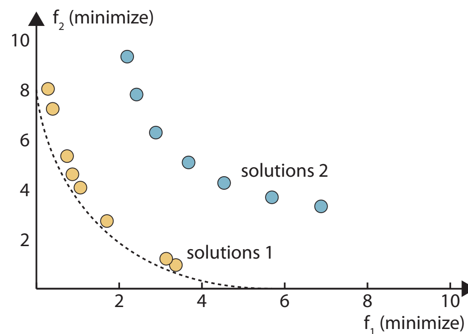


Figure 2.14: Spacing indicator. Blue would receive a better score, as the solutions are more uniformly distributed in the objective space.

In figure 2.14, the blue solution set would get a lower (better) spacing value than the orange solution set because it is more uniform. As the figure also shows, this metric does not say anything about the *quality* of the solutions. Here the orange solutions are much better, but more lumped together.

So, one indicator alone cannot be trusted to show the whole picture. Utilizing these three indicators together, however, one should be able to get solid measurements on the quality of solutions/algorithms.

Statistical Tests

Since the results of evolutionary algorithms are highly stochastic, one cannot run each algorithm once and then draw any conclusions based on the observed values for the different indicators. So what needs to be done is to run each algorithm multiple times and record the values, grouping the values per algorithm.

However, when that is done, one cannot necessarily calculate the average or median value for each group/algorithm and conclude based on that. For instance, if the standard deviation of the recorded values is high and the number of algorithm runs is low, one should be careful when drawing any conclusions.

A recommended approach [19] is to apply the Kruskal-Wallis test on the recorded values. In this case, the H_0 hypothesis would be that the "true" median of the groups of values are the same, and H_1 that they are different. So, if the null hypothesis has to be rejected, this means that there is a statistically significant difference in the groups of values, and it can be concluded that one algorithm was better than the others.

2.4 Domain: Bicycle Wheel Physics

This section provides the needed knowledge in the problem domain, namely how a bicycle wheel works. Domain specific terminology is introduced and explained here.

2.4.1 The Bicycle Wheel

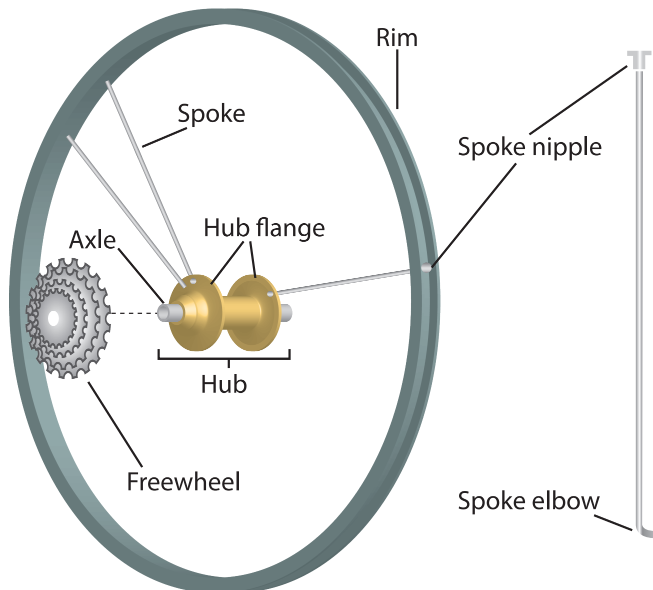
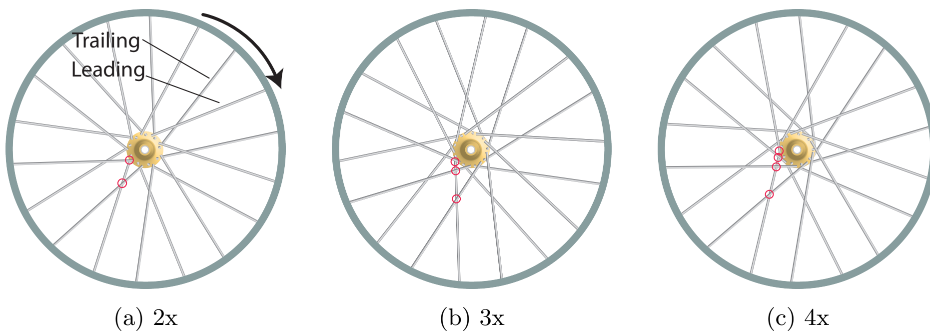


Figure 2.15: A bicycle wheel and close up of a spoke

Figure 2.15 shows the main parts of a bicycle wheel. The hub is the centerpiece and contains an axle that is used when mounting the wheel on a bicycle. The freewheel with the cogs used to move the bicycle forward is mounted on the hub. The hub also has two flanges with holes in it for inserting the spokes. The end of the spoke where the nipple is placed, is inserted first through the flange and then through a hole in the rim where it is secured using the nipple. The elbow on the spoke keeps it from going all the way through the hole in the flange. The spokes can be inserted from both sides of the hub flange, and it is common to alternate

between them. The figure shows two spokes next to each other going both ways through the flange.

There are several standard *lacing patterns* for the spokes. Most commonly, 32 or 36 spokes are used and a "2x" or a "3x" pattern [20]. The number refers to how many times a spoke crosses another spoke from the same hub flange. Figure 2.16 shows three common lacing patterns for a 36-spoked wheel. Every second spoke is angled the same way as the rotation of the wheel ("leading spokes") and the others are angled the other way ("trailing spokes"). Wheels with angled spokes are called tangentially laced wheels. It is also possible to have a wheel where each spoke goes straight from the hub to the rim, these wheels are called radial laced wheels. See figure 2.16 for tangential laced wheels, and figure 2.17b for a radial one.



(a) 2x (b) 3x (c) 4x

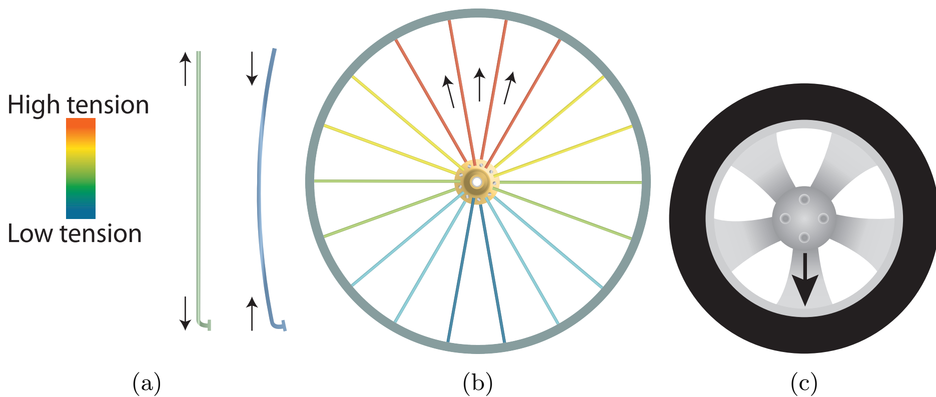
Figure 2.16: Common lacing patterns for a spoked wheel, showing only one side. The patterns are named after how many times a spoke crosses another spoke, marked in red.

When torque from the freewheel is applied to the hub, the trailing spokes are needed. A radially laced wheel would not be able to transfer that force through the spokes out to the rim. So all rear wheels must be tangentially laced, at least on the side where the freewheel is mounted.

2.4.2 The Physics

For being such a simple everyday-object, how the bicycle wheel works can be surprisingly unintuitive. Figure 2.17 shows how a spoke would react to compres-

sion: it would buckle. Looking at a car wheel, figure 2.17c, it is obvious that the force on the axle goes through the lowermost “spoke” as compression. Since a small, thin spoke on a bicycle wheel would be unable to withstand such a force, the workings have to be something completely different. The “secret” is the fact that all spokes are mounted highly tensioned. This is done by turning the spoke nipple until the spoke is stretched. The spoke can handle loads many times its own weight in this direction.



(a) (b) (c)

Figure 2.17: (a) spokes under tension and compression, (b) a wheel with load on the axle "hangs" from the topmost spokes, (c) a car wheel "stands" on the lowermost spoke

Figure 2.17b shows an illustration of how the tension in the spokes of a wheel is after putting weight on the axle. When load is applied, the tension in the lowermost spokes decrease and the tension in the uppermost spokes increase. There is never any compression on the spokes, just various degrees of tension [21][22]. This means that a good way of picturing a bicycle wheel is that the hub “hangs” from the upper spokes.

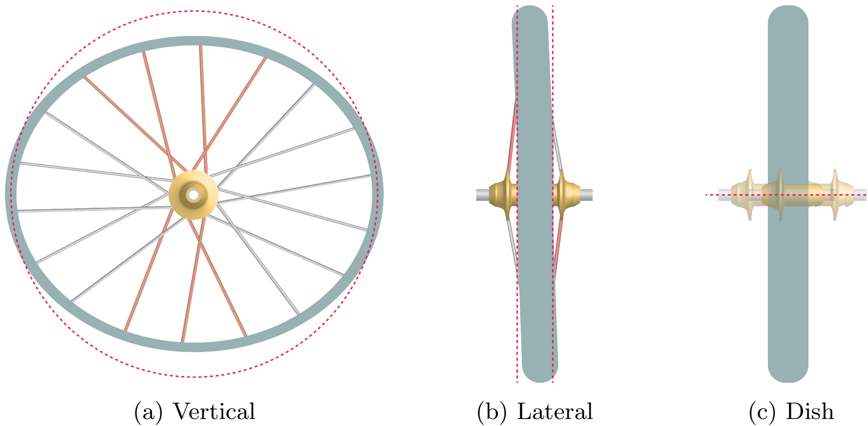
The forces on and by a stretched spoke can be described using *Young’s Modulus*.

$$F = \frac{EA_0 \Delta L}{L_0}$$

Where E is the modulus of elasticity for the material, A_0 the cross-sectional area through which the force is applied, ΔL the change of the object’s length and L_0 the original length of the object.

2.4.3 Truing the Wheel

As mentioned in the previous subsection, a vital part of how a bicycle wheel works is the fact that all the spokes are tensioned. This is done by tightening the spoke nipples, and *truing* a wheel is the art of tightening each nipple the right amount. There are four main goals when it comes to truing, the three first affecting the geometry of the wheel are shown in figure 2.18.



(a) Vertical (b) Lateral (c) Dish

Figure 2.18: Wheel truing, red spokes are too tight. (a) needs vertical truing, it looks like an egg, (b) needs lateral truing, the wheel will wobble from side to side, (c) shows dish, how the hub can be adjusted sideways.

- *Getting a round wheel*, also called vertical truing, is of course beneficial. Figure 2.18a shows a wheel that is too tight and resembles an egg. When rolling, the distance between the hub and the ground would constantly change, making it a very shaky ride.
- *Removing wobble*, also called lateral truing, is to tighten the spokes in such a way that the wheel looks straight when viewed from the front of the bicycle. Figure 2.18b shows a wheel that is trued incorrectly in this manner, being too tight on one side some places, and too loose on other places. So when rolling, this wheel would constantly move from side to side.
- *Dish*, is to tighten the spokes on both sides in a way that controls the relative positioning of the hub in relation to the rim. Depending on how

the wheel should be mounted, one might want the rim aligned exactly above the hub or to one of the sides. Figure 2.18c shows the axis on which the hub can be moved.

- *Proper tension in the spokes.* Too little tension in a spoke means it does not contribute to bearing the load, so having too many under-tensioned spokes will make a wheel dysfunctional. If the tension is too high, the spoke may fail. It is also beneficial to have as little variation in the tension as possible for spoke. Changes in tension under different loads can lead to metal fatigue in the spokes.

These four goals are intertwined, adjusting the spokes to fix one issue will often make another one worse. For instance, loosening the red spokes in figure 2.18b to remove wobbling will also affect how round the wheel is.

2.5 Literature Review

This section lists and discusses related work and how relevant papers were found. For related work, the focus has mainly been papers using evolutionary multiobjective algorithms to optimize mechanical structures. Some papers in the problem domain are discussed as well.

2.5.1 Related Work

The paper **"Evaluation of Injection Island GA Performance on Flywheel Design Optimisation"** [5] by Eby et al. used a layered island model in order to speed up the search for a good flywheel design. Island models are a way to parallelize an EA, by splitting the population into smaller groups (islands) that run the EA almost independently of each other. It is different from running several smaller EAs, because with islands there will be *migration* once in while where copies of the best solutions are distributed to the other islands. Compared to a standard EA, with only a single "island", this can help overcome local optima. By chance, some of the islands will get past the local optima, or become stuck in different local optima. Migration will then mix solutions from the islands and often help the search continue.

An interesting aspect of the island model in this paper is that the different islands use different fitness functions and other parameters. To evaluate the evolved flywheels they use finite element analysis (FEA), which takes a long time. To not waste much time running FEA on bad solutions, the top layers run a simpler

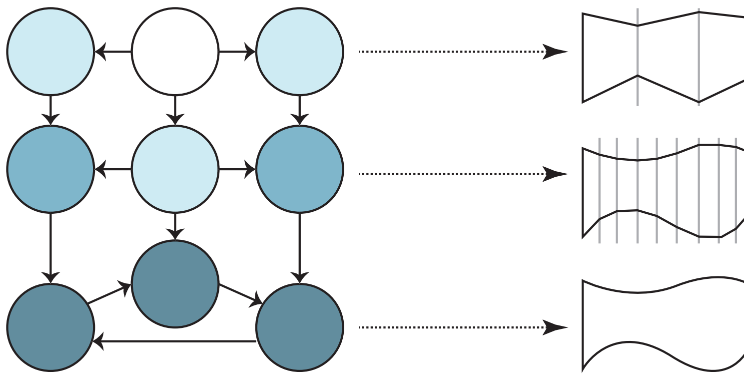


Figure 2.19: Topology of the islands, the darker shaded islands run a more accurate FEA.

version of the FEA on them. The layers have different levels of refinement, the bottom one running a full FEA. Figure 2.19 shows the topology of the islands. The result is that the top layers quickly discards bad solutions, and send the good ones to the lower layers to be refined, saving a lot of time. Their normal EA could not find a solution in 6000 seconds, while this approach found the global optima in 768 seconds on average. Their conclusion is that this approach can make good solutions attainable for problems where it takes a long time to evaluate a solution.

"A Hybrid Multi-objective Evolutionary Approach to Engineering Shape Design" [6] by Deb and Goel tries to find an optimal shape for a bicycle frame. The two objectives are the weight of the frame and the stiffness, and they use the already discussed NSGA-II algorithm to select solutions for the next generation. They have a two-dimensional grid, and the genotype encodes which cells in the grid are solid or not. These shapes are then evaluated with FEA. In order to improve the results, a simple local search is done on each solution when the EA is done. For this local search, they combine all the objectives to one aggregated value and flip one and one bit in the genotype. If the flip makes an improvement, it stays that way. Their results show that this additional optimization in the end moves the solutions closer to the pareto-optimal front.

For a non-dominated sort to work and explore the whole pareto-optimal front, many solutions to choose from each generation are needed. This also means that in the end the result is hundreds of solutions with different tradeoffs. Selecting between them can be hard for a human to do, so after the local search Deb and Goel tries to narrow it down to a few, interesting solutions. This is done by clustering the solutions in the objective space, and remove all but one in each

cluster. They then end up with a few near-optimal solutions that are spread out on the non-dominated front.

"Many-Objective Evolutionary Optimisation and Visual Analytics for Product Family Design" [8] by Shah, Reed, and Simpson. They try to find an optimal configuration of design variables for a small airplane, like cruise speed and seat width. There are nine objectives to optimize, ranging from takeoff noise to purchase price. They want to end up with three different airplane models with different tradeoffs in order to cover most of the market. However, the airplanes should be somewhat similar as this reduces production costs, so a tenth objective is added: commonality between the design variables. As discussed earlier in this chapter, having many objectives makes the search exponentially harder, because most of the solutions become non-dominated to each other.

To solve this 10-objective problem, they use ε-NSGA-II, an extension of NSGA-II that uses a concept called ε-domination instead of the domination explained earlier in this chapter. For each objective, the user of the ε-NSGA-II algorithm can specify a precision for how solutions should be ranked for that objective. In the solution space, a big ε would be a coarse grid and a small ε a finer grid. Instead of a strict comparison to find domination, the values of solutions are first mapped to values in this grid. Since the ε varies between objectives, the search can be fine-tuned to look more into certain objectives while ignoring small changes in others. They exploit this property here to be able to optimize over 10 objectives. ε-NSGA-II also has an adaptive population size that saves computational cost when there is no need for a big population.

To check that the algorithm does not break down to a random walk, they test the results against 25 million results from a Monte-Carlo simulation. The results from ε-NSGA-II dominate these results, so they conclude that it works well. One point, however, is that the ε-NSGA-II does a total of 500 000 fitness evaluations each run. In this case, the objective values are simple to calculate from the design variables. However, in a problem where a heavier simulation is needed, 500 000 simulations can take decades to complete. Therefore, this approach of handling many objectives may not be feasible for all kind of problems.

The article **"Multiobjective Optimization of Space Structures under Static and Seismic Loading Conditions"** [23] by Lagaros, Papadrakakis, and Plevris tries several algorithms in order to optimize these space structures, one of them being a multi-objective EA. The other approaches they try are classical algorithms in the field of this application, modified to handle multiple objectives. This they have done by converting the values of the multiple objectives into a single scalar. One of the methods they used for this is the weighting method discussed earlier. The second method is to have a defined goal for each objective

and calculate the sum of how far away from the goal each value of a solution is. The last method they used is to convert all but one objective into constraints instead.

The result is that the EA finds as good solutions as the classical algorithms in an order of magnitude less time. For each solution, a costly finite element analysis has to be run. The EA needs about five times fewer such evaluations than the other algorithms, and this is what makes it so much faster. The authors conclude that the EA is so much better because there are multiple individuals that can search for multiple solutions in parallel and that these solutions cover the whole pareto-front without converging too early.

In **"Single and Multi-Objective Approaches to 3D Evolutionary Aerodynamic Design Optimization"** [7], Hasenjäger et al. optimize a gas turbine blade. The blades are simulated, and that simulation takes over 2 hours per solution. Therefore, it is in their interest to minimize the number of needed evaluations of individuals. For multiple objectives, approaches based on non-domination need a relatively large population, so they opt out of using NSGA-II. Instead, they try with an algorithm called ES-CMA. ES-CMA uses the search history to calculate the covariance between the objectives, and uses this model of the search space to adapt the search to promising regions. This decouples the population size of the search from the dimension of the search space (e.g. number of objectives).

Using this approach, they ran the algorithm for 300 generations with only ten individuals in the population. Using a cluster of 40 computers, this took six weeks to complete but gave good results. They also tried making the problem single-objective by aggregating the value of each objective into a single value. This did not work out very well, as the EA then ended up focusing too much on only small parts of the search space and quickly converged.

"Optimization of a Supersonic Airfoil Using the Multi-objective Alliance Algorithm" [9] is a paper from Lattarulo, Seshadri, and Parks, also related to the aircraft industry. As with the previous paper, they also need to run computationally heavy simulations, and therefore want an algorithm that works well even with few evaluations. For this, they have compared a relatively new algorithm, the Multi-Objective Alliance Algorithm (MOAA), against the widely used NSGA-II. MOAA uses the idea of tribes (individuals) that form alliances as a way to combine and improve solutions. The tribes have skills (their values in the different objectives) which they need to improve by creating an alliance with a tribe that has skills they do not have themselves. The new tribes in the next generation have their values first copied and then modified with a normal distribution with standard deviation σ. σ will decrease during the search, so the

search starts with high diversity and later on the diversity is lower, similarly to how simulated annealing does it.

To compare MOAA and NSGA-II, they use two of the indicators already discussed: epsilon indicator and hypervolume indicator. Then they perform a statistical test on this data to see if they show something conclusive. While they claim that the MOAA outperforms the NSGA-II after 1000 fitness evaluations, it is only by a little. They find comparably similar solutions, the biggest difference is that MOAA covers a bit more of the pareto-optimal front. Therefore, it can seem that when few evaluations are possible because of computational constraints, using the ES-CMA from the previous paper can be a better idea.

"Optimal seat and suspension design for a quarter car with driver model using genetic algorithms" [24] is a paper by Gündoğdu. There are many papers that deal with evolutionary algorithms and suspension, but many of them do a single-objective optimization. In this paper, the author also considers how a driver of the vehicle would react to rough conditions, both in terms of comfort and safety. Since he wants to minimize all of the objectives and ranks them all as equally important, he gets good results by combining the objective values into a single value.

Finite Element Analysis of Spoke Lacing Patterns [22] is a paper from Williams Cycling R&D, a bicycle manufacturer. They do not use any EA or any other form for search, they do however simulate four common lacing patterns. This project aims to extend what they have done by simulating many more unknown lacing patterns. In their study, they test 2x/2x, 3x/3x, 3x/radial and radial/3x pattern, all with 28 spokes. Before the slash is the pattern on the drive side, after the slash on the non-drive side. Interestingly, two of the wheels they test use different patterns on each side of the wheels. A figure showing different patterns can be seen on page 25.

They have built the wheels in a modeling suite called *ABAQUES CAE* and run a finite element analysis on them in there. They pretension the spokes on the drive side to 100 kgf and the non-drive side to 60 kgf. The difference is to adjust the dish of the hub. They then apply 70kgf torque to the drive side and as weight on the hub. They measure the highest and lowest tension seen in any spoke, the lateral deflection of the rim and power loss from the applied torque. The values from the FEA show that the wheel model works as expected: applying torque to the wheel increases the tension in the trailing spokes and decreases the tension in the leading spokes, and adding weight to the hub decreases the tension in the bottom spokes.

The 3x/3x wheel has the smallest range between highest and lowest tension in any spoke. This is good, as big changes lead to metal fatigue over time. The lateral

deflection of all the wheels is very small, only 0.18 mm to 0.23 mm, so it should have no big effect. As for power lost to wheel flex, the wheels with a radially laced side lose much more than those without. They attribute this to the radial spokes being unable to transfer any power, they only hold the rider's weight. They conclude the 3x/3x wheel is the best, and speculate that this happens because the spokes emerge from the hub at an angle nearly tangential to the hub flange. As this angle will be different for other number of spokes than 28, they believe that for a 32-spoked wheel a 4x/4x lacing pattern will be the best.

2.5.2 Structured Literature Review

To get a soft introduction in the field, chapters in various books were read. An introduction by Deb[25] was very useful in getting to know the core concepts. From there, searching for *evolutionary multiobjective optimization* gave several good papers to read, and checking the latest years' GECCO proceedings was helpful.

As for finding applications, search engines were first tried. Google Scholar and sometimes ACM and Springer were used. For the papers, there were several conditions they preferably should meet: Optimize more than one objective, use an evolutionary strategy and that the problem should be related to the design of some physical object, possibly involving a simulation. Terms like *multiobjective optimization, evolutionary algorithms, genetic algorithms, physical artifact, morphology, design, simulation, application, structural engineering* and more were tried in different combinations. The results were very variable, often too general, containing unwanted applications (for instance circuit design, training of ANNs). Better results were found by searching for survey papers of applications, go through them and there find applications/fields where EAs were used. Then search specifically for those applications, for instance, searching for *"evolutionary multiobjective optimization of airfoil"*.

Chapter 3

Methodology and Implementation

This chapter will discuss the implementation and the tests to be run. The implementation consists of mainly two parts: a wheel simulator and the evolutionary algorithm. After explaining the implementation of these two parts, a more detailed look is taken on how the problem is represented and solved.

3.1 The Wheel Simulator

In order to know how good a lacing pattern is, a wheel with that pattern must be simulated while being subject to some external forces.

3.1.1 Approach

Three possible approaches were considered when implementing the simulator.

Writing the Equations

The workings of the spokes and the whole bicycle wheel can be expressed as a set of equations. Setting up the equations for a bicycle wheel should not be that hard even without a deep understanding of physics, as it is mostly static

mechanical components coupled with forces calculated from Young's Modulus. However, setting up the equations correctly and automatically for the different lacing patterns can be more difficult.

Solving the system of equations may also prove hard, as it probably cannot be solved exactly. Because of all the forces working in various directions, it would need to be solved iteratively until a stable solution has been found. There exists software to help with this.

Physics Library

There are many physics libraries available, most of them written for games but also some for simulations for movies. Instead of writing the equations, one would here add various structures to the *world* the libraries give you. Then one can later add forces to the structures. In this case, a wheel would be built from spokes, rim-segments, and a hub, and then later forces will be added to the world. The library will then calculate what should happen within the world for us.

Physics libraries are often made with real-time performance in mind, which makes them fast. This speed is good when thousands of simulations will be run. They are also made to be integrated with code, which makes them easy to use. However, their functionality is often limited to static dynamics and simpler soft-body dynamics. For a mostly mechanical structure like a bicycle wheel, this may be good enough.

CAE Software

In the industry, Computer Aided Engineering software is used to design new products and improve existing ones. These programs have advanced built-in simulators that can do finite element analysis, computational fluid dynamics, and other simulations. This means that they can handle stuff the physics libraries cannot, for instance, stress analysis of components and advanced deformations.

The added features make the simulations very slow, for simple structures they use minutes and for more complex ones several hours. This can be a showstopper when evolving thousands of individuals that should be evaluated. The programs are often very advanced and hard to learn. The editing of structures happens inside the program, and it can be difficult to programmatically build structures resembling the individuals that should be simulated without a deep knowledge of the program.

Selected Approach

A small prototype was attempted using CAE software. However, it was quickly realized that these programs are huge, have a steep learning curve and can take years to learn. The goal here is not to only model *a* wheel, but to make a system that can evaluate thousands of them without any human intervention. It was deemed unfeasible to implement such a system in these kinds of programs without any prior experience.

Option 2 was selected: Using a physics library. While not being able to handle all the advanced stuff CAE software can do, it should suffice for this experiment.

The selected physics library, or *engine*, is Bullet Physics [26]. It has been used in big games like Grand Theft Auto and to make special effects in Hollywood movies like Shrek and How to Train Your Dragon. So it is well tested, and because of the huge user base also well documented with many code samples. It is written in C++ with the goal of high performance, so it is very fast.

In addition to the physics engine, the simulator uses a framework named Libgdx [27]. Libgdx is a game framework written in Java. It can act as a wrapper around Bullet, extending its capabilities and making it easier to use. Libgdx also has good support for loading 3D-models, rendering graphics, handling input etc., which are all things that are needed in addition to the physics engine.

3.1.2 Implementation

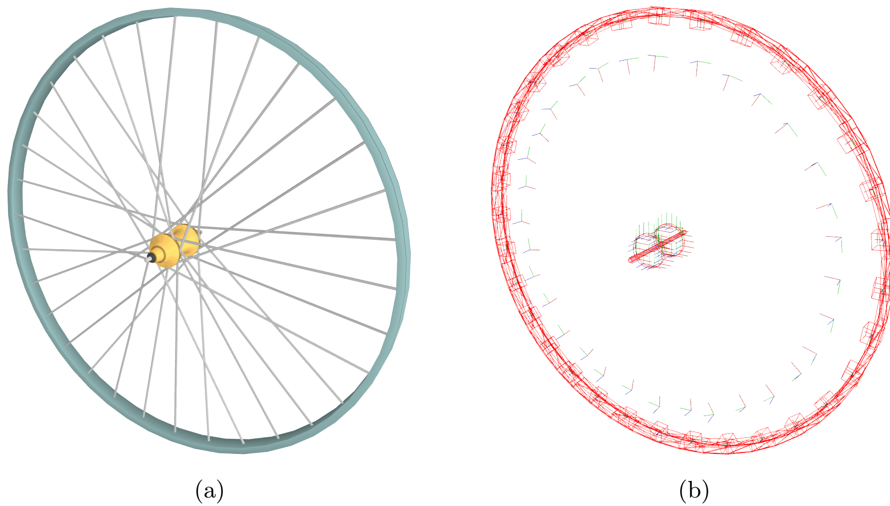
As discussed earlier, the force from a spoke is given by Young's Modulus.

$$F = \frac{EA_0 \Delta L}{L_0}$$

E is the modulus of elasticity for the material. A_0 is the cross-sectional area of the object being stretched. As we are only interested in spokes being stretched and tightened in one axis, this area will be constant. L_0, the initial length is also constant (for each spoke). By baking these constants into a single constant, k, we get:

$$F = \frac{EA_0 \Delta L}{L_0} = (\frac{EA_0}{L_0})\Delta L = k\Delta L$$

This formula is on the form of *Hooke's Law* for a spring. So the spokes can be modeled as springs, albeit really, really stiff ones. This simplifies the simulator implementation significantly, as Bullet (and most other physics libraries) has good built-in support for spring systems.



(a) (b)

Figure 3.1: (a) how the wheel looks, (b) how the physics engine sees it.

The rim is modeled as a stiff ring. This is harder than it sounds, as a real-time physics engine wants the collision shapes to be convex primitives. A convex primitive is a box, sphere, cylinder etc., and these are very efficient when doing collision calculations. The rim is not convex, however, and since we would like to have the hub inside it, we cannot approximate its shape using a short and wide cylinder. There are mainly two ways to fix this. The first one is to build a rim like shape using multiple convex primitives, and then make sure these align correctly with the rim when it moves. This approach is error-prone and inconvenient, but effective. A second way is to force the physics engine to not use a convex primitive, and instead provide it with a custom shape. This is easier to code, as one can then just upload the 3D model to the engine and tell it to use these edges and vertices as the collision shape. It is, however, less efficient as a lot of simplifications can no longer be made inside the engine. For this use-case it did not matter as much, as it was only for a single object, so the second approach was used. The shape of the rim can be seen in figure 3.1b.

The hub is modeled as a long, thin cylinder as the axle, and two wide, short cylinders as the hub flanges. This is also shown in figure 3.1b. The hub has a mass of 0, which means that it becomes fixed in the world.

The spokes are spaced uniformly around the hub flange. The positions of the spokes on the rim are proposed by the EA. The EA only gives positions/angles for one side of the wheel, the other side is mirrored and rotated slightly. Comparing

the two views of the wheel in figure 3.1, it is evident that the physics engine does not "see" the spokes as we do. Where the spokes are, only a small point is visible. Instead of being an object, the spokes act as spring forces in our physics world. Each spoke has a position on the rim and the hub, and the distance between those two points in regards to the start length of the spoke gives a force that pushes on the rim (as the hub is fixed in the center of the world). An added benefit of the spokes not being an object, is that they do not collide with each other. On a real wheel, the spokes would bend slightly to go around each other, but here they can go straight through.

3.1.3 Result

The simulator is able to correctly simulate bicycle wheels, with some limitations detailed below.

Figure 3.2 shows two different wheels where forces are applied to the rims from below. As can be seen, the behavior discussed in section 2.4 is present. The tension in the upper spokes increase, and it decreases in the lower spokes. It is not visible in the figure, but the rim moves further from the center for the 3x laced wheel than for the radially laced wheel under these forces. This behavior is also correct, as the spokes in the radially laced wheel are not angled as those in the 3x laced wheel, so it can better withstand the direct forces.

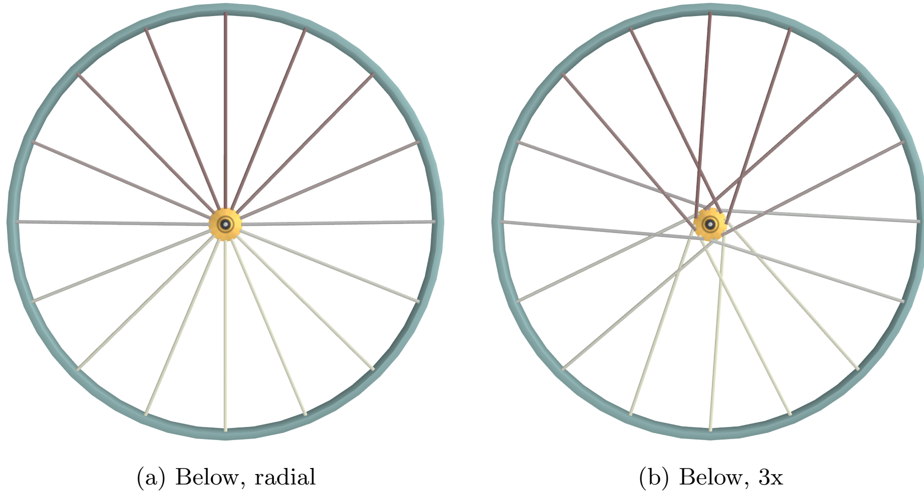
Figure 3.3 shows the same wheels, this time with torque applied to the rim, trying to rotate it. The 3x laced wheel is rotated less than the radially laced wheel. This is correct, as the angled spokes work better to stop the rotation, the radially laced spokes become twisted before stopping the rotation. In figure 3.3b, one can also see how it is the leading spokes absorbing all the force.

As the simulator is a proper 3D simulator, we can also see how wheels behave from the side. This would be the *wobble* described in section 2.4.3. Figure 3.4 shows a randomly generated pattern. From the side it does not look too bad initially, but when applying forces to it, one can see it becomes unusable. The simulator is able to correctly measure and then punish wheels for this.

Limitations

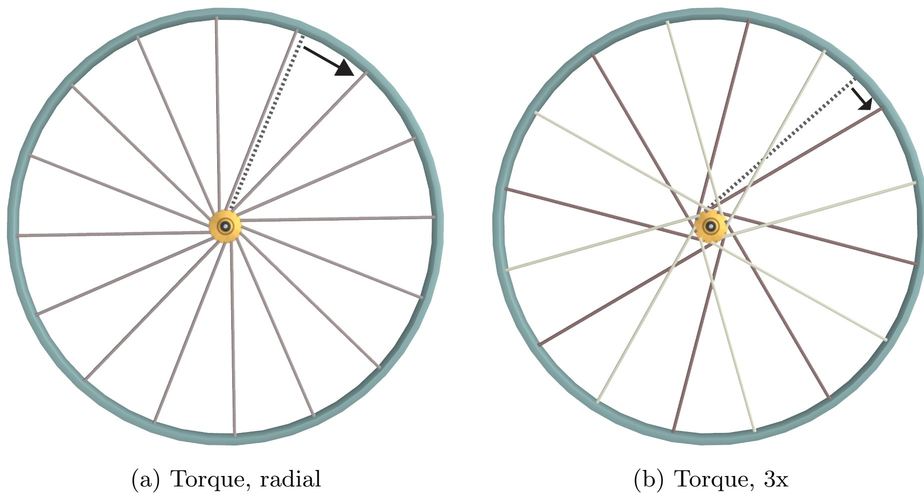
While the previous examples show that the simulator works fairly well, it is not perfect.

The biggest limitation is that it only simulates simple mechanical behavior. Some things it is unable to cover is stress in the spokes, deformation or destruction of



(a) Below, radial (b) Below, 3x

Figure 3.2: Applying a force on the rim from below on two different wheels. Red spokes are under high tension, yellow low tension.



(a) Torque, radial (b) Torque, 3x

Figure 3.3: Applying torque to the rim on two different wheels. Showing the distance the wheel rotates.

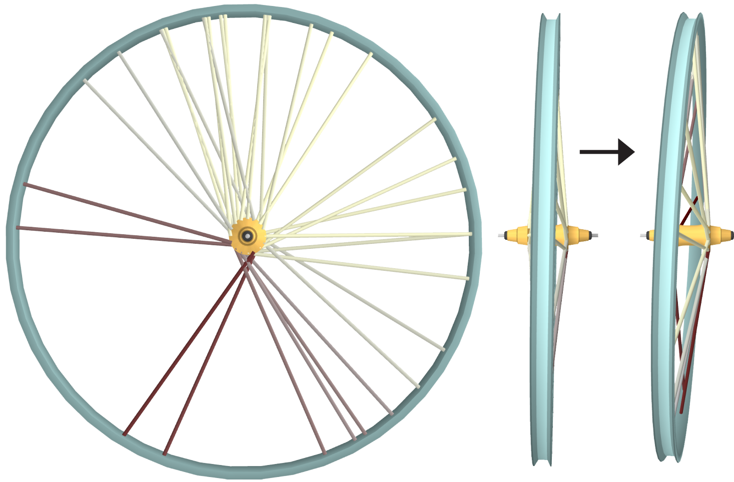


Figure 3.4: A randomly generated wheel from the front and from the side. It looks very unstable when applying forces to it (third image).

the objects during simulation, measuring wind resistance, and other advanced topics.

Some of these things are not that important to measure in this project. While wind resistance is something they take into account when building wheels for professional bicyclists, the lacing pattern of the spokes should not affect this drastically. However, if other parameters of the wheel were to be evolved as well, this might have been more important.

Stress in the spokes, however, is important. Constantly being put under different forces during various loads on the wheel will wear out the metal. While we cannot measure this directly in the simulator, we can instead measure the current force in each spoke at various intervals during the simulation and use these numbers to approximate something similar. Punishing a wheel with big variations in tension is essentially to minimize the stress.

The biggest downside of the simulator is that the objects in it are not deformable. This does not affect the spokes in the simulation, as they are modeled as spring forces. The rim, however, is bendable in the real world but not in the simulation. Figure 3.5 shows that the simulated wheel does not end up with an egg shape when the top and bottom spokes are tightened very hard.

An attempt to make the rim bendable was made, using Bullet Physics' *softbody* abilities. The attempt was unsuccessful, as the rim ended up being too bendable

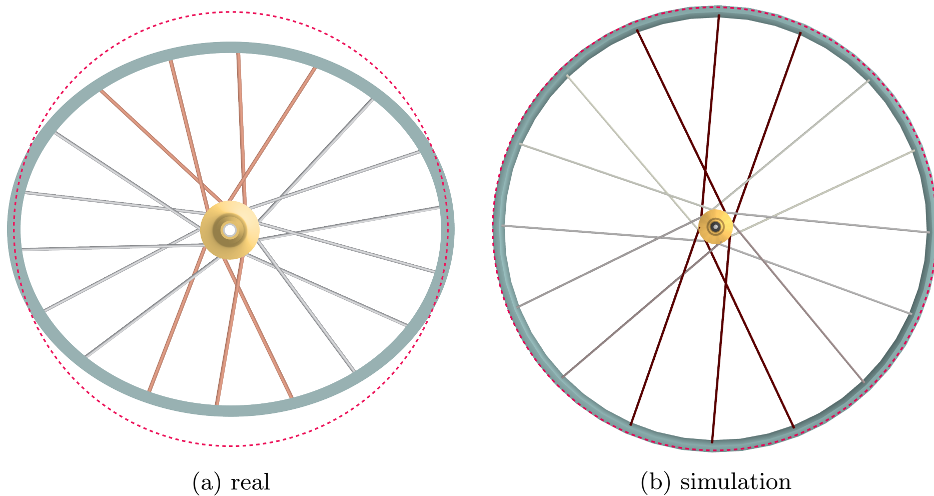


Figure 3.5: The rim in the implementation cannot be bent. Red spokes are tightened.

no matter what. It was more like rubber than metal, and so the results from using it was very bad. Therefore, it was decided to use a stiff rim instead, as it acts correctly in most situations. In the situation in figure 3.5, the wheel would anyway have been punished for having so tight and uneven spokes, so this limitation should not affect the outcome very much.

The simulator is "unitless". Because of floating point precision, the physics engine handles sizes in the range 0.5-20 best, otherwise it may become unstable. This is easily accommodated by scaling all units in the simulation by a fixed constant. However, having 32 well-tightened spokes exerting strong forces in multiple directions on the same object was something the physics engine did not particularly like. It was often unable to converge to a solution, instead it would start vibrating and eventually explode. To fix this, small tweaking of the various lengths and forces has been done. This should not affect the behavior of the simulator, as already shown it is mostly correct, but means that the values from the simulator cannot be directly compared with values from the real world or other simulations.

To conclude, there are things the simulator is unable to simulate, but these things often have correlated effects that *can* be simulated and measured. So, what the simulator thinks is a good or bad wheel should be fairly accurate.

3.2 Evolutionary Algorithm

The evolutionary algorithm is the second core part of the implementation. It is responsible for finding the best wheels and does so by testing each wheel it comes up with in the simulator.

3.2.1 Approach

There were two possible approaches here: write everything from scratch, or reuse existing implementations from elsewhere. A lot of the things needed for this project, like the different crossover and mutation operators, the genotype representations and some of the algorithms are fairly standard. Reusing these from other projects would free up time for the more interesting aspects of this project, and there is no need to reinvent the wheel.

Therefore, some time was spent looking at existing EA implementations. While many have written their own EA and published it, few have written one with multiobjective capabilities. After some searching, MOEA Framework [12] was chosen. It is open source, which makes it possible to modify and/or extend its features to fit the needs of this project. It is written specifically for solving multiobjective problems using evolutionary algorithms, and the structure of the source code looked like it would be easy to modify.

The MOEA Framework has a lot of the needed features out of the box: NSGA-II, NSGA-III, most of the operators and the indicators. However, it still needed modifications to fit this project.

3.2.2 Implementation

The MOEA Framework contains a "diagnostics tool", which basically is a small GUI where one can select which problem and algorithm to run and view some results. This tool has been extended significantly.

The diagnostics tool was originally made to test the algorithms, and so the output from it is not very useful. It is just a list of numbers of the found solutions. Therefore, a way to visualize the evolved wheels were implemented. This visualization consists of a list of all the found non-dominated solutions. Selecting one, the wheel it represents is shown.

The number of solutions found can be in the hundreds. Many of these will represent wheels with almost the same scores in the various objectives. For a

human, wading through all these to make a final decision about what wheel(s) to use can be daunting. One cannot instruct the EA to only evolve a few wheels, however, as it needs a big population to guide the search.

The approach used by [6] Deb and Goel and discussed in section 2.5 has been implemented to alleviate this. When the EA is done, the user, or *decision maker*, has the opportunity to *reduce* the set of solutions to an appropriate size. This reduction is done by adding each solution to a separate group, and then merge groups that are close to each other until the number of groups is the same as the wanted size. Then the solution closest to the center of each group is found, and these solutions will be the reduced set. This set will then have most of the interesting tradeoffs, and a size that is small enough for a human to base decisions on.

A task manager has been implemented and added to the GUI. This allows the user to queue up multiple runs with different parameters and then return a few days later and view all of the results. This saves the user from having to constantly monitor the EA and start a new run when one is finished.

The MOEA Framework has also been extended with new "problems", that handle the setup of the wheel-optimization problems to run. A problem defines the wheel representation the EA should use, how it should evaluate a solution, what the reference set is and other problem specific things. Exactly what these problem definitions are for this project is defined later in this chapter.

In addition to this, a way to parallelize the EA has been implemented.

3.2.3 Parallelization

For each generation in an evolutionary algorithm, possibly hundreds of individuals have to be evaluated. The algorithm is also run for hundreds of generations, and then the algorithm itself is run 10-20 times to get enough statistical data. With each wheel taking almost a second to simulate in the simulator, this can take a very long time.

To speed up the EA, it is natural to consider parallelization. Since each individual in an EA is evaluated independently, this part of the algorithm is very easy to do in parallel. There is always some overhead when parallelizing, but in cases like this, where each evaluation takes a long time, this overhead is negligible in comparison.

One way to do this would be to use island models, as used by Eby et al. [5] and discussed in section 2.5. With this, separate EAs would be run on different com-

puters, and once in a while they would communicate and *transfer* good solutions to each other. However, what made using this island model so good, was that they could run a simple evaluation in some of the islands, and only pass good solutions to islands running a more complex evaluation. Since the implemented simulator really only runs evaluations at one ”level”, this will not give as good results for this project. So the implementation of this was scrapped

Master/Slave

Instead, a simpler master/slave model was used. The EA works as a master, requesting its slaves to evaluate the various lacing patterns it comes up with. This layout is shown in figure 3.8 later in the chapter.

The slaves are just a simple wrapper around the wheel simulator. This wrapper will start the simulator, and then start asking the EA for wheels to evaluate. When it gets a lacing pattern, it gives the pattern to the simulator, waits until the simulator is done and return the results to the EA. This communication is done via sockets, and the data sent back and forth are JSON strings of the values. When starting a slave, one has to provide the IP-address of the computer running the EA, to initialize the connection.

The master is a bit more advanced. To make the distribution as transparent as possible, a *Distributer* poses as an evaluator. However, instead of evaluating any solutions, it adds them to a queue. When a slave connects to the master, a WorkerCommunicator is created in the master. This Communicator constantly polls the queue for unevaluated solutions. If it finds one, the solution is sent to the slave it is communicating with for evaluation. When the results of this evaluation arrive, the Communicator makes sure it is coupled with the correct solution. When the queue is empty and all solutions have been evaluated, the Distributer will pass the solutions and their results to the problem being run, to get assigned the objective values.

The results were very good. For instance, distributing the work to 12 slave simulators reduced the time spent per generation by a factor of about 11. This speedup made it possible to run more tests than planned, and run each test multiple times to get statistically significant data.

3.3 Representations

This section contains the details of how the problem is represented and how the wheels are tested.

3.3.1 Wheel Tests

Test Scenario

A wheel will be tested in a scenario in the simulator. After each step, *SimulationData* is recorded. This data is the current position and orientation of the rim, and the force in each spoke. Figure 3.6 shows the axes and rotation in the simulation world.

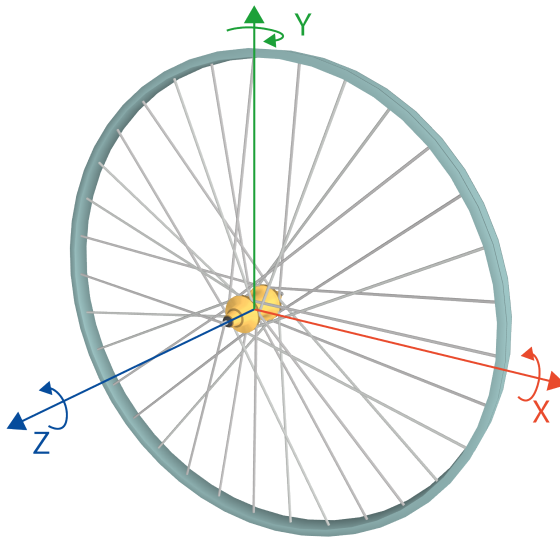


Figure 3.6: The axes of the wheel. Showing both axes of translation and rotation.

The wheel will be subject to various forces during the scenario. One force is applied along the Y and X axes, to test the strength of the wheel. The reason for doing multiple tests of the same force from different directions, is to make sure

the EA do not end up building wheels that are very strong in only one direction, as EAs are known to exploit loop-holes like this. The torque forces will test how well the wheel can handle breaking or pedaling. Between each force, the wheel is reset by simulating it without any external forces for a couple of seconds. The time units are in respect to the time inside the simulation world.

The scenario:

- Simulate the wheel just hanging from the hub for 1 second, to make it converge to an initial position.
- Gather data for the initial position.
- Apply force on the rim **along the X axis** for 3 seconds, gather data.
- Reset by waiting 1 second.
- Apply force on the rim **along the Y axis** for 3 seconds, gather data.
- Reset by waiting 1 second.
- Apply force on the rim **opposite direction of the X axis** for 3 seconds, gather data.
- Reset by waiting 1 second.
- Apply force on the rim **opposite direction of the Y axis** for 3 seconds, gather data.
- Reset by waiting 1 second.
- Apply **torque** to the rim **around the Z axis** for 3 seconds, gather data.
- Reset, this time wait 3 seconds as it takes longer for the wheel to return to the initial position after being rotated.
- Apply torque to the rim **around the Z axis**, but the **opposite direction** this time, for 3 seconds.

When the scenario is finished, we have a list of SimulationDatas.

Simulator Output

To make meaning of all the data, some calculations are done on it. These calculations give what is considered the result of the simulation, and will be called *SimulationResult*.

The SimulationResult contains:

- **MaxDstXY**, which is the max distance the rim moved from the center in the XY-plane during the scenario.
- **MaxDstZ** is the max distance the rim moved in the Z axis *from the initial position.* This is done because the *dish* of the wheel is unimportant. If the rim starts with a dish to the right, that is fine as long as the rim stays to the right during the scenario.
- **MaxRotX**, **MaxRotY** and **MaxRotZ** is the max rotation the rim had. MaxRotZ is similar to MaxDstZ as in that we calculate it from the initial rotation. MaxRotZ tells how well the wheel could withstand the torque applied in the scenario.
- **SpokeMax** is the highest force seen in any spoke at any time during the scenario.
- **SpokeMaxDiff** is the biggest difference in forces any spoke had.

3.3.2 Problem Representation

Two problem representation will be used, one having four objectives and one having two. These will be named OBJ_4 and OBJ_2, respectively. The reason for testing with two different number of objectives is to see what difference this makes when solving the problem. Is it really necessary with a many-objective representation, or could it be solved with a simpler, multi-objective representation and algorithm? The more objectives, the harder the search will be. However, representing a problem with fewer objectives means that it may not be able to optimize it in all aspects. So there is an interesting tradeoff there.

For both representations, the goal is to *minimize* the various objectives.

OBJ$_4$

Based upon the values from a SimulationResult, the four objectives are:

- **Dst** = $\sqrt{MaxDstXY^2 + MaxDstY^2}$, combining the distances to one using Euclidian distance.
- **Rot** = MaxRotX + MaxRotY, combining two of the rotations to one. This combined value tells how much the wheel is displaced while under external forces.

- **RotZ** = MaxRotZ, keeping this as a separate objective, as one of the steps in the scenario is to rotate the wheel around the Z axis. This makes the rotation in Z measure something else than the other rotations, namely how well the wheel withstands torque.
- **Spokes** = SpokeMax + SpokeMaxDiff, combining the two spoke measurements to one.

The reason for "only" using four objectives, not all the seven values in the SimulationResult, is that many of these values are positively correlated. E.g., an increase in MaxDstXY would most likely also lead to an increase in MaxDstZ as well, so combining them makes sense.

The range of the variables will vary greatly. For instance, the Dst will be somewhere around 0.5 while the forces in the spokes can be around 3500. Some algorithms may optimize based on the best difference in absolute value, instead of relative value. For instance, optimize the Spokes value with 0.12, when optimizing Dst with 0.11 would be much better relatively speaking. To combat this, one can normalize the values based on an already known range for them. However, the creators of NSGA-III say that their algorithm can handle these differences fine [10], so no tweaking will be done.

$\mathbf{OBJ_2}$

The two objectives are defined as:

- **Displacement** = 25 * MaxDstXY + 1000 * MaxDstZ + 800 * MaxRotX + 900 * MaxRotY
- **Strength** = 1 * MaxRotZ + 0.0045 * SpokeMax + 0.0066 * SpokeMaxDiff

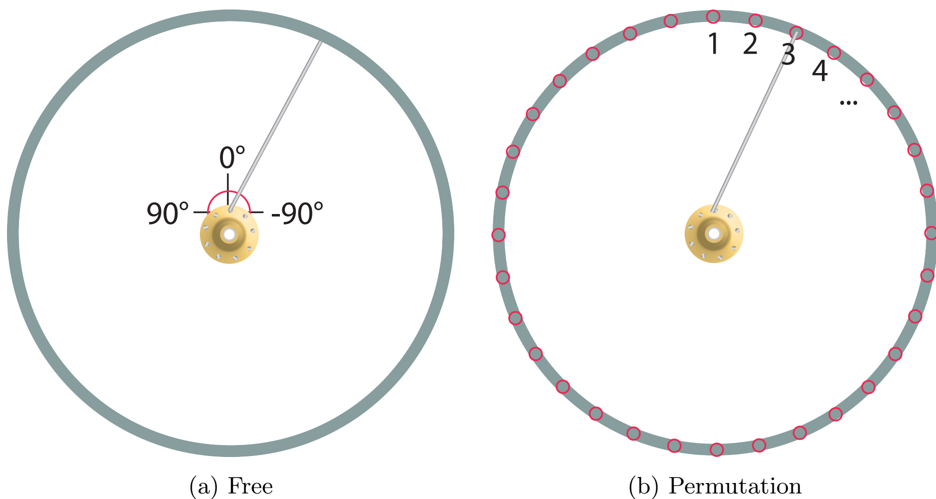
For the four-objective problem it was possible to not scale or normalize the values. However, for the two-objective problem, multiple values with different ranges are *combined* to a single value. If nothing is done to the values prior to combining them, the biggest value will dominate the others and become more important to optimize, biasing the search.

The constants are those deemed best after extensive testing. First, ball-park figures for the constants were calculated using the known ranges for the values by solving the OBJ_4 problem, before they were tweaked during multiple runs.

3.3.3 Wheel Representation

There are also two different representations for the wheel. In both representations, the spokes are placed uniformly around the hub. The difference is how the spokes are connected to the rim. The representations only tell how spokes on one side of the hub should be, and then the other side is just a mirrored and slightly rotated version of this. So for a 32-spoked wheel, the representation would represent the 16 spokes on one side.

The first representation is called REP_{free} as this representation allows almost all kinds of wheels to be evolved, for better or worse. The representation is a list of real values, one value for each spoke. This value indicates the *angle* of the spoke. It can have a value from -90 to 90 degrees, where 0 degrees means the spoke is straight. See figure 3.7a for a visual description. Real rims have pre-drilled holes for the spokes. This representation gives spokes that connect to the rim in all kinds of places, and may thus not be possible to build without special-ordering a rim.



(a) Free (b) Permutation

Figure 3.7: The two representations used for the wheels. In (a), each value indicates the angle of the spoke. In (b), each spoke goes to a spot on the rim, based on the permutation. In this case, the spot in the permutation for the spoke had the value 3.

The second representation has been named REP_{perm}, as it is based on a permutation genotype. For a 32-spoked wheel, the genotype is 16 unique values in a permutation. Each number corresponds to a position on the rim, all of them being uniformly laid out. The number in each spot in the permutation then says which position on the rim the spoke should go. For instance, for the representation [2, 4, 3, ...], the first spoke should go to position 2 on the rim, the second spoke to position 4 etc. An example is shown in figure 3.7b. These wheels are possible to build with existing rims if one has spokes of correct lengths.

The second wheel representation is believed to give better results. It can only generate lacing patterns where the spokes are uniformly spaced on the rim, so the representation has a bias for symmetry. And for a wheel, symmetry is probably a good idea.

Still, it is interesting to see what the EA can find with no bias towards what is believed to be good solutions. EAs are known to find unconventional solutions that exploit the problem, so giving it a chance to fully explore the problem is a good idea. Comparing the performance of the two representation, one can also see how important the representation is when it comes to getting good results.

3.4 Overview of the system

Figure 3.8 shows the system as a whole. It is not exactly a class diagram, it focuses on showing the core parts and their interactions on a higher level. The EA Controller is the core of the EA, handling the setup and the running of everything. The architecture makes it possible to combine freely various problem representations, wheel representations (and their operators) and algorithms.

When a new slave is started on a remote computer, it connects to the Distributer and a new communicator for this slave is created. The diagram shows the interactions to make this distribution work: When an algorithm has a population to evaluate, it gives the population to the distributor, which adds it to a queue. The communicators take these solutions and send their angles to the slaves. On the slave, the test scenario is run. The SimulationData is converted to SimulationResult before being returned. The Distributer takes this result and asks the Problem being run to convert it to objective values, which is then returned to the Algorithm.

The intended use-case is to programmatically add ”tasks” to be run and then start the EA using the user interface. The EA will then run multiple times with different parameters, problem representations, etc. based on the defined tasks. When everything is done, plots of the various indicators can be shown. It is also

possible to open up a window containing a list of all the evolved lacing patterns and test them in the wheel simulator.

A short video of some of this, most importantly showing the workings of the simulator, can be seen at http://master.matsemann.com/. The video is also included in a zip archive as part of the delivery of the thesis.

3.5 Experimental Plan

This subsection lists the experiments to be run and their configurations.

3.5.1 Parameters

There are a lot of variables and parameters, both in regards to the simulator and the EA. For the simulator, most of these are simply defined, but for the EA extensive testing had to be done to find the parameters yielding the best results.

Evolutionary algorithm parameters:

- *Population size of 100.* A smaller population gave better results early in the run, but would quickly get stuck. A bigger population would converge very slowly, but not find any better solutions than a population of 100.
- *Run for 200 generations*, for a total of 20 000 wheel evaluations each run. After 150 generations, most runs have found the best solutions they can find and are stuck.
- *52 remote simulators* were used during the runs.
- *Run each algorithm/variation 20 times.* Since evaluating a wheel in the simulator takes a lot of time, the runs need hours to complete. The distributed architecture is what made it possible to run each algorithm so many times, which should give enough data to draw conclusions based on the results.
- *Simulated Binary Crossover and Polynomial Mutation* are used when the wheel representation is REP_{free}. For SBX, the crossover rate is 1, meaning that all solutions will be subject to crossover. As both the NSGA algorithms contain an archive of the best non-dominated solutions so far in the run, picking a solution and not modifying it is a waste as we then get duplicates. Therefore, the crossover rate is high. For the PM, the rate is 1/16, so on average one variable will be mutated each time. The distribution index,

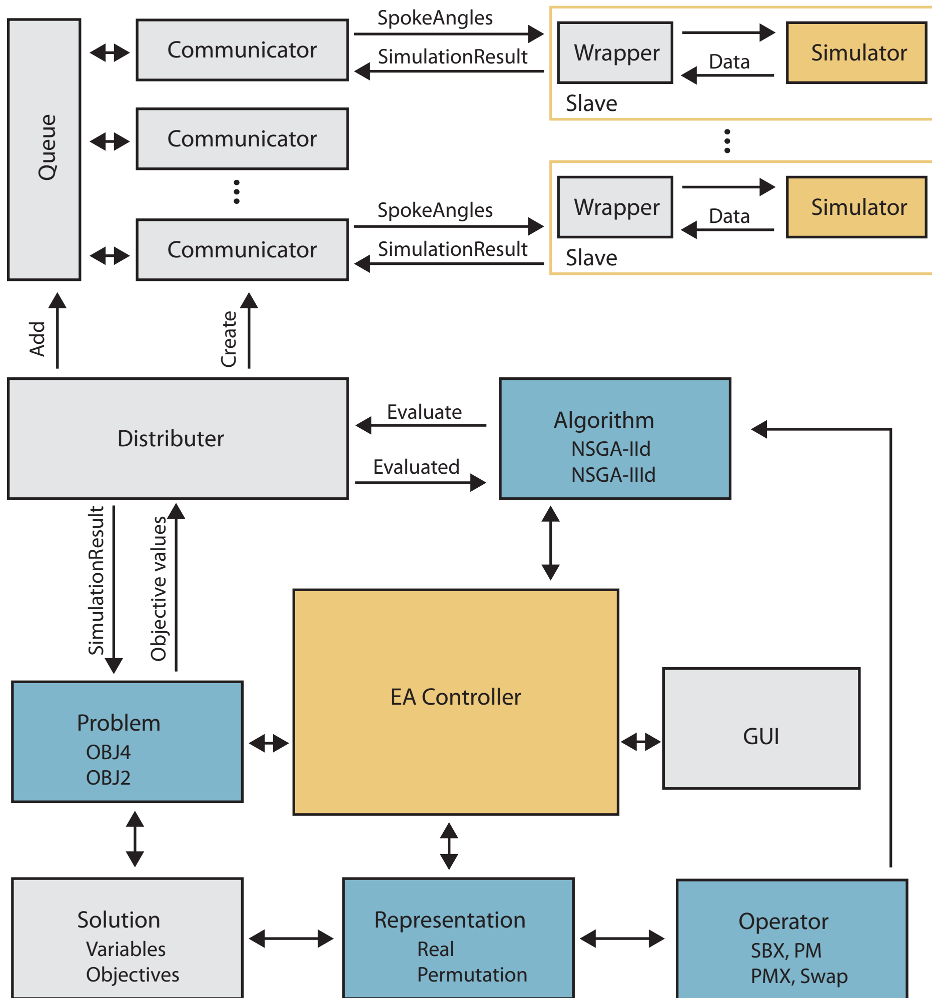


Figure 3.8: Overview of the system. The simulator and the core of the EA are highlighted in orange. The architecture allows combinations of the blue parts to be used.

discussed in section 2.2.1, is 20. This makes the mutation keep the values close to the original values, with a small chance of bigger variations.

- *Partially Mapped Crossover, Swap and Insertion* are used when the wheel representation is REP_{perm}. The PMX rate is 1, with the same reasoning as for SBX. The rates for swap and insertion are both 0.3, as this gave the best results during testing.
- *NSGA-III divisions is 4.* NSGA-III divides the objective space, as discussed in section 2.3.3. The number of divisions was initially set based on a table in the paper introducing the algorithm [10], and this value ended up being the best one.

Wheel and simulation parameters:

- *32 spoked wheel*, as it is one of the more common types and a nice, round number.
- *Simulated with 60 iterations per second.* A higher frequency increases the accuracy of the simulation, but also the computational cost. There were no perceivable differences with more than 60 iterations per second.
- *Force of 10 000 F* was applied to the wheel during the scenario.
- *Torque of 45 000 F* was applied to the wheel during the scenario.
- *Spoke stiffness 1500 F*, this is the k in the formula for a spring presented earlier. The spokes are tensioned 1 unit in length.
- *Wheel dimensions:* Hub flange radius is 2.25 units, the distance between the two flanges on the hub is 5.5 units. The radius of the rim is 31.25 units. These values are proportionate to a real wheel, just using the simulator's units.

3.5.2 Experiments to Run

These experiments should answer the research questions defined in the introduction: investigate how well an evolutionary multiobjective algorithm can solve the problem, test the performance of NSGA-III, and compare the various approaches.

Solving the Problem

By running the EA hundreds of times, one should get a lot of data on how well an EA can solve the problem. Analyzing the problem domain and the solutions is interesting.

NSGA-II vs. NSGA-III

Compare the results of using the newer NSGA-III algorithm compared to the older NSGA-II. Based on the results in the paper introducing NSGA-III and the claims that NSGA-II has troubles handling more than two objectives, NSGA-III should give better results than NSGA-II. For this test, REP_{free} and OBJ_4 will be used for both algorithms.

Wheel Representations

REP_{free} will be compared to REP_{perm}. REP_{perm} is believed to give better results, as the wheels it produces are biased to be somewhat symmetric. It is however interesting to see how much difference there is, and if the free representation can come up with something novel.

Number of Objectives

OBJ_2 and OBJ_4 will be compared. The best solutions found using OBJ_2 will be recalculated to get OBJ_4 values, this allows us to compare the solutions. This will show if there is any need to model the problem as a many-objective one, or if two objectives would be enough. The belief is that four objectives should be best, allowing the algorithm to optimize the wheels in all aspects.

Chapter 4

Results and Discussion

This chapter displays and discusses the results achieved from the tests introduced in the previous chapter.

For the plots showing the solutions in the objective space, the ranges of the various axes may vary greatly. This is not an attempt to make the plots show spurious relations, but happen because the ranges of the various objectives are different and not normalized. Also, one should be careful when making assumptions based on the plots in the objective space for higher dimensions, as will be done here, as we only see projections of the higher dimensional space onto two dimensions.

The three indicators discussed in section 2.3.4 will be used when comparing various approaches. For the hypervolume indicator, the goal is to get a high value. For the epsilon and spacing indicators, the goal is to get a low value. To make the comparisons, 20 runs of each variation have been done in order to be able to make conclusions. While it really is the end results of the indicators that matter, graphs will show how they progressed throughout the runs. The main line is the median of all the runs and the shaded area covers the first to the third quartile, so it can be seen as a progressing box plot. Before making any conclusions about the indicators, the Kruskal-Wallis test will be used on the data to make sure any differences are statistically significant.

4.1 Solving the Problem

The main goal of the project is to solve the problem of finding good lacing patterns. There are several objectives that should be minimized, in this analysis the OBJ_4 problem representation has been used. The solutions being analyzed are the solutions in the *reference set*. This is a set containing all the non-dominated solutions ever found when running the EA. This includes all kinds of runs using all representations and algorithms. The differences between representations and algorithms are discussed later.

4.1.1 The Problem Domain

To explore the problem domain, each objective is mapped against the other objectives to see how they relate. As discussed above, it can be hard to conclude from these, as we only see slices of a four-dimensional space. The six combinations of objective pairs and their plots are shown in figure 4.1. A small reminder about the objectives: *Dst* is the distance the rim moves from the center, indicating the strength of the wheel, *Rot* the rotation of the rim during forces, *RotZ* how much the wheel rotates around the Z axis when torque is applied, *Spokes* how well distributed the load on the spokes are.

Plot (a) in figure 4.1 shows Dst mapped against RotZ. The conventional wisdom about bicycle wheels is that one can have a tangentially laced wheel, like 3x, and have a good (low) RotZ, or one can have a radially laced wheel and have a good Dst. But not have both at the same time. Therefore, one would expect the trend to follow an asymptotic line, where a decrease in one leads to an increase in the other. However, the trend line in plot (a) seems to show something else. It looks like wheels with arbitrary combinations of RotZ and Dst can be made, without any tradeoffs between them. Should we rewrite the books about wheel building, or is there something wrong with the simulator? A closer look on the solutions says that neither of these options are true. The solutions that have low values for both RotZ and Dst have incredibly bad values for the two other objectives. If we remove solutions where Rot and Spokes are too high, the remaining solutions lies more along the dotted line in plot (a). Unconstrained the EA is able to build wheels that by the looks of it defy the conventional wisdom in wheel building. However, if the solutions are constrained, the known relation between Dst and RotZ shines through.

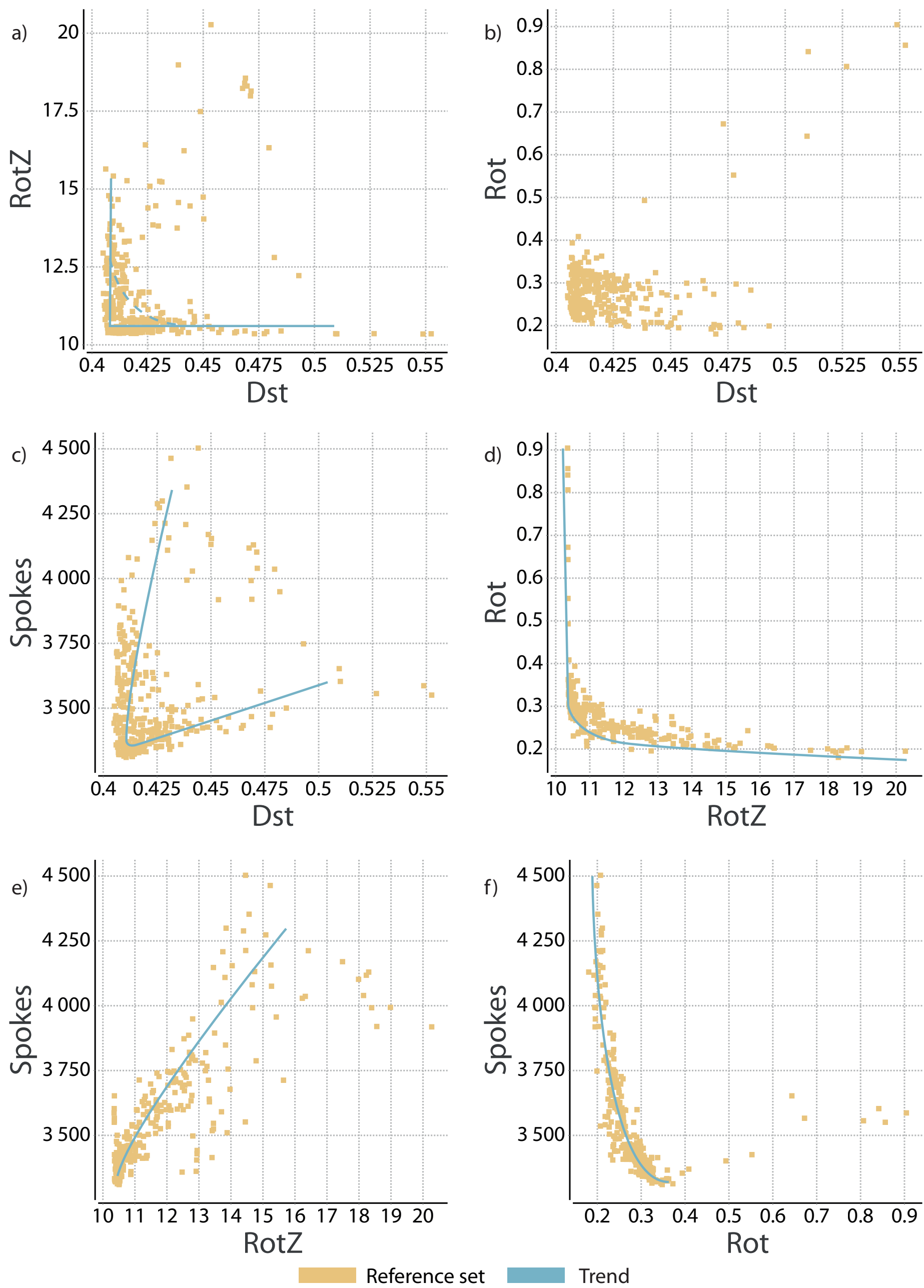


Figure 4.1: Best solutions plotted in the objective space.

Plot (b) shows Dst mapped against Rot. They are a bit lumped together, but not in any particular shape. Investigating the plot further, by removing solutions that were very bad in other objectives, did not expose any underlying relation. By the looks of it, these two objectives are independent.

In plot (c), one can see that Spokes and Dst are positively correlated. A decrease in one leads to a decrease in the other. This makes sense, as to get a low Dst, a wheel needs many spokes working together to withstand the forces. Moreover, many spokes working together will also distribute the tension between the spokes, leading to a low Spokes value. It is interesting to see that the values decrease together from two directions. This happens because of tradeoffs in the other, unseen objectives.

Rot vs. RotZ is shown in plot (d). One objective can be decreased much without any big increase in the other until a certain point is reached. After this point, an improvement in one leads to a huge disimprovement in the other.

In plot (e), we again see a positively correlated relation, this time between Spokes and RotZ. The reasoning here is similar to the one for Spokes and Dst. The wheel needs several spokes working together to withstand the torque applied to the rim to not be rotated that much. This also leads to a low Spokes value.

The last plot, (f) shows Spokes mapped against Rot. Here we can see that a small decrease in Rot leads to a huge increase in Spokes.

An interesting remark is that while a decrease in Dst gives a decrease in Spokes (c), and a decrease in Spokes gives a decrease in RotZ (e), a decrease in Dst does not give a decrease in RotZ (a). This also happens with other combinations of objectives. So the relations are clearly not transitive in this regard.

4.1.2 The Evolved Wheels

The EA was successful in evolving good lacing patterns, even given the computational constraints of evaluating each wheel and the incredibly huge search space. Some evolved wheels can be seen in figure 4.2.

As can be seen in the figure, the EA managed to find the *3x* laced wheel discussed earlier. It ended up as one of the non-dominated wheels in the reference set. As any other wheels did not dominate it, the EA did not find a wheel that can be said to be *better* than the 3x laced wheel already used heavily on today's bicycles. So, while no breakthrough in the bicycle world has been made, this proves that the wheels evolved match real-life and that this approach worked well.

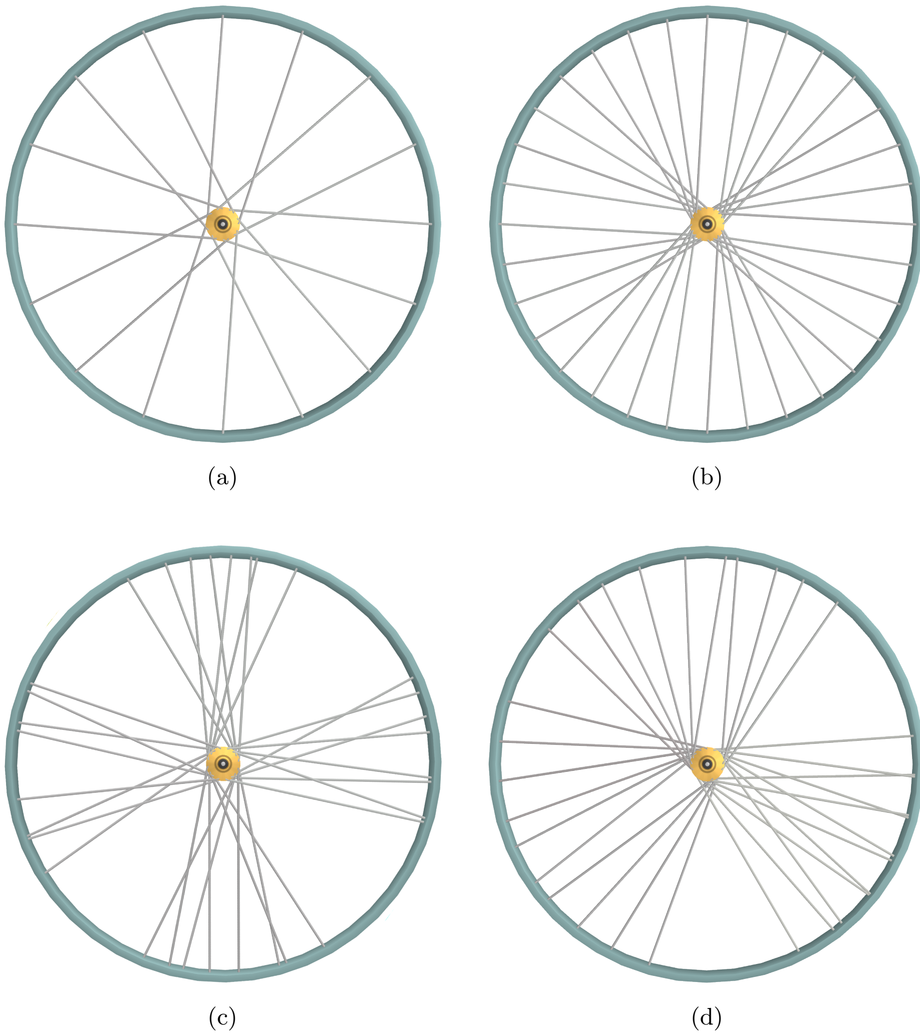


Figure 4.2: Some of the evolved wheels. (a) the EA found the 3x wheel, here only showing spokes on one side, (b) an interesting wheel with good tradeoffs, (c) a wheel exploiting the simulator, (d) an unconventional wheel with okay results.

The 3x laced wheel found has low values for Spokes and Rot. It also has a low value for RotZ, but only a medium good value for Dst. The wheel in figure 4.2b is also non-dominated, and therefore as good as the 3x wheel in the eyes of the EA. This design is interesting, it has four groups of spokes where the spokes in each group cross each other. Compared to the 3x wheel, this gives a better Dst value, but a slightly worse RotZ. So the tradeoff here is strength vs. withstanding rotations.

The weird looking wheel in figure 4.2c is also non-dominated. It has a low value for Dst, and it is easy to see why: The strength of the wheels are tested in the simulator by pushing from four directions. This wheel has been evolved to have all spokes pointing in these four directions, countering the pushes very well. This wheel fails in all other objectives, so no one would probably ever use a wheel like this. This wheel has been included to show some of the pitfalls of evolutionary multiobjective optimization. Namely that a wheel only focusing on one objective will become non-dominated but not very good, and that the evolutionary process is good at exploiting the fitness assignment procedure, in this case, the simulator.

The wheel in figure 4.2d is another non-dominated wheel found by the EA. It has some interesting properties. Compared to the other three wheels shown, this is far from having any kind of symmetry. It has a medium Dst value, a low Spokes and RotZ value, but a very high Rot value. So the wheel is good in most regards, except that it wobbles a bit from side to side. This is probably the lack of symmetry that makes the wheel a bit angled.

A couple of hundred additional non-dominated wheels have also been found. Many of these wheels are very similar to each other, though. This similarity happens because a small change to a wheel can make it a bit worse in one objective, but also a bit better in another objective. Then the new wheel is not dominated by the previous wheel. For the wheels shown in figure 4.2 there exist several non-dominated wheels with only small variations. The *reduce* functionality added to the EA was very helpful. It made it possible to reduce the reference set to only a few solutions, showing most of the various kinds of wheels evolved and the different tradeoffs.

Additional wheels and the objective values for the shown wheels can be seen in appendix C.

4.2 Comparing NSGA-II to NSGA-III

The new NSGA-III was compared to the older NSGA-II over 20 runs, each using the same problem representations and parameters.

4.2.1 Results

Figure 4.3 shows the three indicators throughout the runs. All of them improve until about evaluation 15 000 (generation 150 with a population size of 100) where they stagnate. The plots also show a close-up of the end results, which is what really matters. All of the end-of-run values can be seen in appendix B.

The median lines for hypervolume are almost identical, and according to the Kruskal-Wallis test they are indifferent. So we can conclude that the hypervolume is probably equal.

For epsilon, the NSGA-II is much lower than NSGA-III. The test says that there is a significant difference, so NSGA-II is better.

For spacing, the median of NSGA-III is lower than the median of NSGA-II. However, the box showing the first to third quartile is much bigger for NSGA-III. Because of this, the test says that the median actually may be equal, and that one cannot draw any conclusions that NSGA-III is better.

So to sum it up, the algorithms are deemed equal for spacing and hypervolume, while NSGA-II has a better epsilon value.

4.2.2 Discussion

The difference between NSGA-III and NSGA-II is how they select solutions within a given rank. NSGA-II sorts them based on the size of a cuboid and picks those with the largest one, while NSGA-III tries to pick solutions uniformly from the objective space using reference points. More details can be found in section 2.3.3.

NSGA-III was thought to be best. In the paper introducing it, it is tested on multiple example problems and is there clearly much better than the other algorithms. However, the results here show differently. NSGA-II is actually slightly better at this problem. What is causing this needs to be discussed.

One theory may be that the fitness landscape, discussed in the previous section, erases some of the differences between NSGA-II and NSGA-III. NSGA-II will sometimes have problems handling more than two objectives, which does not

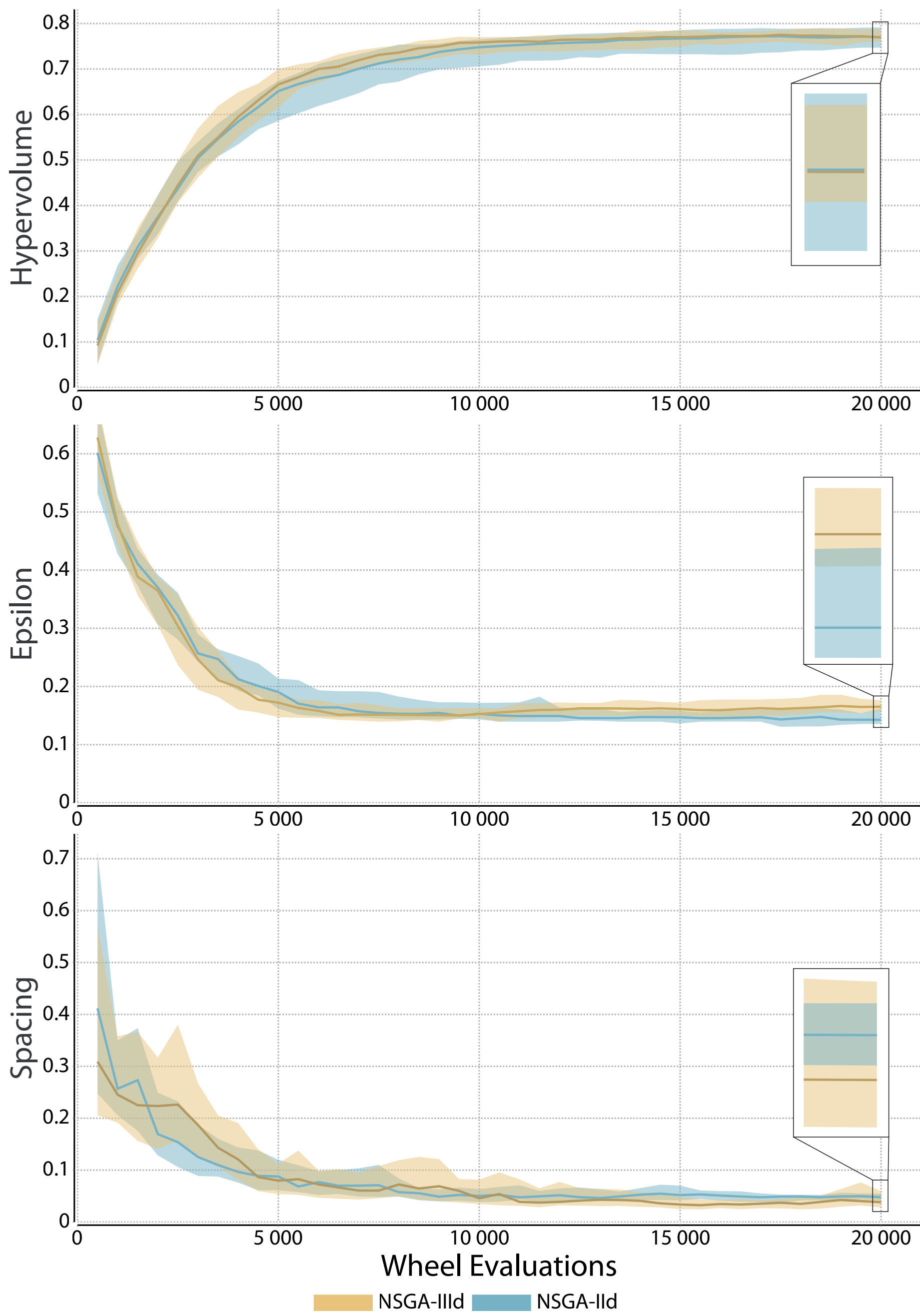


Figure 4.3: Plots showing the indicators during multiple runs of NSGA-II and NSGA-III. Zoomed in on the end result.

happen here. This may be because not all objectives are negatively correlated, as shown earlier. So the problem may not be as hard as a four-objective problem where there are strict tradeoffs between each objective, making the NSGA-II over-perform. The NSGA-III algorithm was in the paper mostly tested on example-problems where the objective space is very uniform. The objective space here is clearly not uniform or nicely shaped, which may render its way of picking solutions useless.

One challenge to the validity of the results is that there is no reference implementation of NSGA-III, while one exists for NSGA-II. This means that the correctness of the NSGA-II implementation used has been verified to be correct, but not the NSGA-III implementation. However, the NSGA-III implementation used has been tested on the same example-problems used in the paper introducing it, and the results on those problems were similar to those claimed. Therefore, it is believed that the NSGA-III implementation should be mostly correct.

A second challenge could have been that NSGA-III is better, but that given the number of generations both the algorithms converge to similar solutions over time. For instance, the NSGA-III could have found optimal solutions after 50 generations, and NSGA-II first after 200 generations. If the results were measured after 50 generations, NSGA-III would then have been deemed better. However, the plots in figure 4.3 clearly shows that both algorithms improve at the same rate and have converged long before the search is discontinued, so the results from the indicators should be valid.

While this discussion may explain why NSGA-III does not perform better, it does not excuse it. Other real-life problems will probably be similar to this problem: having a complicated, non-uniform objective space. This means that NSGA-III may not be an improvement over NSGA-II, other than on made up example-problems.

4.3 Comparing the Wheel Representations

The two representations, REP_{free} and REP_{perm}, discussed in section 3.3.3, have been compared.

4.3.1 Results

Figure 4.4 shows the three indicators throughout 20 runs of each representation. For both the hypervolume indicator and the epsilon indicator, REP_{perm} quickly

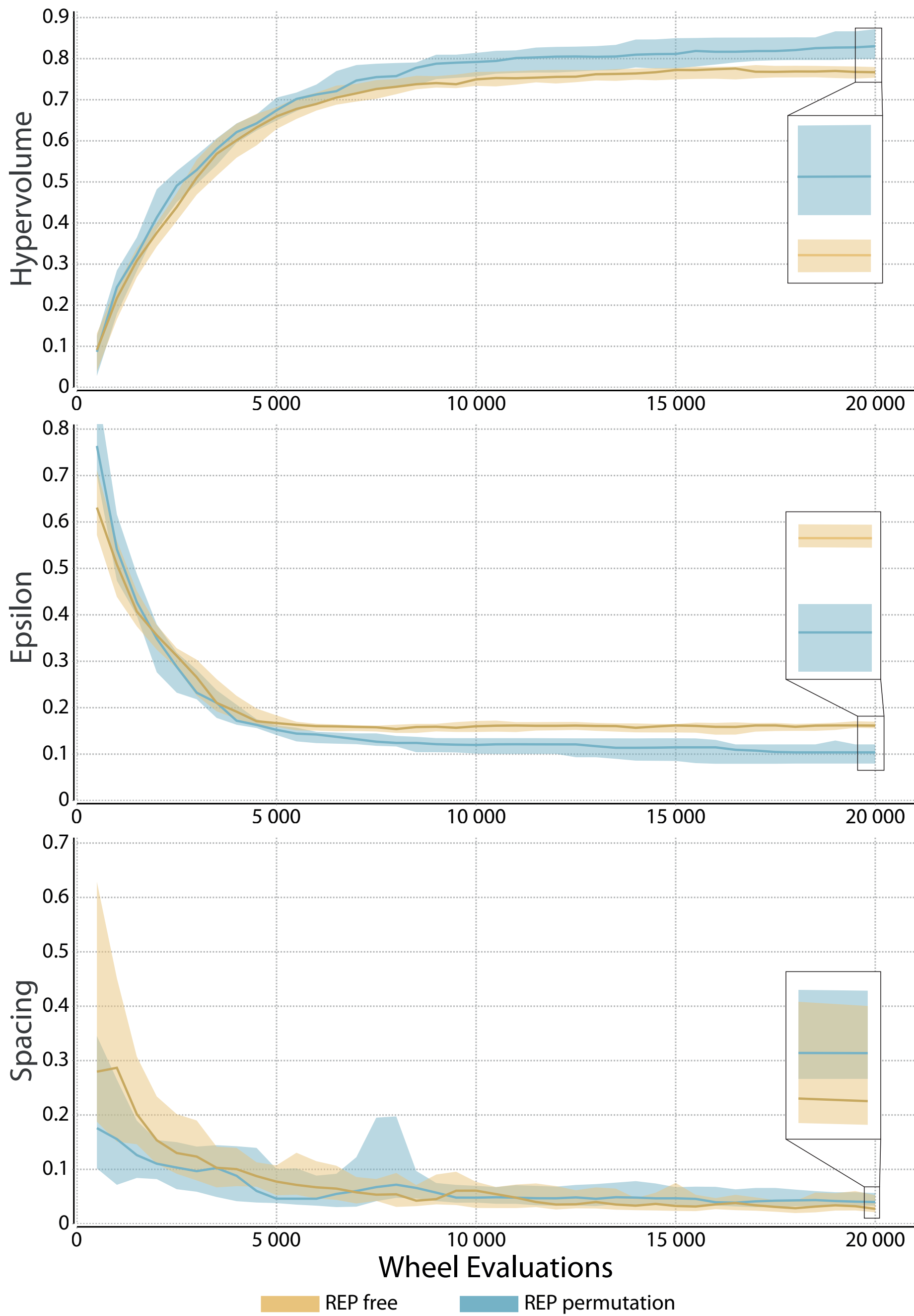


Figure 4.4: Indicators comparing REP$_{free}$ and REP$_{perm}$.

becomes better than REP_{free}. At the end of the runs, the difference is significant.

The spacing indicator starts with a higher value for REP_{free}, but halfway in the run both representations are fairly equal. By the end, REP_{free} is a bit lower than REP_{perm}, and according to the statistical test this difference is significant.

REP_{perm} is much better in hypervolume and epsilon, while REP_{free} has slightly better spacing values. The difference in spacing values is very small compared to the differences in the two other indicators. It is therefore concluded that an EA using REP_{perm} finds better wheels than an EA using REP_{free}.

4.3.2 Discussion

In REP_{perm}, all lacing patterns will have the spokes uniformly spaced on both the hub and the rim. Having this is probably a good idea for a wheel, all wheels in use today have it, so this bias in the representation is probably what makes it better.

It is also possible for the free representation to build wheels like this, but they are only a small part of the incredibly huge search space. The angle for each spoke in REP_{free} has 2^{32} possible values, and for 16 spokes this give a search space of $2^{32^{16}}$ possible lacing patterns. While there for REP_{perm} is "only" 16! patterns.

It is harder for the free representation to mutate a lacing pattern to a better one. Moving a single spoke in a good wheel will probably make it worse, as it is dependent on the position of other spokes. It becomes a kind of local optimum that needs big changes to get the wheel out of. So, to evolve a new and better wheel, multiple changes at once may be needed. However, the chance of mutating multiple spoke angles such that the wheel becomes a good one is very small. REP_{perm} does not have this issue, mutating it will make some bigger changes but at the same time preserve the uniformity of the wheel.

However, only being able to make these bigger changes is probably what makes the spacing value worse for REP_{perm}. The free representation can make small adjustments and thus create wheels that cover segments of the objective space the permutation representation is unable to.

The wheel in figure 4.2c shows an interesting aspect of the free representation. The EA has here exploited the fact that the simulator tests the wheel by applying forces from four directions. The free representation evolved several wheels that were very good in only a single objective, which was almost impossible with the other representation. While these wheels are almost useless, they can show interesting aspects of the problem being solved. The wheel just mentioned shows

that straight spokes lead to a low Dst value, while other evolved wheels show how angled spokes affect the RotZ value.

To conclude, the REP_{perm} finds better lacing patterns, most likely because of the smaller search space that has a bias for uniform wheels. This shows that the representation used when solving a problem is important. However, using a more free representation can also lead to interesting solutions not previously considered and reveal important relations in the underlying problem.

4.4 Comparing the Number of Objectives

Two different problem representations have been compared. The goal of this comparison is to see if there is any use in representing a problem using many objectives. The two representations are OBJ_4 and OBJ_2 discussed in section 3.3.2.

4.4.1 Results

Results from two different problem representations can not be compared directly, as they would be in completely different objective spaces. Therefore, to do this comparison the best wheels found using OBJ_2 has been recalculated and given an OBJ_4 value. Then these wheels can be compared to the wheels found using OBJ_4. However, since this recalculation of OBJ_2 wheels needs to happen after the EA is complete, there is no data on how the indicators progressed during the run. Box plots of the end result, comparing OBJ_4 and OBJ_2, can be seen in figure 4.5. Data for the plots are found in appendix B.

The hypervolume covered by the solutions found using OBJ_4 is clearly bigger than the one for OBJ_2 solutions. The epsilon indicator is fairly equal, with no statistically significant difference. The spacing looks to favor OBJ_4, but the Kruskal-Wallis test says that the true median may be equal based on the samples. Therefore, the representations are deemed equal in spacing.

4.4.2 Discussion

The OBJ_4 problem representation uses the output from the simulator almost directly. The OBJ_2, however, needs to combine multiple output values in order to only use two objectives.

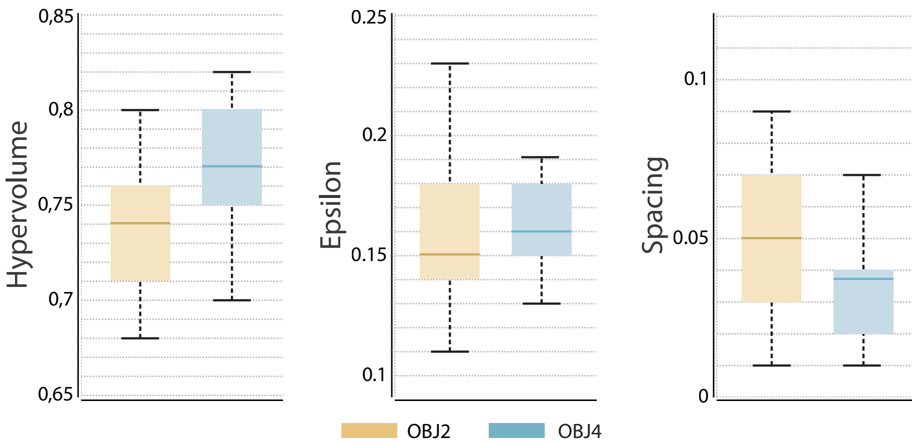


Figure 4.5: Indicators for OBJ_2 and OBJ_4.

These values are of fairly different ranges, as can be seen in the plots in figure 4.1. Just adding multiple values of different ranges would give bad results, as the value with the highest max value would dominate any differences in the other values. What has been done is to weigh each value when adding them, so that they should become equally important to optimize for the EA.

Finding the correct weights for each value to make objective values is very hard. In this case, it has taken hundreds of EA runs with different weights to find what gave good results.

Compare this to the OBJ_4, which needed no tweaking at all and gave better results, using more objectives is clearly the way to go. In addition, the only reason such good weights were found for OBJ_2 at all, is that the problem had already been solved using OBJ_4. This made it possible to analyze the problem and guess on initial weights, that later were iterated upon. Without this possibility, the search space for good weights would probably be too large.

This result is interesting, as it shows that research in many-objective (more than three objectives) optimization algorithms is important. Many of today's multi-objective algorithms have troubles when the problem has more than two to four objectives, depending on the problem. This experiment clearly shows that reducing the number of objectives gives sub-optimal results compared to representing a problem in its "true" number of objectives.

Chapter 5

Conclusions

This chapter is the conclusion to the report. The answers to the research questions are presented first. Then, a discussion about what has been done together with a comparison of what has been done in other papers is presented. Lastly, the contributions and future work are discussed.

5.1 Evaluation

5.1.1 Research Questions

The research questions posed in section 1.2 has formed the work and experiments done throughout the project. They are answered here based on the results in chapter 4.

Research Question 1
Can an evolutionary multiobjective optimization algorithm be used to optimize lacing patterns of a bicycle wheel?

The answer to this question is *yes*, based on the results in section 4.1. The evolutionary approach evolved wheels that were good, it even found lacing patterns commonly used today. The use of multiple objectives made sure that wheels with different tradeoffs were evolved.

Research Question 2
How does the performance of the new NSGA-III EMO algorithm compare to the older NSGA-II algorithm?

As the results in section 4.2 show, NSGA-III was not found to be better than NSGA-II in the various indicators. It was even worse in one indicator, the epsilon distance.

Various explanations for this were suggested and dismissed. Therefore, it is concluded that NSGA-II is better than NSGA-III for this problem.

Research Question 3
How do the representations of the problem affect the outcome?

Two types of representations have been tested, wheel representation and the number of objectives for the EA.

For the wheel representations, discussed in section 4.3, it was found that having a representation with a bias for uniformly spaced spokes gave the best results. This shows how important it is to thoroughly consider the solution representation to use, as it will determine the quality of the solutions found.

For the objective representation, two versus four objectives were tested. The results, found in section 4.4, shows that the higher number of objectives made the EA find the best wheels. This was not given, as a higher dimension also makes the search harder for the EA.

5.1.2 Summary

The system programmed together with the experiments conducted were able to answer the research questions. The system consists of mainly two parts, the simulator and the EA.

The simulator was able to simulate wheels in a manner mostly consistent with the theory presented in the background chapter. There were some limitations, mainly from the simulator being programmed using a physics engine instead of using complex CAE software. Section 3.1.3 discusses this in greater detail.

The EA built upon the work of the MOEA Framework [12], with several own additions and changes. One of the biggest changes was the introduction of a way to distribute the simulation of the wheels. Other approaches could have been some of those discussed in the literature review. Hasenjäger et al. [7] had a simulation that took over two hours per solution. To minimize the number of

simulations they used an algorithm that calculated the covariance between the objectives in order to adapt the search to promising regions of the search space. Eby et al. [5] used island models with various complexities of the simulation. Neither of these methods was used in this project. The covariance method was not needed, as the simulation in this case took much shorter time than two hours. The simulator could not run at different resolutions, so island models would not be useful as well.

Hasenjäger et al. tried to solve their problem using both a normal EA and a multiobjective one. They found the solutions from using multiobjective optimization to be best. This is in line with the tests in this project, where OBJ_4 was found to be better than OBJ_2.

Advanced variations (crossover and mutation) for the genotypes were used. Polynomial mutation and SBX were used for the real values. Swap, Insertion and Partially Mapped Crossover were used when the genotype was a permutation. These variations are some of the more commonly used variations, used in many other papers dealing with evolutionary optimization. The reduce procedure from Deb and Goel [6] was implemented. It reduces the solution set to a set containing the most interesting solutions, making it much easier to get an overview of the evolved solutions and the properties of the problem.

The results were tested to see if they were statistically significant before drawing any conclusions. This is needed because an EA is highly stochastic, and helps ensure that the answers to the research questions should be valid.

5.2 Contributions to the field

A complicated real-life optimization problem has been solved using evolutionary multiobjective optimization algorithms. It has been shown how important it is to consider the representation of solutions, as this can guide the EA more efficiently in the search. Representing the problem with more objectives gave better wheels, indicating that using a multiobjective approach should be done when possible, and that further research on these kinds of algorithms is important.

This thesis is one of the first applications using the new NSGA-III algorithm on a real-life problem, possibly its first use in optimizing a mechanical structure. NSGA-III was found to be no better than NSGA-II on this problem, which differs from the results of using these algorithms on academic example-problems. As it is the results on real-life problems that ultimately matter, the results presented here should be considered when selecting algorithms for future problems.

5.3 Future Work

5.3.1 Problem Domain

While the wheel simulator worked satisfactorily and was able to properly simulate the wheels, it still has some limitations as discussed earlier. What is possible to do using a physics library has probably been done, so to take this further, wheels should be simulated inside advanced CAE software.

Such software would make it possible to have bendable rims and spokes, measure air resistance and stress in the metal, and other topics not covered in this thesis. This is a significant increase in complexity, and the simulations would probably take hours.

It would also be interesting to model more parameters in addition to the lacing pattern. For instance, hub flange radius, the distance between the flanges, tire type, tire pressure, and radius of the rim. Most of these would not be possible to model inside the current simulator, or at least not lead to any measurable differences. However, a more advanced CAE simulation should be able to handle this.

5.3.2 EA

A more advanced simulator would also need changes to the EA. Fewer evaluations would be possible per EA run, as the time per evaluation increases. To overcome this, the covariance approach used by Hasenjäger et al. could be investigated further.

However, if one has two simulators, one simple and one advanced, using island models is a good idea. Instead of distributing simulators on multiple computers, as done in this project, one would here distribute separate EAs across the computers. Some of the EAs would run the simple simulator and pass good solutions found there to EAs running the more advanced simulator.

One of the drawbacks of using multiple objectives, and often basic EAs as well, is that a big population and, therefore, many evaluations are needed. For optimizing mechanical structures, this would often make an evolutionary multiobjective approach unfeasible. However, today's computers are powerful enough that running thousands of simulations is possible. More problems should be solved using an evolutionary multiobjective approach in the future, instead of manually testing a few variations in some CAE software.

As for algorithms, it would be interesting to see the performance of more algorithms on this problem in addition to the NSGA variants tested. It should also be investigated if the results of NSGA-III found in this project happen when used to solve other real-life problems as well, to determine if it really is no better than NSGA-II.

Bibliography

[1] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* 1st. Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0201157675.

[2] N. Srinivas and Kalyanmoy Deb. "Muiltiobjective Optimization Using Nondominated Sorting in Genetic Algorithms". In: *Evol. Comput.* 2.3 (Sept. 1994), pp. 221–248. ISSN: 1063-6560.

[3] K. Deb et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *Evolutionary Computation, IEEE Transactions on* 6.2 (Apr. 2002), pp. 182–197. ISSN: 1089-778X.

[4] Eckart Zitzler et al. *SPEA2: Improving the strength Pareto evolutionary algorithm.* 2001.

[5] David Eby et al. "Evaluation of Injection Island GA Performance on Flywheel Design Optimisation". In: *Adaptive Computing in Design and Manufacture.* Ed. by IanC. Parmee. Springer London, 1998, pp. 121–136. ISBN: 978-3-540-76254-6.

[6] Kalyanmoy Deb and Tushar Goel. "A Hybrid Multi-objective Evolutionary Approach to Engineering Shape Design". In: *Evolutionary Multicriterion Optimization.* Ed. by Eckart Zitzler et al. Vol. 1993. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 385–399. ISBN: 978-3-540-41745-3.

[7] Martina Hasenjäger et al. "Single and Multi-Objective Approaches to 3D Evolutionary Aerodynamic Design Optimization". In: *6 th World Congress on Structural and Multidisciplinary Optimization, Rio de.* 2005.

[8] Ruchit A. Shah, Patrick M. Reed, and Timothy W. Simpson. "Many-Objective Evolutionary Optimisation and Visual Analytics for Product Family Design". In: *Multi-objective Evolutionary Optimisation for Product Design and Manufacturing.* Ed. by Lihui Wang, Amos H. C. Ng, and Kalyanmoy Deb. Springer London, 2011, pp. 137–159. ISBN: 978-0-85729-617-7.

[9] Valerio Lattarulo, Pranay Seshadri, and Geoffrey T. Parks. "Optimization of a Supersonic Airfoil Using the Multi-objective Alliance Algorithm". In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*. GECCO '13. ACM, 2013, pp. 1333–1340. ISBN: 978-1-4503-1963-8.

[10] K. Deb and H. Jain. "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints". In: *Evolutionary Computation, IEEE Transactions on* 18.4 (Aug. 2014), pp. 577–601. ISSN: 1089-778X.

[11] Mohamed Wiem Mkaouer et al. "High Dimensional Search-based Software Engineering: Finding Tradeoffs Among 15 Objectives for Automating Software Refactoring Using NSGA-III". In: *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*. GECCO '14. ACM, 2014, pp. 1263–1270. ISBN: 978-1-4503-2662-9.

[12] Dave Hadka et al. *MOEA Framework*. Version 2.4. Jan. 2, 2015. URL: `http://www.moeaframework.org`.

[13] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Prentice Hall Press, 2009. ISBN: 0136042597.

[14] Kalyanmoy Deb and Mayank Goyal. "A combined genetic adaptive search (GeneAS) for engineering design". In: *Computer Science and Informatics* 26 (1996), pp. 30–45.

[15] Kalyanmoy Deb and Ram B Agrawal. "Simulated binary crossover for continuous search space". In: *Complex Systems* 9.3 (1994), pp. 1–15.

[16] David E Goldberg and Robert Lingle. "Alleles, loci, and the traveling salesman problem". In: *Proceedings of an International Conference on Genetic Algorithms and Their Applications*. Vol. 154. 1985.

[17] H. Jain and K. Deb. "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Nondominated Sorting Approach, Part II: Handling Constraints and Extending to an Adaptive Approach". In: *Evolutionary Computation, IEEE Transactions on* 18.4 (Aug. 2014), pp. 602–622. ISSN: 1089-778X.

[18] Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. "MOEA Testing and Analysis". In: *Evolutionary Algorithms for Solving Multi-Objective Problems*. Ed. by David E. Goldberg and John R. Kaza. Springer US, 2008. ISBN: 978-0-387-33254-3.

[19] Carlos M Fonseca et al. "A tutorial on the performance assessment of stochastic multiobjective optimizers". In: *Third International Conference on Evolutionary Multi-Criterion Optimization (EMO 2005)*. Vol. 216. 2005, p. 240.

[20] David C. R. Hunt. *Professional Wheelbuilding: The Manual*. DCR Wheels Publication, 2011.

[21] C. Burgoyne and R. Dilmaghanian. “Bicycle Wheel as Prestressed Structure”. In: *Journal of Engineering Mechanics* 119.3 (1993), pp. 439–455.

[22] Williams Cycling R&D. *Finite Element Analysis of Spoke Lacing Patterns.* 2014. URL: `http://www.williamscycling.com/RD_ep_39.html` (visited on 11/19/2014).

[23] Nikos D. Lagaros, Manolis Papadrakakis, and Vagelis Plevris. “Multiobjective Optimization of Space Structures under Static and Seismic Loading Conditions”. In: *Evolutionary Multiobjective Optimization.* Ed. by Ajith Abraham, Lakhmi Jain, and Robert Goldberg. Advanced Information and Knowledge Processing. Springer London, 2005, pp. 273–300. ISBN: 978-1-85233-787-2.

[24] Ö. Gündoğdu. “Optimal seat and suspension design for a quarter car with driver model using genetic algorithms”. In: *International Journal of Industrial Ergonomics* 37.4 (2007), pp. 327–332. ISSN: 0169-8141.

[25] Kalyanmoy Deb. “Multi-objective Optimisation Using Evolutionary Algorithms: An Introduction”. In: *Multi-objective Evolutionary Optimisation for Product Design and Manufacturing.* Ed. by Lihui Wang, Amos H. C. Ng, and Kalyanmoy Deb. Springer London, 2011, pp. 3–34. ISBN: 978-0-85729-617-7.

[26] Erwin Coumans et al. *Bullet Physics Library.* Version 2.82. Oct. 23, 2013. URL: `http://bulletphysics.org`.

[27] Mario Zechner et al. *Libgdx.* Version 1.5.3. Jan. 16, 2015. URL: `http://libgdx.badlogicgames.com`.

Appendix A

Source Code

This appendix gives a quick overview of the source code and how to run it.

A.1 Getting It

The code is included with the delivered zip archive, but can also be found online on http://master.matsemann.com/.

The project uses Maven to manage dependencies (libraries and frameworks used), compile, build and run the code. This means there is no complicated setup, one just has to have Maven installed, which is very common for Java developers.

A.2 Running It

Note: This describes how to compile and run the code, which is needed when changes to it are made. If you're only interested in running the code unmodified, pre-compiled code is included in the folder *compiled*.

The code can be run in three different modes. EA-mode launches the EA interface, in which the various problems and algorithms can be run. Remote-mode launches an external wheel simulator that evaluates the wheels the EA wants to test. The normal way to run this project is to start EA-mode on one computer and Remote-mode on multiple other computers. The algorithms ending with a *d* are the distributed ones, these have to be used for the wheel problems.

There is also a Simulator-mode that just displays a randomly generated wheel in the simulator. One can add forces to it by holding down "3" or "4" on the keyboard while pressing an arrow key. For more ways to manipulate the simulator, see appendix C.

To run the code, be inside the code folder in a terminal and run one of the following commands:

EA-mode: `mvn compile exec:java -Pea`

Remote-mode: `mvn compile exec:java -Premote`

Simulator-mode: `mvn compile exec:java -Psimulator`

A better way to run the remote than having to install Maven and compile the code on multiple computers is to build it:

Build the code: `mvn package`

This will generate a file named *ea-master-1.0-jar-with-dependencies.jar* inside the *target* folder. This file can then be copied to various computers and started like a normal program.

A.3 Overview

A small recap of the various folders and files inside the code folder.

`data` contains the 3D-models used in the simulation.
`pf` contains reference sets for the various problems.
`runs` contains saved runs of the EA, see appendix B.
`logs` contains logs, a new file is generated for each run, useful for debugging.

`src/main/java` contains all the source code:
`org.moeaframework` is the modified version of the MOEA Framework.
`com.matsemann` is the main part of the code, and the two big parts can be found in `com.matsemann.ea` and `com.matsemann.simulator`.
`com.matsemann.ea.ipc` is the code for distributing the simulations.

Appendix B

Results

B.1 Viewing Results Yourself

Launch the program in EA-mode, as described in appendix A. Go to *File*, select *Load* and select one of the runs in the *runs* folder. This opens all the data from the selected run. Graphs can be seen, the evolved wheels, the reference set etc.

B.2 Results From Runs

Listed here are the data from the various runs. The *indifferent* value states if the Kruskal-Wallis test applied to the values indicates any significant difference in the median values, and therefore if any conclusion can be drawn based on the median value.

NSGA-II vs. NSGA-III

Table B.1 shows the end values for all the runs comparing the performance of NSGA-II to NSGA-III. These results are discussed in section 4.2.

Comparing Representations

Table B.2 shows the end values for all the runs comparing the performance of the permutation wheel representation with the real valued representation. These results are discussed in section 4.3.

Comparing Number of Objectives

Table B.3 shows the end values for all the runs comparing the performance of OBJ_2 and OBJ_4. These results are discussed in section 4.4.

Hypervolume		Add. Epsilon		Spacing	
NSGA-II	NSGA-III	NSGA-II	NSGA-III	NSGA-II	NSGA-III
0.792341	0.785902	0.142572	0.157549	0.054284	0.020390
0.788778	0.765387	0.135868	0.165232	0.055828	0.178410
0.742326	0.811469	0.196139	0.118649	0.049734	0.026335
0.750077	0.781115	0.149589	0.159800	0.041457	0.032034
0.737772	0.761211	0.161956	0.167773	0.060676	0.048261
0.763218	0.779413	0.136006	0.178884	0.040546	0.021874
0.816137	0.727471	0.136882	0.191313	0.026539	0.092017
0.767956	0.755884	0.162983	0.185908	0.049378	0.034149
0.778395	0.768529	0.152345	0.169375	0.051005	0.031518
0.717989	0.788265	0.145641	0.130236	0.046393	0.051694
0.654709	0.765412	0.207078	0.159802	0.054472	0.030977
0.804211	0.769462	0.142861	0.161081	0.042559	0.052337
0.767035	0.761003	0.164100	0.157559	0.046455	0.086567
0.780184	0.746461	0.130729	0.181795	0.045492	0.023749
0.763997	0.791925	0.135452	0.166452	0.031378	0.031883
0.741336	0.722007	0.128106	0.200241	0.059819	0.042693
0.768274	0.763428	0.171967	0.175335	0.056579	0.059846
0.829704	0.781481	0.111139	0.176269	0.187506	0.028267
0.818666	0.792123	0.131891	0.164329	0.055505	0.050135
0.771278	0.791885	0.127689	0.149850	0.058297	0.150192
Median		Median		Median	
0.768115	0.768995	0.142717	0.165842	0.050369	0.038421
Indifferent		Not indifferent		Indifferent	
Equal		**NSGA-II better**		**Equal**	

Table B.1: End data from 20 runs of NSGA-II and NSGA-III using OBJ_4 and REP_{free}

Hypervolume		Add. Epsilon		Spacing	
Perm	Real	Perm	Real	Perm	Real
0.766810	0.807530	0.108305	0.161363	0.075869	0.025783
0.812854	0.745978	0.120957	0.170985	0.028123	0.038183
0.815733	0.789594	0.096083	0.140015	0.034760	0.030824
0.825263	0.773204	0.133837	0.150245	0.041292	0.067176
0.794687	0.781209	0.120957	0.155384	0.080057	0.016455
0.788488	0.776475	0.132177	0.169771	0.313430	0.019274
0.835741	0.773775	0.113931	0.169693	0.057940	0.038963
0.880343	0.817106	0.074257	0.149529	0.048995	0.009761
0.878272	0.763275	0.087284	0.166300	0.054193	0.027942
0.852341	0.749745	0.079034	0.183630	0.039999	0.016283
0.771241	0.757057	0.132362	0.189829	0.028332	0.018300
0.835206	0.776266	0.102818	0.161590	0.038527	0.096946
0.837898	0.763499	0.087284	0.174729	0.056240	0.022083
0.881057	0.727254	0.074257	0.160132	0.035774	0.022626
0.881951	0.755792	0.075276	0.165162	0.021298	0.026661
0.786276	0.766372	0.120957	0.161368	0.037380	0.023358
0.824144	0.766812	0.134928	0.171495	0.033226	0.042842
0.830325	0.753030	0.081466	0.161463	0.029020	0.035679
0.878856	0.769866	0.069813	0.159598	0.049521	0.070479
0.849566	0.736987	0.104552	0.151988	0.054762	0.056690
Median		Median		Median	
0.832765	0.766592	0.103685	0.161527	0.040646	0.027302
Not indifferent		Not indifferent		Not indifferent	
Perm better		**Perm better**		**Real better**	

Table B.2: End data from 20 runs of NSGA-III using OBJ_4, comparing the two wheel representations, REP_{free} and REP_{perm}

Hypervolume		Add. Epsilon		Spacing	
OBJ_2	OBJ_4	OBJ_2	OBJ_4	OBJ_2	OBJ_4
0.762617	0.756391	0.186639	0.146573	0.055548	0.054022
0.707475	0.822394	0.148014	0.144881	0.010365	0.225295
0.707147	0.727726	0.153151	0.187868	0.077926	0.018311
0.733011	0.750330	0.181934	0.177876	0.063909	0.020247
0.753991	0.754949	0.168507	0.178587	0.048784	0.035609
0.758128	0.696102	0.138511	0.155855	0.078254	0.047681
0.710637	0.801009	0.150210	0.164536	0.081851	0.041612
0.678579	0.779174	0.201423	0.172084	0.010908	0.035483
0.795440	0.789911	0.105239	0.176276	0.056774	0.016914
0.687864	0.751104	0.162191	0.186871	0.035397	0.032911
0.703862	0.811608	0.213132	0.149380	0.035104	0.016943
0.753552	0.784103	0.130345	0.141395	0.027111	0.041785
0.699824	0.792749	0.212317	0.151234	0.019424	0.094245
0.730803	0.745243	0.150115	0.177607	0.049806	0.023379
0.763905	0.810274	0.122665	0.153354	0.035346	0.069321
0.737178	0.759274	0.145356	0.130992	0.094366	0.023146
0.738307	0.791003	0.171760	0.156187	0.064188	0.027929
0.775155	0.762083	0.155799	0.188362	0.036773	0.035578
0.572808	0.759444	0.229853	0.193054	0.071854	0.013564
0.768433	0.803104	0.130404	0.144816	0.021570	0.040295
Median		Median		Median	
0.735094	0.770628	0.154475	0.160361	0.049295	0.035530
Not indifferent		Indifferent		Indifferent	
OBJ_4 better		**Equal**		**Equal**	

Table B.3: End data from 20 runs of NSGA-III using REP_{free}, comparing the two problem representations, OBJ_2 and OBJ_4.

Appendix C

Additional Wheels

This section lists some of the evolved wheels and their objective values. The values can be seen in table C.1. The value in the *Wheel* column refers to which wheel it is in figure C.1.

C.1 Testing a wheel

View the results, as described in an earlier appendix. Then right click a result and select "Show in Bullet Window". This will open a window where the evolved wheels can be selected and played with.

Controls for the wheel simulator:

- Forces: Hold down 3 and use the arrow keys to add forces from various directions.
- Torque: Hold down 4 and use the arrow keys to add torque.
- Spin: Press and hold R to rotate the wheel
- Camera: Press 1, 2 or 3 on the Num Pad to change views, or rotate it manually by pressing the left mouse button and dragging the wheel.
- Spokes: Select a spoke using the left mouse button. It can be tightened by pressing 7, or loosened by pressing 8.

Wheel	Rep	Dst	RotZ	Rot	Spokes
a	perm	0.40723	10.5278	0.30610	3352.3
b	perm	0.40598	12.4745	0.28871	3358.4
c	free	0.41560	10.4396	0.30693	3353.8
d	free	0.47300	10.3564	0.67224	3566.2
e	free	0.44340	10.3938	0.32402	3411.7
f	free	0.46970	18.3041	0.18043	4129.6
g	perm	0.40666	10.8717	0.31941	3349.2
h	perm	0.40689	11.1454	0.26173	3465.0
i	perm	0.40910	14.4501	0.23177	3551.8
j	perm	0.40960	15.4155	0.21348	3956.4
k	perm	0.44138	16.2284	0.20759	4028.6
l	perm	0.41027	10.4140	0.42311	3351.5

Table C.1: Objective values for some evolved wheels.

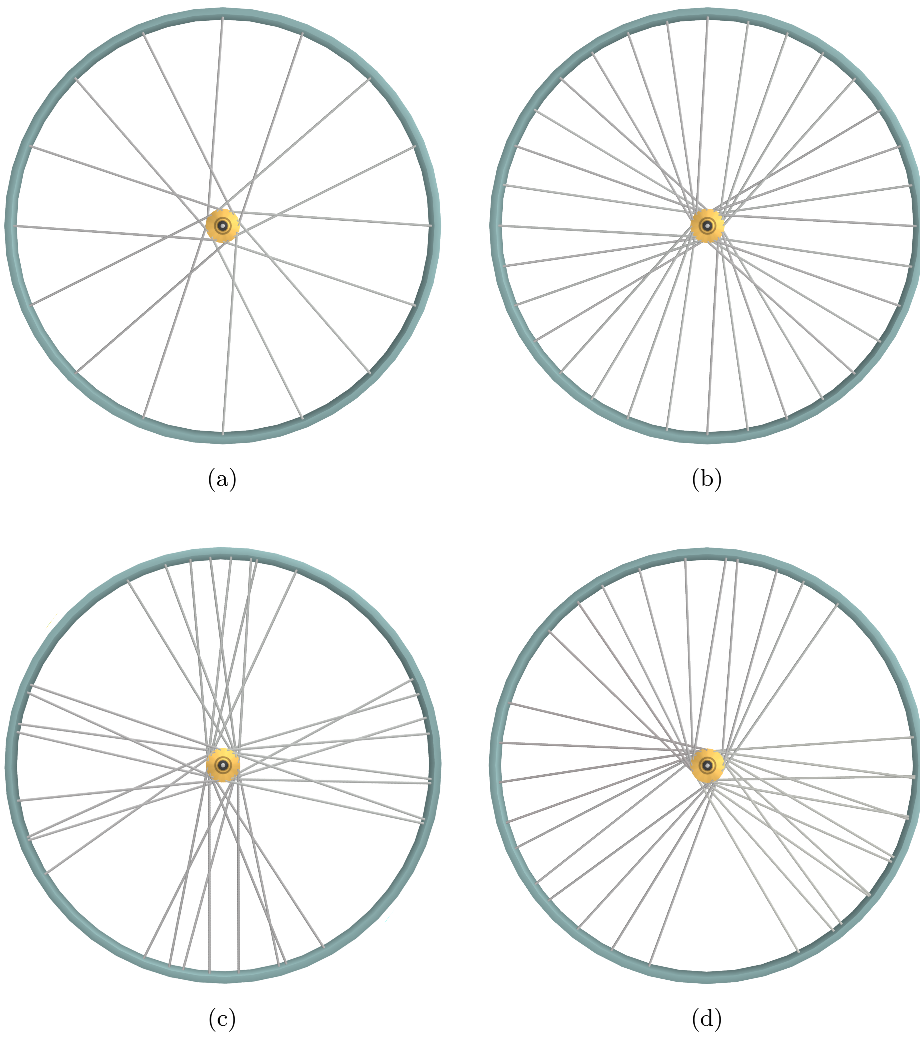


Figure C.1: Some evolved wheels.

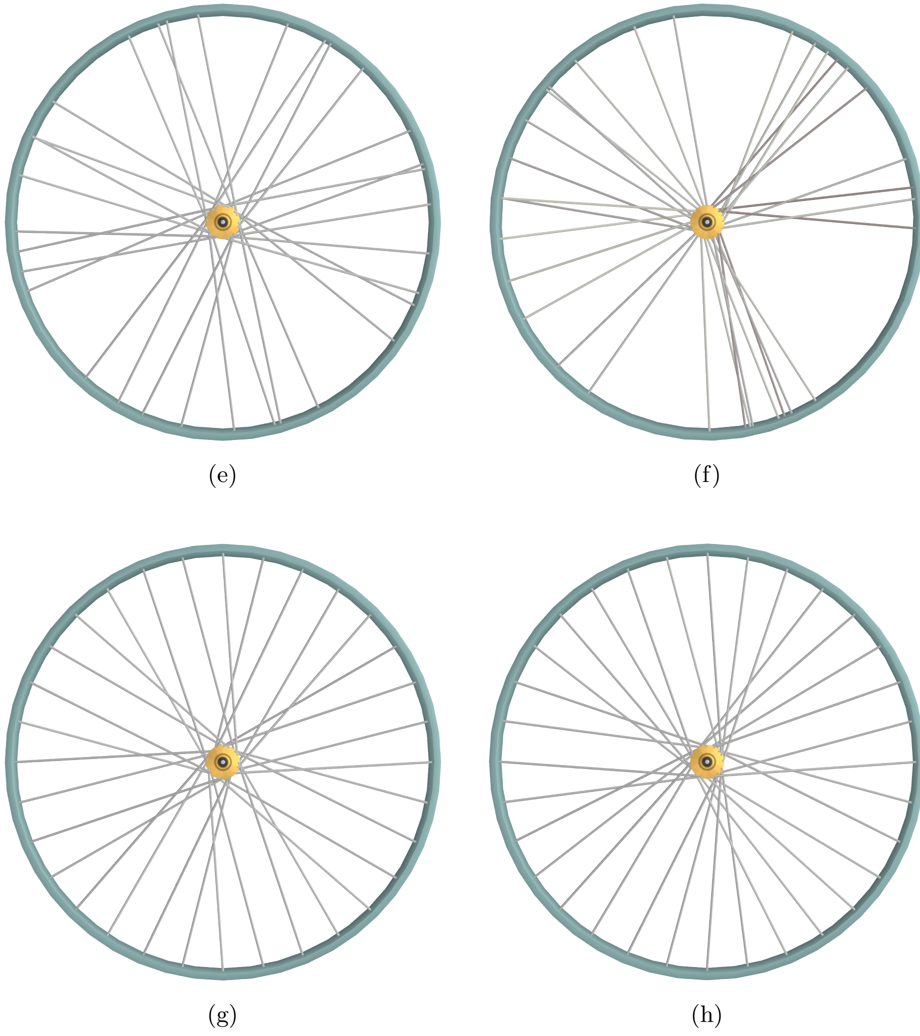


Figure C.1: Some evolved wheels.

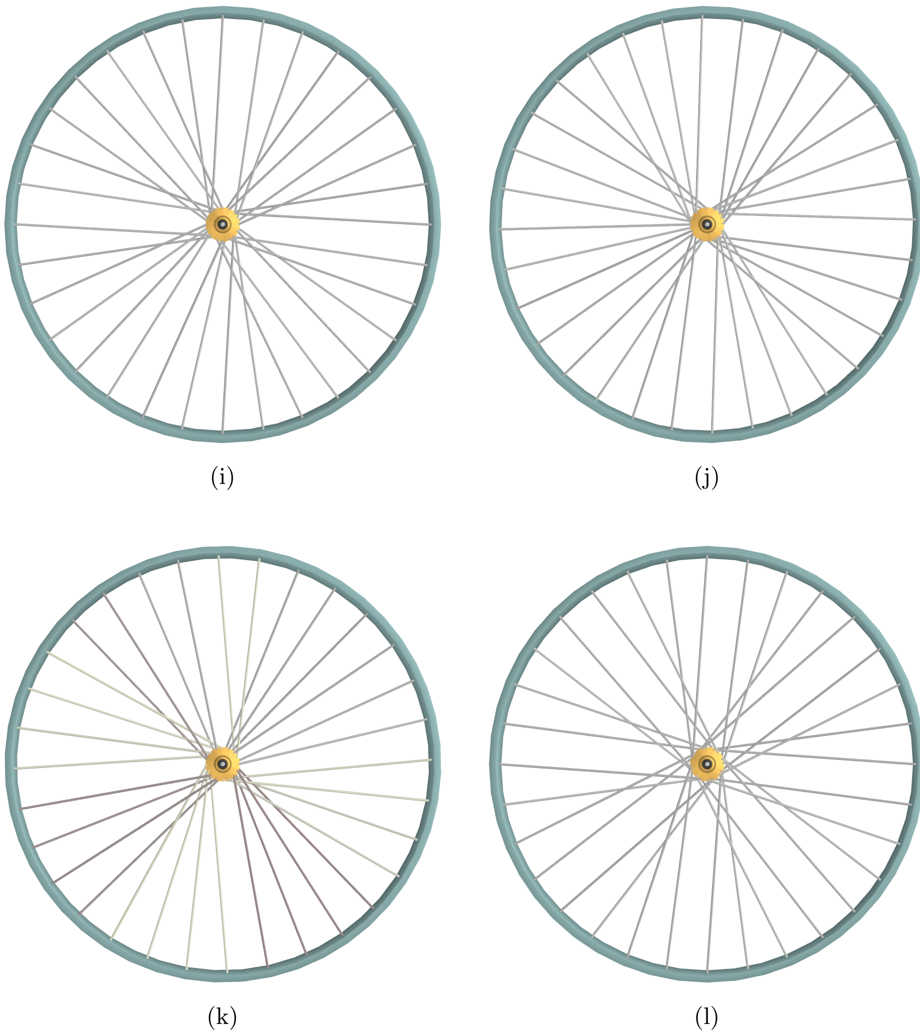


Figure C.1: Some evolved wheels.